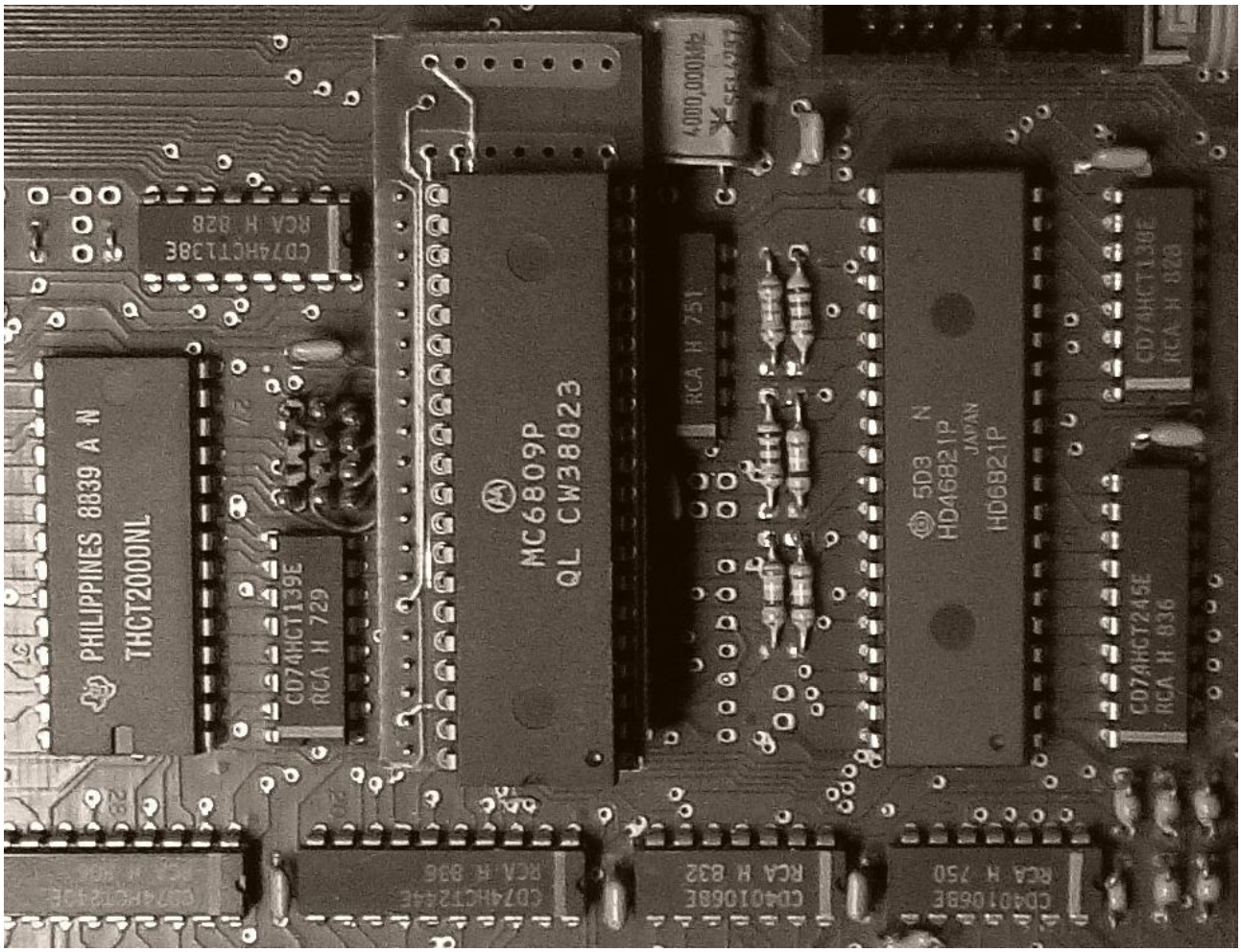
INTRODUZIONE ALLA PROGRAMMAZIONE ASSEMBLY 8 BIT



Volume I: MCS 65XX

Seconda edizione

RetroCampus & RetroProgramming Italia

Introduzione alla (retro)programmazione Assembly 8 bit

di M.A.W. 1968, aka András Vajda

Edizione speciale per il blog `www.valoroso.it`

Prefazioni

Prefazione di Attilio Capuozzo.

Il libro che avete fra le mani rappresenta un prezioso strumento per imparare a programmare in Assembly sulle gloriose macchine a 8 bit dotate di microprocessori (CPU) della famiglia 65xx, il cervello siliceo di iconici computer del passato quali il celeberrimo Commodore 64, suo fratello minore VIC 20, l'Atari 800XL, il BBC Micro e via discorrendo.

Riteniamo, a ragion veduta, che non sia affatto anacronistico proporre un'opera – peraltro completa e professionale – che oggi, nel 3° millennio, insegni la programmazione Assembly su computer degli anni '80 che hanno fatto la storia dell'informatica. Il presente tomo rientra, infatti, a pieno titolo nell'alveo del cosiddetto retrocomputing e, conseguentemente, della retroprogrammazione.

Gli appassionati del retrocomputing sono soprattutto gli ex bambini e gli ex adolescenti che 40 anni fa possedevano quelli che all'epoca venivano definiti "home computer", macchine – basate su un'architettura a 8 bit – che svolsero di fatto il ruolo di volano, di motore propulsore, per l'informatizzazione di massa. Con questi pionieristici e meravigliosi computer, che potevamo collegare anche a un comune apparecchio televisivo oltre che a un più professionale – e costoso – monitor (inizialmente solo monocromatico!), ci giocavamo soprattutto.

Chi non ricorda le indimenticabili e indimenticate cassette che ci avevano abituati a lunghi e spasmodici tempi di attesa per il caricamento di ogni singolo videogioco? Solo pochi fortunati nei primi anni '80 avevano la possibilità di possedere le costose unità drive per leggere i floppy disk con una velocità di caricamento di gran lunga superiore alle succitate cassette. Il settore videoludico fu senza dubbio un aspetto fondamentale per la diffusione degli home computer anni '80. Ed è stato proprio grazie ai giochi che la generazione degli anni '80 si appassionò al mondo della programmazione.

Lo scopo principale di quei ragazzi di 40 anni fa non era solo quello di giocare a quei mitici videogiochi – titoli entrati di diritto nella storia videoludica nonché nell'immaginario collettivo – ma anche capire come era possibile creare queste meraviglie videoludiche. Questo sacro fuoco della passione fece nascere un'intera generazione di programmatori!

Le macchine a 8 bit erano solitamente equipaggiate con un'interprete BASIC, il più famoso e diffuso tra i cosiddetti linguaggi di alto livello. Ma ci si accorse ben presto che il BASIC era sì un linguaggio di programmazione facile da apprendere e piuttosto flessibile ma, purtroppo, poco adatto allo sviluppo dei videogiochi. Al fine di poter creare un videogame degno di questo nome era necessario conoscere e interagire con l'hardware della macchina tramite un linguaggio di basso livello come l'Assembly.

Alcuni di quei ragazzi degli anni '80 riuscirono nell'impresa di studiare, capire e padroneggiare l'Assembly creando – completamente da soli! – giochi o anche programmi di altro genere. Molti altri trovarono, purtroppo, ostico ed eccessivamente impegnativo, oneroso, l'apprendimento dell'Assembly, un linguaggio così lontano dal ben più rassicurante e relativamente facile da apprendere linguaggio BASIC.

Il presente libro si rivolge, dunque, a tutti coloro che in quell'epoca gloriosa, indimenticabile e mitica abbandonarono la sfida dell'Assembly e oggi hanno le conoscenze e soprattutto la passione per imparare finalmente un linguaggio che sfrutti pienamente le potenzialità hardware delle macchine a 8 bit. Ma il target dell'opera che avete appena acquistato comprende, ovviamente, anche coloro che all'epoca avevano imparato a programmare in Assembly e che oggi, a distanza di tanti anni, hanno bisogno per così dire di un "refresh" per rivivere le emozioni programmatorie di 40 anni fa. E infine tra i potenziali lettori non possiamo non menzionare anche i ragazzi di oggi che – onore al merito – sono entrati a far parte della sempre più folta schiera degli amanti del retrocomputing.

A tutti voi auguriamo, dunque, una buona lettura e soprattutto tante liete ore da trascorrere programmando in Assembly il vostro amato 8 bit!

Attilio Capuozzo, Fondatore «Retroprogramming Italia – RP Italia»

Prefazione di Francesco Fiorentini.

Siamo nel 2024 e la parola d'ordine di quest'anno - e, molto probabilmente, degli anni a venire - é sicuramente A.I., Artificial Intelligence o, per dirla in italiano, Intelligenza Artificiale. Gli sforzi di tutte le maggiori realtá tecnologiche mondiali sono concentrati a realizzare sistemi informatici in grado di simulare la capacità e il comportamento del pensiero umano. I linguaggi utilizzati per creare modelli basati sull'intelligenza artificiale o sistemi di machine learning, sono quasi esclusivamente linguaggi di alto livello, cioé con un'elevata astrazione dai dettagli del funzionamento del calcolatore e dalle caratteristiche intrinseche del linguaggio macchina, quindi ben lontani dall'argomento presentato in queste pagine.

Il libro che stringiamo fra le mani é dedicato alla programmazione assembly dei computer ad 8 bit con un occhio di riguardo alle macchine dotate del processore della famiglia 65XX; sí, proprio lui, il cuore pulsamte di alcuni dei piú iconici computer degli anni 80 come il Commodore 64, l'Apple IIe, il VIC-20, il BBC Micro e l'Atari 800XL... Ma anche di console che hanno fatto la storia, come l'Atari 2600. La programmazione Assembly é per antonomasia in antitesi con i linguaggi ad alto livello e, a parte il vero e proprio linguaggio macchina, la cosa piú distante da loro.

La domanda quindi sorge spontanea: "non é anacronistico nell'anno 2024 acquistare, leggere e studiare un libro che descrive i dettagli e le istruzioni di una serie di processori vecchi di 50 anni?". L'Autore, nella sua introduzione, ha giá risposto a questa domanda, argomentando la questione in maniera molto lucida ed io non sono in grado di aggiungere niente di piú dal punto di vista tecnico. Ma posso provare ad aggiungere una nuova dimensione. Molte delle azioni umane, almeno quelle compiute per diletto e non per dovere, sono guidate da un sentimento molto forte che possiamo chiamare passione! Ed é indubbio che chiunque stringa questo libro fra le mani, abbia una passione per il retrocomputing e per la (retro)programmazione.

Quanti di noi negli anni '80, giocando ai videogiochi dell'epoca sui computer ad 8 bit, si sono chiesti almeno una volta come fosse possibile creare quei capolavori videoludici. E quanti di quelli che provarono a cimentarsi nella programmazione realizzarono quasi immediatamente che il linguaggio che accompagnava nativamente queste macchine fosse inesorabilmente limitato rispetto all'obiettivo da raggiungere. I computer dell'epoca erano quasi tutti equipaggiati con un interprete Basic che, nonostante fosse un primo passo importante verso il mondo della programmazione, era insufficiente per creare giochi o applicazioni professionali. Per fare quello occorreva studiare, imparare e padroneggiare il linguaggio Assembly che, giocoforza, era strettamente legato alle specifiche hardware della macchina che lo ospitava.

Alcuni di questi ragazzi riuscirono nell'impresa e, forti della fantasia e dell'intraprendenza che caratterizza le menti giovani, ci lasciarono delle vere e proprie perle. Alcuni esempi sono il 'nostro' Ivan Venturi, che puó essere definito un pionere dell'industria videoludica italiana, oppure il grande Dino Dini che nel 1989 ci regaló l'immenso Kick Off dopo aver iniziato a programmare dall'etá di 13 anni su un Acorn System 1. E come non ricordare David Simons che all'etá di soli 16 anni confezionó una delle piú celebri estensioni Basic per il Commodore 64, quel Simons' Basic che fu accolto in maniera entusiasta da tutta la stampa dell'epoca e dagli utilizzatori di questa stupenda macchina.

Per altri invece il linguaggio Assembly risultó una montagna insormontabile, forse a causa della sua apparente complessitá o forse perché a prima vista ostico da comprendere a differenza del ben piú amichevole Basic. Ebbene, questo libro si rivolge proprio a loro. A quei ragazzi che adesso sono diventati adulti e che forse adesso hanno un lavoro non legato all'infomatica, che tanto li aveva affascinati da adolescenti, ma che hanno conservato dentro il fuoco della passione per i computer che sono stati i compagni di giochi di tanti pomeriggi. Adesso, a distanza di anni, molti di loro hanno allargato le loro conoscenze e possono affrontare di nuovo la sfida dell'Assembly grazie ad un rinnovato bagaglio culturale. Hanno quindi una seconda opportunitá per realizzare il sogno di quando erano bambini: capire come funziona un computer, apprendere il linguaggio che piú di tutti permette di ottenere il massimo dalla macchina che si ha a disposizione e provare a creare quello che 40 anni fa sembrava una chimera.

Francesco Fiorentini, Editor in Chief «Retromagazine»

Prefazione dell'Autore.

Considerando il solo Commodore 64, che con oltre 17 milioni di unità prodotte è indiscutibilmente l'home computer più venduto della storia, abbiamo avuto a che fare con un mercato bibliografico che conta non meno di duecento titoli inerenti la programmazione stampati nell'arco di almeno tre lustri da decine di editori di ogni dimensione, firmati da autori spesso di fama internazionale nel campo dei microcomputer e relativi processori 8 bit. Di questi, almeno quattro dozzine riguardano la programmazione in linguaggio Assembly ai più vari livelli.

Moltiplichiamo tutto questo, con i dovuti fattori di scala, per le innumerevoli altre piattaforme basate sulle medesime CPU 65xx, poi aggiungiamo l'ancora più vasta galassia dei sistemi basati su Z80 (che quasi subito, dai primi anni Ottanta, ha imposto massicciamente la propria presenza anche nel mondo dei sistemi per automazione industriale) e per sovrammisura anche il vasto reame dei 68xx. Quello che si ottiene è un vero e proprio muro di libri di programmazione, con almeno centottanta titoli degni di nota che contengono il termine "Assembly".

A ciò si aggiunga che i relativi diritti editoriali sono quasi sempre scaduti da almeno venti o trenta anni, essendo sovente spariti nel nulla gli editori e risultando comunque definitivamente fuori catalogo i testi. Con minimo sforzo è oggi possibile reperire online le copie digitali di una intera biblioteca di libri di programmazione *low level* delle tre principali CPU a 8 bit nate negli anni Settanta: Z80, 68xx, 65xx.

Con simili premesse, per quale motivo nel terzo millennio qualcuno dovrebbe prendersi la briga di scrivere da zero un testo per principianti sull'Assembly delle CPU a 8 bit ideate più di mezzo secolo fa? Le ragioni, in realtà, sono molteplici: ma la principale risposta viene dalla Rete. Non si poteva restare insensibili alle sempre più numerose richieste da parte di una vastissima base di utenti, che a gran voce richiedevano un corso chiaro, sintetico, schematico, di facile comprensione e scritto in lingua italiana.

Tali richieste si sono dimostrate persistenti e sempre più insistenti nel tempo e nel (cyber)spazio, attraverso FIDOnet, Usenet, i forum, fino agli odierni gruppi FB afferenti all'ampia galassia del Retrocomputing sotto l'egida di RetroCampus & RetroProgramming Italia. Sono richieste decisamente trasversali: ci sono giunte indistintamente da chi sviluppa software da una vita (magari in COBOL su mainframe bancari, o in ABAP su SAP, oppure in PHP per siti web...), da chi ha sempre e solo usato il computer per giocare, da chi è bravissimo col saldatore e la meccanica ma si arena alle prime definizioni algebriche o di modalità di indirizzamento, da chi a suo tempo aveva tentato di imparare l'Assembly ma poi si è dimenticato tutto, da chi voleva programmare giochi e poi si è ritrovato impiegato al Catasto... eppure, nonostante la platea così eterogenea, queste richieste hanno sempre molti elementi in comune su come dovrebbe, e soprattutto come **non** dovrebbe essere strutturato un simile corso.

Tra questi spiccano:

• Una trattazione in lingua italiana, chiara, lineare, senza pretese di esaustività ma non lacunosa;

• Spiegazioni stringate ma non ermetiche, che illustrino al meglio la funzione delle varie sezioni di codice degli esempi anche a chi non mastica codice ogni giorno;

• No alla solita raccolta inconcludente di idiomi Assembly «tipici» (come se fossero ricette regionali...) scorrelati e privi di spiegazioni;

• No a quei testi che contengono una specie di puzzle, i cui tasselli sono in realtà centotrenta subroutine da 20 LOC in media cadauna, e solo dopo avere «unito tutti i puntini numerati nell'ordine corretto» ci si trova in mano un unico «programmone» finale da 2k5 LOC e più;

• No a ubriacature accademiche di teoria astrusa, pretenziosi diagrammi di flusso ISO 5807 che rimangono irrimediabilmente ambigui o continui riferimenti all'hardware;

• No ai testi stagionali usa e getta (una moda allora nascente e mai più abbandonata) troppo specifici per questo o quell'home computer, magari un clone dell'Est venduto col contagocce o il solito improbabile "oggetto di culto" perché tale lo ha decretato un gruppetto di barattatori entusiasti, talmente inconsistente che potrebbero svolgere i propri raduni annuali nell'ascensore di un hotel;

• No ai corsi legati a doppio filo a qualche giurassico Assembler e alle sue macro ultraproprietarie...

Si tratta di un'alchimia estremamente difficile, a prima vista. Soprattutto, queste richieste escludono praticamente la totalità della letteratura importante disponibile in lingua inglese (quella in italiano è *naturaliter* presente in quantità omeopatiche anche sui siti mantenuti con la massima acribia filologica).

Così, nella lunga e quotidiana interazione in veste di Moderatore e Admin con le esigenze esposte dagli utenti, incoraggiato dai colleghi e amici ai vertici delle Associazioni più importanti nell'ambito del Retroprogramming,

l'Autore negli anni Dieci del nuovo millennio ha deciso di affrontare l'improba sfida, seguendo le linee guida sopra tratteggiate.

Non è certo un'iperbole retorica parlare di *sfida*. Ad una prima scrematura, in occasione della preparazione del primo corso a puntate sotto l'egida di RetroCampus & RetroProgramming Italia, l'Autore si è trovato a mettere le mani sull'equivalente di oltre *milleduecento pagine* A4 cumulative di materiale eterogeneo prodotto in prima persona nell'arco di oltre trenta anni sulle sole tre CPU 65xx, 68xx e Z80 tra risposte e BOA FIDOnet, mailing list, forum, post su blog, comunicazioni private, esempi di codice didattici e da suoi progetti, e molto altro. Il lavoro di selezione è stato immane e quasi mai banale: quando esattamente un esempio o un argomento da «semplice» diventa troppo «avanzato» per un corso da principianti? Questa non è che una delle tante possibili forme del *paradosso del sorite*.

Guidato dalle reazioni (spesso entusiastiche, ad onor del vero) alla pubblicazione delle varie puntate del corso, l'Autore ha poi affinato l'idea di condensare e semplificare al massimo in un singolo testo, o meglio in una breve collana, i concetti salienti e l'approccio assolutamente peculiare che caratterizzano la mentalità del programmatore Assembly anni Ottanta.

Partendo dalla piena consapevolezza di rivolgersi ad un pubblico estremamente diversificato, si è cercato un denominatore comune in un costante equilibrio tra semplificazione e correttezza, navigando tra la Scilla della superficialità praticona all'americana in stile «for dummies» e la Cariddi dei testi «avanzati» troppo verticalizzati. Si è costantemente evitato di incentrare il discorso su una profusione di tecnicismi, formule arzigogolate, zuppe di acronimi, riferimenti algebrici e logici troppo astratti, alluvioni di informatica teorica e complessità algoritmica, e soprattutto si è tentato di evitare il più pernicioso degli errori: usare concetti «moderni», oggi comuni, ma drammaticamente errati se applicati a ritroso - del tutto anacronisticamente - a quei progetti.

Vorremmo con ciò rendere ben consapevole il lettore della estrema difficoltà insita nello scrivere - nel terzo millennio - un testo per principianti relativo alle CPU 8 bit e al loro Assembly.

Abbiamo accennato sopra alla vastità della produzione editoriale, che a suo tempo l'Autore, in compagnia di legioni di altri studenti e professionisti, ha affrontato integralmente, nonché alla mole di materiale prodotto in proprio a cui attingere per i contenuti. La bibliografia riportata per le singole CPU è per contro volutamente sfrondata e ridotta ai minimi termini, per evitare di spaventare e confondere inutilmente il lettore con una pletora di riferimenti, spesso oggettivamente introvabili se si vuole parlare di edizioni cartacee. Tuttavia, per la preparazione del corso per le tre CPU e del presente testo sono stati rivisti oltre centocinquanta manuali in tutto, anche e soprattutto per effettuare un controllo incrociato che ha fatto (ri)emergere numerose sviste ed inesattezze, talora piuttosto gravi, in molti libri dell'epoca.

Il che ci porta con decisione al secondo punto: in realtà non solo la bibliografia ma *l'intero testo*, in ogni sua parte, è frutto di una vera e propria sintesi sottrattiva o, se si preferisce, una lunga lavorazione meccanica con abbondante asportazione di sfrido. La selezione degli argomenti, la scelta degli esempi e degli idiomi di codice (tutti originali e rigorosamente testati), la ricerca di una forte coerenza di fondo hanno richiesto tempo e sforzi, al contrario del facile *cut&paste* di tanti siti e blog odierni. L'impalcatura teorica, sebbene faticosamente nascosta e mantenuta minimale, risulta sempre la base e la giustificazione di ogni scelta, al contrario dell'approccio caratterizzato dal pragmatismo superficiale e teatralizzato di quasi tutta la letteratura d'oltre oceano. Il testo è ispirato al noto aforisma del genio di Ulm: «Things should be stated simply, but not simpler than that.».

Tutte le illustrazioni presentate sono state concepite e realizzate dall'Autore. Tutti i problemi, gli algoritmi e gli esempi di codice, dai più banali stralci fino ai sorgenti più articolati, sono stati selezionati, progettati e implementati appositamente per la progressione didattica prevista, dal più semplice al più complesso, cercando costantemente di coniugare ottimalità dell'algoritmo scelto a monte (una premessa semplicemente scontata per chiunque voglia definirsi «programmatore», ma che sempre più spesso è necessario esplicitare), compattezza del codice, prestazioni... ma privilegiando sempre la leggibilità, l'intuitività e la buona strutturazione del codice stesso: per il principiante è quasi impossibile camminare in piena sicurezza sulla sottilissima linea di confine tra ottimizzazione estrema (una vera e propria arte oscura, con queste CPU) e codice spaghetti, pertanto ci siamo tenuti prudentemente assai lontani da tale linea di confine, giungendo appena ad intravederla a distanza forse solo in un paio di occasioni.

Nella speranza che un simile sforzo di selezione, semplificazione e divulgazione non sia stato vano, questo primo volume si propone come pietra miliare di una futura collana editoriale che tratterà varie CPU 8 bit d'epoca.

Nella prima parte del presente testo si illustrano molto brevemente le principali basi comuni ai vari design dell'epoca, passando poi nella seconda parte a mostrare per mezzo di alcuni fondamentali esempi i concetti e gli idiomi di programmazione del MOS 6502 e derivati, in maniera volutamente priva di riferimenti ad alcuna particolare piattaforma, anche se in background l'architettura di riferimento sottintesa è il Commodore 64 in quanto più diffuso home computer in assoluto.

In questa seconda edizione sono state apportate alcune modifiche minori, la sezione relativa all'algebra booleana è stata rivista e ampliata, alcune spiegazioni sono state ulteriormente semplificate e chiarificate, si sono corretti alcuni banali refusi e aggiunte ulteriori illustrazioni, tabelle ed esempi di codice.

L'Autore desidera sentitamente ringraziare tutto lo staff «Retroprogramming Italia - RP Italia» e «RetroCampus», nonché la Redazione di «Retromagazine» e alcuni amici in particolare, rigorosamente in ordine lessicografico ascendente di cognome:

- **Attilio Capuozzo**, Presidente e Fondatore di «Retroprogramming Italia - RP Italia»
- **Francesco Fiorentini**, Editor in Chief di «Retromagazine»
- **Fabrizio Lodi**, Presidente di «RetroCampus»

per il loro continuo impegno e supporto nella diffusione della cultura tecnoscientifica legata all'affascinante galassia del Retrocomputing, e i primi due anche per avere impreziosito questa seconda edizione con le loro graditissime Prefazioni.

In omaggio al «paradosso della prefazione» del logico D. C. Makinson [Mak65], si ricorda che il presente lavoro contiene sicuramente *almeno un errore*.

About the Author...

Nato nel 1968, l'Autore appartiene a quella generazione che ha avuto la fortuna di vivere la propria adolescenza negli irripetibili anni Ottanta: in particolare quella minoranza di *nerd* e *geek* (la vera e sola «generazione digitale») che è stata protagonista assoluta fin dal primo momento dell'avvento degli home computer e della rivoluzione del personal computing, vivendone l'intera storia da fenomeno di nicchia a mainstream, e da passione giovanile a professione.
In estrema sintesi, l'Autore ha attraversato, quasi sempre con ruoli di spicco e di responsabilità, tutte le stagioni della IT e della telematica: dalle BBS a FIDOnet, a Usenet, alle mailing list, ai primi portali tecnici, ai forum, fino agli odierni user groups, condividendo con entusiasmo parte del suo *know-how* matematico, logico, tecnoscientifico - prima come studente STEM e poi come professionista impegnato in una lunga carriera di ricerca e sviluppo a livelli multinazionali nel campo dei sistemi embedded distribuiti e relativi RTOS per applicazioni altamente critiche in vari settori del *vehicle engineering*. Laddove molti scrivevano prevalentemente per porre domande, l'Autore aveva già affrontato quei testi e quelle esperienze che gli consentivano di fornire la maggior parte delle risposte corrette e circostanziate, e di creare bibliografie che sono rimaste un riferimento fino ai giorni nostri.
Produrre in questa sede interminabili elenchi dei numerosi linguaggi, librerie, framework, paradigmi, sistemi, hardware, ambienti di sviluppo e tecnologie (quasi tutti esotici, qualcuno decisamente esoterico) padroneggiati dall'Autore sarebbe del tutto sterile e irrilevante. Vale invece forse la pena di soffermarsi brevemente su una singola curiosità: quell'acronimo M.A.W. 1968, ossia Master Assembly Wizard, che da almeno sette lustri è il nickname dell'Autore. In un'era in cui molti, se non tutti, lavoravano in Assembly era già stato soprannominato dai colleghi «Assembly Wizard», in riconoscimento della sua prolificità e competenza su un vasto range di piattaforme. Qualche anno dopo, nei primi Novanta, qualcuno ha iniziato sempre più insistentemente a definirlo anche Master, specialmente dopo l'annuncio ufficiale del raggiungimento della milionesima linea di codice nello sviluppo di progetti Assembly (superiori ai 10 kLOC) per un totale di quarantasei costruttori, centosettantatre famiglie e circa seicento distinti modelli di MCU, CPU e DSP - dai più ovvi Motorola, Zilog, Rockwell, Intel, Hitachi, Mitsubishi, Fujitsu, Atmel, Microchip, SGS... fino a NEC, Holtek, Siemens e molti ASIC specifici.
Oggi, trascorsi ormai più di quaranta anni dai giorni gloriosi del primo Commodore 64 dal quale tutto ha avuto origine, la passione inalterata dell'Autore per il retrocomputing pionieristico e il suo costantemente attivo impegno in questo settore sono certamente suffragati da nuove e ben più ampie conoscenze, da una maturità professionale e culturale che dona una visione d'insieme e retrospettiva decisamente più ampia, ma restano sempre accompagnati da quel genuino stupore e amore per la scoperta che caratterizzavano l'approccio dei primi tempi.

Le immagini mostrano una parte della collezione privata dell'Autore, utilizzata anche per il collaudo degli esempi Assembly del presente testo.

Ho avuto il piacere di contattare Amedeo nel 2021, per discutere principalmente di domotica. L'intesa che si è creata, in un clima di reciproca stima e fiducia, è stata tale che il discorso si è rapidamente allargato alle numerose passioni e attività che ci accomunano: su tutte il retrocomputing e la retroprogrammazione, naturalmente.
L'attività di Amedeo su numerosi fronti della divulgazione e nel supporto all'organizzazione di Varese Retrocomputing non ha certo bisogno di presentazioni: moltissimi, come me, lo conoscono grazie alle sue attività, e come me ne apprezzano la preparazione tecnica tanto quanto le doti umane e grande passione. Per questo motivo ho ritenuto giusto e opportuno creare la presente versione personalizzata per i suoi canali social e i suoi numerosi follower di questa riedizione celebrativa del ventennale di un corso introduttivo che a suo tempo ha riscosso molto interesse, e che oggi sperabilmente potrà essere apprezzato anche dal suo vasto pubblico.

Parte I

Introduzione alla programmazione Assembly 8 bit.

Capitolo 1

Introduzione.

Il presente testo è esplicitamente e specificamente rivolto a principianti della retroprogrammazione in Assembly, possibilmente con pregressa esperienza nell'uso di linguaggi di alto livello. Si cercherà quindi di mantenere elementare il livello della trattazione, mirando alla massima chiarezza e rinunciando a priori a qualsivoglia eccesso di astrazione, rigore formale, riferimenti teorici non strettamente indispensabili.

La programmazione in linguaggio Assembly a 8 bit non assomiglia ad alcuno dei compiti che il lettore può avere fin qui affrontato. Su una nota enciclopedia online, alla voce dedicata all'Assembly, si legge (non senza un ampio sorriso) che qualche ingenuo anonimo estensore ha ritenuto necessario specificare che il linguaggio Assembly «non offre alcun controllo sui tipi». Ciò equivale a dire che, entrando in un grande autosalone, potremmo trovare un vistoso cartello che con serietà ci redarguisce: «Attenzione! Le autovetture in vendita in questa concessionaria non sono omologate per la pesca d'altura.».

Umorismo involontario a parte, un file sorgente in linguaggio Assembly si riduce sostanzialmente ad una **sequenza di mnemonici** di poche lettere (in corrispondenza biunivoca con le *istruzioni* vere e proprie) eventualmente seguiti da **operandi** (i *dati* o *indirizzi* su cui dette istruzioni operano) laddove previsti. Si lavora esclusivamente su **registri** e **locazioni di memoria** (spesso individuate da apposite *label*, ossia etichette); anche i cicli e le scelte, ossia le più elementari operazioni previste dal teorema di Böhm-Jacopini [BJ66] che sancisce le caratteristiche minimali di un qualsiasi linguaggio di programmazione Turing-equivalente, vengono implementate in modo implicito e decisamente criptico per l'occhio di un programmatore di alto livello.

Il risultato finale, dopo il cosiddetto **assemblaggio** (l'equivalente della *compilazione* per i linguaggi HLL) che avviene tramite apposito programma Assembler, non è altro che una lunga sequenza di *valori numerici naturali*, ossia il codice eseguibile binario vero e proprio, nel quale sono intercalati secondo un preciso schema le istruzioni e i relativi dati.

Si tratta di un approccio unico, a stretto contatto con una mole notevole di dettagli hardware, anche per i più «semplici» microprocessori a 8 bit con set di istruzioni RISC[1]. Un approccio che richiede una costante attenzione agli aspetti di ottimizzazione ed efficienza, a maggior ragione quando si parla di retroprogrammazione (o, ancora oggi, sui sistemi embedded con core a 4 e 8 bit di dataword).

Per la massima efficacia e qualità dei risultati, il percorso di studio individuale deve necessariamente seguire la consueta progressione:

1. **Architettura hardware** della piattaforma;
2. Il **linguaggio macchina** vero e proprio per la piattaforma d'interesse;
3. I vari **Assembler**, con le loro idiosincrasie sintattiche e sistemi di macro profondamente incompatibili gli uni con gli altri (inclusi i moderni ambienti di *cross-development*, ormai sempre più necessari);
4. La programmazione dei singoli **chip periferici** specializzati;
5. Last, but not least, **le applicazioni**: studiando e ristudiando il codice applicativo che ormai si trova in giro, tra disassemblati e rilasci al pubblico dominio, in quantità esorbitanti.

L'assioma fondamentale per la programmazione low level a 8/16 bit è che si impara leggendo **molto** codice Assembly avanzato scritto da programmatori professionisti e ben preparati: lo studio della mediocrità, come soleva dire il grandissimo critico d'arte Harold Bloom, può solamente generare ulteriore mediocrità. A sua volta, comprendere nei dettagli tale codice richiede lo studio di svariate tipologie di testi, senza limitarsi al solo Assembly che è il punto di arrivo.

[1] RISC è acronimo di Reduced Instruction Set Computer, opposto a CISC (Complex Instruction Set Computer). In realtà applicare tali schemi alle CPU di nostro interesse, nate in prevalenza negli anni Settanta quando i criteri progettuali dei semiconduttori programmabili erano appena agli albori, richiede come minimo una interpretazione estensiva delle definizioni.

1.1 Gli strumenti di lavoro.

Una dotazione oggigiorno davvero minimale per sviluppare in Assembly consiste nell'uso di un home computer d'epoca e di un programma assemblatore[2]. All'epoca tale dotazione poteva (e doveva, se si voleva fare sul serio) essere estesa con una **cartridge** dotata di monitor/debugger e, nei casi migliori, con un **emulatore** di CPU (ad esempio il potente HP 64000, col quale l'Autore ha lavorato per decenni). Ragioni piuttosto ovvie di praticità ed efficienza inducono oggi a prediligere l'uso di **simulatori**[3] e **cross-assembler** per PC.

1.2 Perché studiare l'Assembly?

Si presume che il lettore del presente testo sia già in qualche modo motivato all'apprendimento del linguaggio Assembly, per ragioni principalmente ludiche e di passione per il retrocomputing. I fondamentali vantaggi dell'Assembly nell'ambito home e SOHO («Small Office - Home Office», in sostanza il vasto mercato delle piccole imprese e dei professionisti) sono comunque ben noti dagli anni Settanta dello scorso secolo:

Velocità. Sulle piattaforme di home computing degli anni Ottanta, taluni risultati prestazionali sono notoriamente ottenibili **solo e unicamente** con un oculato ricorso all'Assembly **accuratamente ottimizzato**.

Compattezza. Un assioma fondamentale dell'elettronica recita che lo spazio sulle EPROM e ROM non è mai sufficiente. Quasi inutile sottolineare che il codice più compatto in assoluto, con il minore footprint possibile rispetto a quanto ottenibile con qualsiasi HLL, si può ottenere unicamente lavorando in Assembly.

Conoscenza. Lo studio del linguaggio Assembly «costringe» ad una più approfondita conoscenza hardware, del firmware di sistema, dei protocolli di I/O. Ne scaturisce quindi un generale miglioramento della qualità della programmazione e delle competenze generali.

Interfacciamento. Lavorare a basso livello, a contatto con l'hardware, porta rapidamente a poter sviluppare software di interfacciamento e controllo di hardware e sistemi esterni efficiente ed ottimizzato sotto ogni aspetto.

Vi è anche di più. Studiare l'Assembly a 8 (e 16) bit ha un valore professionale ancora attuale per la programmazione di sistemi embedded. La maggioranza dei core odierni a 8 e 16 bit, infatti, è ancora basata sulle CPU e MCU classiche degli anni '80: 6800 e 6502 (tra loro fortemente simili), Z80 e Z180, 8051, 80186. Il motivo è molto semplice: nonostante alcune ingenuità nella progettazione (i metodi formali e le algebre di processo per la progettazione razionale dei set di istruzioni erano ancora di là da venire!), questi core possono vantare **centinaia o migliaia di miliardi di ore** di funzionamento sul campo in milioni di applicazioni eterogenee, e quindi offrono il requisito dell'affidabilità che, nel mondo embedded, vale molto più di altre caratteristiche. Tutto ciò senza voler magicamente trasformare il lettore in un esperto di sistemi embedded, metodi formali e normative cogenti (occorrono anni di studi specialistici e certificazioni estremamente selettive!), ma chiarendo che nei livelli di base dell'industrial automation e della domotica, per tacere del DIY, c'è ancora ampissimo spazio per questo genere di programmazione su core di derivazione tradizionale, che sono centinaia: dai Rabbit (Z180) a Tezzaron Semiconductors che offre cloni 8051 capaci di operare fino a ben 300 MIPS.

[2] Ad esempio, per il Commodore 64: Turbo Assembler, Commodore Assembler, Merlin 64, Panther 64, Zeus 64...

[3] Per acribia filologica, è bene notare che i termini «emulatore» e «simulatore» usati con leggerezza nel mondo della virtualizzazione non sono in alcun modo sinonimi, né intercambiabili. Se si clicca su «aggiungi al carrello» sulla pagina di un sito di e-commerce che descrive un **emulatore ICE** (In-Circuit **Emulator**) sarà recapitato a domicilio un oggetto fisico piuttosto ingombrante e decisamente non a buon mercato. L'*emulatore* è solo e unicamente l'apparato **fisico** che da un lato si interfaccia al PC tramite seriale, parallela, USB, Firewire... e dall'altro viene fisicamente inserito, tramite un apposito *pod* con un *transition socket*, su una qualsiasi scheda elettronica, nel socket che normalmente ospita la MCU o CPU (o SoC, DSP, ASIC, FPGA...); in altri casi, tale apparato si interfaccia fisicamente al device sotto test (DUT) tramite un idoneo connettore e un semplice cavo di debug ed emulazione a standard specifico (tipicamente JTAG, oppure strettamente proprietario).

Viceversa, si dice propriamente e unicamente *simulatore* il software che crea una macchina virtuale simile in tutto ad una data piattaforma hardware, comprensiva di SO e periferiche simulate: che sia un Commodore 64 o piuttosto una SPARCstation o un HP Apollo.

Capitolo 2

Brevissimi cenni di storia del calcolo automatico.

La storia dei moderni calcolatori inizia, di fatto, con Gottfried Wilhelm von Leibniz (Lipsia 1646 - Hannover 1716). Egli è considerato il **padre della scienza informatica** non solo a causa dell'invenzione del sistema di numerazione binario, basata sui suoi studi dell'antico sistema combinatorio cinese dell'I-Ching, ma anche per i suoi ampi studi in logica e per numerose intuizioni e anticipazioni che lo collocano all'origine del grande filone di pensiero della scienza computazionale.

Leibniz è infatti *il vero padre della logica moderna* in un senso tecnico molto preciso, che qui possiamo riassumere in maniera estremamente concisa per grandi temi:

- Ha ideato il sistema binario, che diventerà fondamento stesso della logica matematica grazie ai successivi sforzi di George Boole (1815-1864) e Augustus De Morgan (1806-1871);
- Ha compiuto estesi studi dell'*ars combinatoria*, madre e fondamento della matematica discreta;
- Ha recuperato in modo compiuto Aristotele e la sua matematizzazione della logica, che dopo Boole sarà oggetto di intenso lavoro per decine di logici e matematici di primissimo ordine in tutto il resto dell'Ottocento e soprattutto nel Novecento;
- Ha genialmente anticipato la semantica dei "mondi possibili" con la sua idea di "ottimismo", ossia il concetto che grazie al piano divino viviamo nel "migliore dei mondi possibili". In logica moderna, infatti, una proposizione è considerata vera se è sempre vera la sua interpretazione in qualsiasi "mondo" possibile, ovvero entro qualsiasi sistema assiomatico considerato.

Una vasta comunità dei più influenti nomi nell'informatica e nella filosofia della matematica odierne (tra i quali gli informatici Gregory Chaitin e Stephen Wolfram, il fisico Paul Davies, il giovane filosofo Ugo Pagallo...) ha ufficialmente patrocinato il riconoscimento di Leibniz come padre della logica e della computazione, precursore di una linea di pensiero ancora attualissima e centrale nella filosofia e nella scienza del calcolo.

Dopo Leibniz, merita una menzione incidentale anche la vicenda ottocentesca della «strana coppia» costituita da Lady Ada Augusta, contessa di Lovelace (unica figlia legittima del noto poeta Lord Byron) e Charles Babbage: vicenda che, per i suoi presunti risvolti romantici e per i conflitti ideologici[1] che ancora suscita, occupa spesso ampi spazi nei testi di storia del calcolo automatico. Tuttavia, all'atto pratico e gossip a parte, la mostruosa Macchina Analitica a vapore (per la quale Babbage è ricordato come padre putativo dei moderni sistemi di calcolo) non è mai stata realizzata[2].

[1] Il fatto che esistano ben sei diverse biografie di Ada, un paio delle quali decisamente poco lusinghiere nei suoi confronti, è un importante indicatore del dibattito e degli interessi anche ideologici animati dalla sua figura storica, riaccesisi in tempi recenti: nel 2015 ricorreva infatti il bicentenario della nascita di quella che è considerata la prima programmatrice della storia per la sua traduzione annotata (`http://www.fourmilab.ch/babbage/sketch.html`) di un articolo in francese di Luigi Federico Menabrea, noto scienziato italiano e allievo di una vera gloria italica ottocentesca, l'ingegnere e matematico Giorgio Bidone, studioso di idrostatica e fluidodinamica di fama internazionale. Tale articolo riassumeva l'intervento dello stesso Babbage a Torino durante il secondo congresso degli scienziati italiani, nel quale descriveva e illustrava i principi del suo progetto di Macchina Analitica, un mostro meccanico composto da molte più parti di una moderna autovettura e azionato a vapore per il calcolo «error-free» di tavole numeriche con la creazione diretta dei relativi cliché di stampa. Tra le note aggiunte alla traduzione, Ada aveva sviluppato una vera e propria procedura esemplificativa per il calcolo dei numeri di Bernoulli. Si veda anche l'articolo «Happy birthday, Lady Ada» in questa raccolta: `https://n9.cl/znahe`.

[2] Invece il successivo progetto di Macchina alle Differenze n° II, concepito da Babbage nel triennio 1847-49, è stato costruito... sebbene oltre un secolo dopo, da un gruppo di ricercatori dello Science Museum di Londra, seguendo fedelmente i disegni e gli appunti originali. La prima parte è stata completata nel 1991, per il bicentenario della nascita di Babbage: la sezione di stampa è stata ultimata nel 2000. La macchina, che si compone di circa *ottomila* parti meccaniche ed è azionata a manovella, funziona perfettamente: i materiali e le tolleranze sono stati tenuti accuratamente conformi a quanto sarebbe stato possibile realizzare

Altra doverosa menzione per un giovane ingegnere minerario, anch'egli ottocentesco: tale Herman Hollerith. Come primo modesto impiego prende parte al censimento decennale USA del 1880. La fatica di quell'immane lavoro amanuense e i rischi di errori sono tali che il giovane (rivelandosi dotato di quella forma di geniale «pigrizia» che caratterizza il vero informatico: **non reinventare la ruota!**) inizia a pensare ossessivamente ad un modo per meccanizzare le operazioni. Nel 1889 si presenta al concorso indetto per il censimento dell'anno successivo con una classificatrice elettromeccanica che consentirà di elaborare i dati in «soli» tre mesi anziché nei cinque-sei anni mediamente richiesti per l'elaborazione manuale dei milioni di schede raccolte, con un risparmio di oltre cinque milioni di dollari dell'epoca. L'idea di H. H. è quella di usare le schede perforate del telaio Jacquard, usando quella che ancor oggi - non a caso! - chiamiamo «codifica Hollerith» per memorizzare i dati e alimentando in automatico le schede nella tabulatrice-classificatrice meccanizzata. Con i denari del censimento, questo ingegnoso signore fonda una piccola società di costruzioni elettromeccaniche, denominata Tabulating Machine Company: un nome oscuro, destinato a rimanere sepolto nei polverosi archivi della storia. Almeno fino a quando la società, tramite una serie di acquisizioni e fusioni strategiche perfezionate nel 1911, cambierà definitivamente nome (nel 1924) in International Business Machines Company, forse meglio nota con l'acronimo IBM.

Dobbiamo però arrivare agli anni Trenta dello scorso secolo per poter completare il quadro teorico iniziato dagli studi di Leibniz e perfezionato dalla coppia ottocentesca Boole-De Morgan. In quel decennio tre colossi del pensiero logico-matematico pongono le ultime, definitive basi teoriche per la nascita dei moderni calcolatori:

- Il britannico Alan Mathison Turing (1912-1954) concepisce un completo **modello matematico** di calcolatore universale, oggi noto come Macchina di Turing (MdT);
- A breve, il logico Alonso Church (1903-1995) aggiunge alla teoria altri tipi equivalenti di sistemi di calcolo, creando quella che oggi si chiama **tesi di Church-Turing** e riguarda una vasta serie di formalismi tra loro equivalenti in grado di elaborare qualsiasi algoritmo che risulti «praticamente computabile»[3];
- L'ingegnere americano Claude E. Shannon (1916-2001) dimostra nella sua tesi di laurea (1937) che circuiti elettronici arbitrariamente complicati, se progettati seguendo la logica booleana, sono in grado di effettuare qualsiasi calcolo binario ed elaborazione simbolica, grazie a semplici ma efficaci **codifiche** che riconducono qualsiasi simbolo ad un valore numerico. Assieme con la tesi di Church-Turing, ciò costituisce il fondamento teorico della scienza del calcolo, e dimostra la possibilità di costruire calcolatori in grado di svolgere ogni tipo di elaborazione, non solo numerica.

A questa nota terna di risultati, che spalanca le porte anche alla nascente teoria delle funzioni ricorsive (modernamente denominata **teoria della calcolabilità**), parte essenziale di quella branca della logica poi denominata *informatica teorica*, occorre aggiungerne uno troppo spesso dimenticato, assieme al nome del suo ideatore: il logico polacco Jan Łukasiewicz (1878-1956) il quale viene erroneamente (e per giunta solo raramente) ricordato come inventore della RPN[4], mentre uno dei suoi risultati di gran lunga più importanti è l'ideazione della **logica trivalente**, nella quale viene modificato un assioma fondamentale della logica classica, detto «principio del terzo escluso» (*tertium non datur*). Tale logica, facilmente estesa già dai suoi contemporanei alla creazione di sistemi logici **polivalenti**[5], trova diretta applicazione nella progettazione delle porte logiche *three-state* o 3-state, nelle quali le uscite possono assumere, oltre ai normali livelli H ed L (corrispondenti alle costanti logiche Vero e Falso, 1 e 0) anche un **terzo stato** di elevata impedenza denominato *Hi-Z*, che nel caso specifico serve ad evitare conflitti elettrici su un bus al quale sono connesse staticamente più uscite.

---

all'epoca della progettazione. I meriti di Charles Babbage non si limitano comunque a tali progetti: il lettore interessato è caldamente invitato a consultare [Dub78] per una visione d'insieme dei suoi contributi matematici.

[3] La definizione originale è necessariamente ampia e volutamente vaga: tale rimane anche dopo decenni di ricerca logica in tale senso. Non a caso si tratta di una tesi, non di un teorema.

[4] RPN è l'acronimo di Reverse Polish Notation, una notazione postfissa per le formule che ha mantenuto una certa utilità in ambito informatico, in quanto semplifica la stesura di parser, oltre ad essere molto amata dagli entusiasti delle calcolatrici tascabili programmabili di HP. Tuttavia, Łukasiewicz ha in effetti ideato la cosiddetta notazione polacca (PN), strettamente simile alla precedente ma che è invece *prefissa*, per eliminare la necessità delle parentesi e semplificare così la notazione nel calcolo proposizionale: un'idea ad oggi largamente desueta nell'ambito della logica formale. Ritroviamo invece la RPN nella implementazione dell'ALU (Arithmetical-Logical Unit) di molte CPU, incluse quelle di nostro interesse, per la sua inerente semplicità implementativa a livello di circuiti logici sequenziali e di RTL (Register-Transfer Logic). Il suggerimento di adottare tale sequenza postfissa nella dinamica funzionale delle CPU si deve al noto e poliedrico matematico John Von Neumann (1903-1957), che ritroveremo tra qualche paragrafo tra i protagonisti della storia del calcolo automatico.

[5] I più diffusi e utili sistemi logici polivalenti sono quelli che ammettono *tre* valori (es. Vero, Indeterminato, Falso) e *quattro* valori (ad esempio: Vero, Indeterminato, Non-specificato, Falso). Vale solo la pena di notare che la prima tipologia di sistemi logici si basa su un ordine totale (catena) di valori, perché Indeterminato è chiaramente «minore» di Vero e «maggiore» di Falso, intuitivamente: ma la seconda famiglia di logiche prevede un *ordine parziale* delle costanti, in quanto (se resta chiaro che i valori estremali sono sempre Vero e Falso) i due valori intermedi non sono tra loro comparabili. Ecco quindi un ordine parziale finito il cui diagramma di Hasse è il tipico rombo.

L'idea delle logiche polivalenti, portata ai suoi limiti, ha generato a sua volta la moderna logica *fuzzy*, nella quale si manipola un'infinità numerabile (o superiore) di valori logici.

2.1 Analogico o digitale?

Il fatto che in letteratura si parli universalmente di **logica binaria** o booleana implica chiaramente che l'oggetto dell'attenzione generale sono i *calcolatori digitali*, caratterizzati dall'uso di soli due valori (bit = **bi**nary digi**t**) associati semplicemente alla presenza o assenza (entro soglie molto ristrette) di una data tensione elettrica.

L'uso di sistemi analogici, nei quali sia gli input che i risultati dell'elaborazione sono invece tutti i possibili *livelli di tensione* continui o alternati in un dato range, sebbene valutato e tentato praticamente dagli albori della storia dei calcolatori automatici, ha avuto una genesi assai travagliata e una brevissima storia applicativa. Storia terminata con una vera e propria *damnatio memoriae* e la reclusione in un ristrettissimo settore dell'automazione industriale e dei sistemi di controllo per processi continui, per non dire di applicazioni ancora più esotiche note (ed utili) ad un numero di specialisti così ristretto che, a livello globale, sarebbe già difficile mettere insieme una squadra di basket o un paio di tavoli di poker riunendoli tutti.

Dopo numerose false partenze e costosissimi tentativi fallimentari nei decenni pionieristici della ricerca accademica (spesso usati come esempio negativo nei corsi di ingegneria dei sistemi), il quarto d'ora di gloria di tali sistemi inizia a fine anni Sessanta con la massiccia disponibilità commerciale di circuiti integrati analogici (principalmente amplificatori operazionali e loro varianti o configurazioni integrate: Norton/CFA, differenziali, comparatori...) con banda passante ampia ed elevato CMRR, relativamente immuni al rumore per l'epoca e quindi in grado di renderne minimamente affidabile il funzionamento. La breve parabola termina poco più di un decennio dopo, quando il raggiungimento di velocità di clock dell'ordine del MHz per i calcolatori digitali fa venire definitivamente meno la ragione dell'adozione di un calcolatore analogico, ossia la velocità di risposta relativamente elevata. Trascorso un ulteriore decennio, chiusi definitivamente da tempo i grandi centri di calcolo analogici che avevano caratterizzato gli anni Settanta, anche nel mondo dell'automazione industriale (ultimo baluardo per tali sistemi) il calcolatore analogico era definitivamente diventato un sottosistema completamente integrato (inizialmente con tecnologie ibride a film spesso, poi massicciamente sotto forma di System-on-Chip[6]) e infine un vero e proprio componente o sottosistema on-chip all'interno di DSP[7] avanzati, motion controller, process supervisor eccetera. Soprattutto, dai primissimi anni Novanta il sottosistema in questione era diventato *programmabile* nel senso pieno del termine, ossia con programma memorizzato: un senso che evidentemente non implica lo spostamento manuale di ponticelli di filo, l'aggiunta o rimozione *a caldo* (hot-swap) di schede dal sistema, l'uso di matrici di diodi e interruttori o la rotazione manuale di potenziometri, selettori e *contraves* (esattamente in tal modo si svolgeva la «programmazione» del tipico calcolatore analogico degli anni Settanta, ovvero ricablando e riconfigurando fisicamente il sistema!). Pertanto, quando si parla di calcolatori automatici, s'intende da decenni per default e senza ulteriori perifrasi riferirsi unicamente al *calcolo digitale*.

2.2 Harvard vs Von Neumann.

Dopo la parentesi sulla (breve) storia dei calcolatori analogici, torniamo al discorso principale. Dopo i prototipi indipendenti Z3 (interamente costruito dall'ingegnere tedesco Konrad Zuse con relais telefonici di scarto, come i suoi due predecessori Z1 e Z2), ultimato nel 1941, e ABC di Atanasoff (1942) si sono succeduti numerosi progetti fallimentari, come Whirlwind del MIT (finanziato dalla USAF) iniziato come calcolatore analogico e poi ultimato fuori tempo massimo tra il 1949 e il 1951, dopo un cambio di destinazione in corso d'opera, quando i militari avevano ormai perso interesse in quel tipo di realizzazione. Nel 1944 vede la luce ad Harvard (USA), sotto la guida del fisico Howard Aiken, il già obsoleto Mark I, macchina a relè costruita con l'apporto di IBM, che ne aveva iniziato il progetto a fine anni Trenta con la sigla ASCC. Tra il 1942 e il 1946 viene costruito ENIAC presso l'università della Pennsylvania, a cura di John Mauchly e ispirato ad ABC. In ambedue i casi parliamo di macchine il cui tasso di inoperatività a causa di guasti supera il 90%, e con il grave difetto di non essere programmabili. La configurazione di una nuova procedura di calcolo richiedeva di fatto un ricablaggio della macchina, risultando così in scarsissima flessibilità ed elevatissimi costi operativi.

Nel giro di poco tempo viene però completato un successore di ENIAC, denominato EDVAC, che risulta invece più facilmente programmabile e offre maggiori velocità. Il geniale matematico di origine ungherese John von Neumann, trasferitosi a Princeton, trascorrerà molto tempo attorno ad ambedue le macchine, scrivendo poi

[6] Un System-on-Chip (SoC) è un circuito integrato che riunisce internamente su uno o più chip VLSI tutti i componenti tipici di un completo sottosistema di acquisizione e controllo, incluso il condizionamento dei segnali analogici, le uscite analogiche e ogni altro elemento integrabile, compresi quelli di media potenza (es. FET, driver, regolatori di tensione, etc.) oltre ai più tipici componenti di un sistema di elaborazione digitale: MCU, memorie on-chip, canali di I/O seriali e paralleli, altro. Oltre alla programmazione tramite firmware vero e proprio destinato all'esecuzione da parte della/e MCU, il device deve essere anche opportunamente *configurato* (di solito in modo non volatile) a livello di interconnessioni tra i vari stadi ed elementi interni, sia analogici che digitali, normalmente organizzati in matrici e blocchi.

[7] DSP (Digital Signal Processor) è una CPU specificamente dedicata all'elaborazione di segnali e al relativo calcolo numerico, con caratteristiche spesso molto esotiche e architetture portate al loro limite concettuale, come la SHARC (Super-Harvard ARChitecture) che tra le tante features avveniristiche rispetto alle CPU mainstream prevede VLIW, Very Long Instruction Words e addirittura il dimensionamento dinamico della dataword per segmenti di memoria: configurando a runtime alcuni registri si ha che alcune pagine della medesima RAM vengono accedute a 32 bit, altre a 80 bit, altre come matrici di bit (bit array), eccetera.

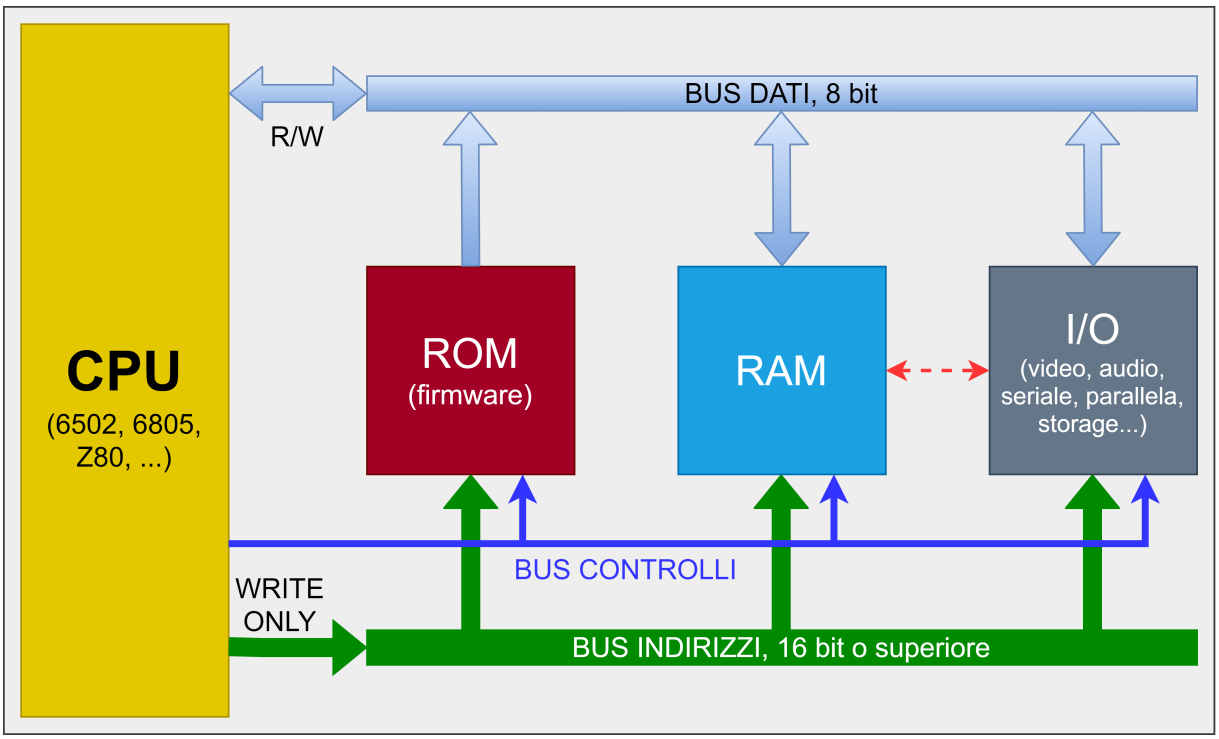


Figura 2.2.1: Tipica architettura Von Neumann per un home computer a 8 bit.

nel 1945 il suo famosissimo rapporto su EDVAC nel quale mette a fuoco alcune idee già anticipate in un articolo del decennio precedente e prefigura compiutamente l'architettura dei futuri calcolatori, che oggi porta il suo nome. Nel 1947, intanto, Aiken e il suo gruppo completano il nuovo Mark II ad Harvard, e potremmo continuare a snocciolare acronimi che caratterizzano anche tutto il decennio successivo, con consumi elettrici e ingombri che nulla avevano da invidiare ad un moderno centro di calcolo o server farm, ma con prestazioni mediamente inferiori a quelle della calcolatrice cinese oggi usata come cadeau se superate i 30€ di spesa alimentare in qualche catena GDO più recente e affidabilità reciproca al tanto sbandierato «five nines».

Dopo questa rapidissima carrellata di primitive macchine più o meno funzionanti (nelle quali i «bug» erano letteralmente veri e propri insetti che proliferavano al calduccio tra i contatti dei relais e/o i filamenti delle valvole termoioniche), appartenenti quasi più al mito che alla storia, sulle quali comuque avviene la genesi dei primi linguaggi HLL come il COBOL di Grace Hopper o ALGOL, la vera maggiore età del calcolatore elettronico arriva solo negli anni Sessanta, quando i pochi ma già solidissimi «big» del settore prendono saldamente in mano il mercato e iniziano la realizzazione delle prime vere serie di macchine commerciali a transistor e poi a IC discreti. L'invenzione del primo microprocessore integrato a 4 bit Intel 4004, ad opera di Federico Faggin, Marcian E. (Ted) Hoff e Stan Mazor chiude poi il cerchio nel 1971: appena tre anni dopo lo stesso Faggin fonderà la Zilog (dove tra l'altro progetterà quasi interamente da solo lo Z80), e in quegli stessi anni vedono la luce in rapidissima successione i progetti del Motorola 6800 e del capostipite MOS 6500, pin compatibile col Motorola e diretto antenato del 6502/6510 sul quale ci concentriamo in queste pagine. Questo sancisce il passaggio definitivo all'era dei computer «a basso costo» (rispetto a mostri sacri come i Vax 11/7xx, già a 32 bit, ed ai vari MIDI e MINI IBM, Bull, Honeywell e altri) e dopo pionieristiche realizzazioni, non meno mitiche (come ALTAIR), spalancherà le porte alla rivoluzione informatica di PET, home computer, personal, workstation.

Come appena accennato, entro la fine degli anni Quaranta del secolo scorso erano comunque già emerse nelle prime realizzazioni pratiche le due principali architetture ancor oggi in uso, ciascuna con pregi e difetti ingegneristici. L'architettura denominata Von Neumann, destinata a larghissima diffusione nell'ambito dei PC mainstream, prevede in sostanza l'uso di un'unica memoria RAM condivisa per dati e programmi indistintamente. L'architettura Harvard, oggi usata nella maggioranza dei sistemi embedded, prevede invece memorie *fisicamente separate* per dati e programmi. Basta un minimo di riflessione per comprendere i numerosi vantaggi di quest'ultima architettura:

1. Possibili ampiezze di parola differenti per dati e programmi, con diretta influenza sull'ampiezza dei registri;
2. Prelievo *contemporaneo* di dati e istruzioni dalla memoria, con un effettivo **raddoppio prestazionale** a parità di ogni altro fattore;

3. Possibilità di scegliere tipologie di memoria per dati e programmi con caratteristiche molto diverse tra loro (esempio classico, una memoria di programma di sola lettura, non riscrivibile) secondo le specifiche economiche, prestazionali (tempi di accesso differenziati), di sicurezza e affidabilità, di protezione della proprietà intellettuale, etc.

Per contro, è palese che l'architettura Harvard richiede una più precisa stima a priori della dimensione massima del codice e dei dati per il dimensionamento delle relative memorie, il che la rende principalmente adatta ai sistemi di elaborazione dedicati, mentre la flessibilità applicativa offerta da un sistema Von Neumann risulta ottimale per un sistema di calcolo general purpose. Tale motivazione, assieme a intuitive ragioni di semplificazione progettuale ed economicità della componentistica (in un'epoca in cui il costo esorbitante delle memorie era ancora uno dei principali parametri limitanti nel cost engineering di un sistema di calcolo), ha portato la maggioranza dei progettisti di home e personal computer a scegliere quest'ultima architettura.

Come chiaramente visibile in figura (2.2.1), nell'architettura Von Neumann dei comuni home e personal computer il microprocessore o CPU gestisce tre soli bus, costituiti da aggregati di segnali logici con molteplicità ben definita: bus dati (tipicamente 8 bit per le CPU di nostro interesse), bus indirizzi (ad esempio, 16 bit consentono di indirizzare 64kib, ovvero l'intera memoria del Commodore 64) e il bus dei controlli costituito da segnali logici eterogenei di interrupt, reset, direzione (es. lettura/scrittura) ma anche sincronizzazione, selezione, abilitazione, arbitraggio, consenso e altro necessario alla gestione e convalida dei flussi di dati. A ciascun indirizzo corrisponde una data locazione di memoria fisica: semplificando in modo estremo, quando l'indirizzo compare sul bus, il contenuto della corrispondente locazione può essere letto o sovrascritto dalla CPU.

Dati e programmi utente condividono *indistintamente* il medesimo spazio di indirizzamento e la medesima memoria RAM[8] (Random Access Memory), mentre il firmware di sistema (ad esempio, il Kernal e l'interprete BASIC V2 di un Commodore 64) risiedono su memorie di sola lettura (ROM, Read-Only Memory), comunque connesse ai medesimi bus di indirizzi e dati delle RAM e quindi appartenenti in aggregato ad *un unico spazio fisico* e logico di indirizzamento. Ovvero: ambedue le tipologie di memoria sono mappate entro un unico spazio di indirizzamento, in locazioni diverse. Ciò si applica, nel nostro esempio, anche ai chip specializzati di I/O, che presiedono all'interfacciamento del calcolatore con il mondo esterno.

Nella maggioranza delle CPU che prenderemo in considerazione, infatti, viene usato l'approccio denominato **MMIO**, ossia Memory Mapped Input/Output: i registri dei vari chip di supporto alla CPU sono mappati in normali locazioni di memoria, che quindi non risultano disponibili per i programmi utente. In alternativa, altre CPU (inclusi gli Intel 8088/86) hanno un vero e proprio spazio di indirizzamento separato (gestito da una specifica linea nel bus dei controlli) e istruzioni distinte (es. `IN`, `OUT`) per le funzioni di I/O.

Proprio nelle architetture MMIO è invalsa la consuetudine (poi universalizzatasi per i suoi numerosi vantaggi) di fare uso dei cosiddetti *registri serializzatori*: in questo modo la periferica occuperà solo una o due locazioni nello spazio di indirizzamento, ma tramite sequenze di scritture e letture da e verso il registro serializzatore sarà possibile accedere ad un numero di registri interni potenzialmente molto elevato, con evidente vantaggio in termini di footprint, controbilanciato ovviamente da un relativo rallentamento delle operazioni sul bus a causa delle sequenze di scritture e letture richieste da tale protocollo di accesso.

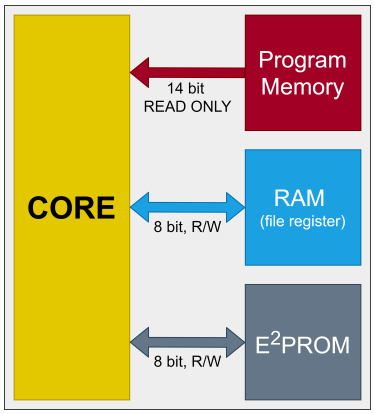


Figura 2.2.2: Esempio fortemente semplificato di architettura Harvard di un moderno microcontroller. Sono visibili tre distinte memorie on-chip, ciascuna col proprio bus separato, con differenti ampiezze (rimarchevole vantaggio di tale architettura), modalità e velocità di accesso.

[8] Si invita il lettore a riflettere su come tale caratteristica consenta di *eseguire codice arbitrario* dopo averlo modificato in RAM: in sostanza consente anomalie logiche come l'esecuzione di dati (voluta o meno) e tecniche di self-morphing del codice, che di norma sono fisicamente impossibili nelle architetture Harvard, con tutto ciò che ne consegue in termini di affidabilità e sicurezza.

Capitolo 3

Elementi di aritmetica per il calcolo digitale.

Riportiamo qui in forma estremamente sintetica e per pura comodità del lettore alcune elementari nozioni su nomenclatura e basi numeriche: la trattazione sarà volutamente informale e limitatissima, per due principali ordini di ragioni. In primis, questo non è un corso di reti logiche, logiche programmabili, aritmetica digitale, logica formale o calcolo numerico: i nostri scopi sono altri. Secondo, ma non meno importante: tali argomenti sono reiteratamente squadernati in modo più o meno esaustivo non solo in tutti i testi indicati in bibliografia, ma anche in una pletora di altri testi. Ad esempio: architettura dei calcolatori, introduzione alla programmazione, problem solving, logica applicativa, elettronica digitale, algoritmica, matematica discreta, monografie su aritmetica digitale e funzioni booleane... e anche noi, come il buon Hollerith, non abbiamo alcuna intenzione di reinventare la ruota, né di indulgere troppo su concetti e formule oggettivamente stranoti anche per i programmatori di alto livello.

3.1 Nomenclatura di base.

Abbiamo accennato al fatto che un calcolatore digitale sostanzialmente elabora segnali elettrici codificati in modo da rappresentare individualmente solo i valori 0 e 1, in elettronica digitale denotati Low e High (L e H, basso e alto) rispettivamente, corrispondenti alle costanti logiche *Falso e Vero, False e True, F e T*. Tali valori, considerati *per gruppi contigui* in un ordine rigidamente prefissato, consentono in realtà di rappresentare un dato intervallo di valori interi, limitato unicamente da alcuni parametri architetturali (come abbiamo appena visto, l'ampiezza del bus dati e dei registri). L'ampiezza del bus dati corrisponde alla *parola di memoria* (**memory word**), il che è doppiamente importante nelle architetture Harvard (nelle quali la **program memory word** ha praticamente sempre ampiezza diversa dalla data memory word), anche se l'espressione rischia una parziale sovrapposizione con altri termini invalsi nell'uso, che qui riassumiamo in modo estremamente sintetico.

Bit: come già anticipato, è la crasi di **bi**nary digi**t** ossia cifra binaria. Può assumere valori solamente nell'insieme $\mathbb{B} = \{0, 1\}$.

LSB: acronimo di Least Significant Bit. Indica sempre il bit meno significativo, situato più a *destra* nella notazione posizionale. Si presti attenzione al fatto che talora, in letteratura, si fa riferimento con tale acronimo anche all'intero byte meno significativo, in caso di valori multibyte.

MSB: Most Significant Bit, referenzia il bit posto all'estrema *sinistra* del gruppo considerato. In taluni contesti si pone arbitrariamente attenzione alla differenza tra maiuscolo e minuscolo, riservando quest'ultimo per il lsb, come talora avviene in letteratura (principalmente anglosassone, o ad essa ispirata) per sottolineare ulteriormente la differenza tra il *massimo* comun divisore MCD (GCD nella letteratura internazionale) e il *minimo* comune multiplo, mcm (lcm). Tuttavia, nella maggioranza dei contesti tali convenzioni vengono del tutto ignorate.

Nibble: un gruppo di quattro bit. In genere, per i bit più significativi si parla di *nibble alto* (high nibble) e *nibble basso* (low nibble) per i meno significativi, relativamente all'ampiezza di parola di 8 bit (un tempo universale e pervasiva: il mondo dei grandi sistemi a 32 bit, come i DEC VAX 11/750 e superiori, era molto lontano e inaccessibile per il tipico utente home e personal).

Byte: insieme di 8 bit.

Word: un doppio byte, ossia 16 bit.

Doubleword: o dword, indica un'ampiezza di 32 bit, pari a 2 word, 4 byte, 8 nibbles.

Quadword: o qword, gruppo contiguo di 64 bit, pari rispettivamente a 2 doubleword, 4 word, 8 byte, 16 nibbles.

Tenword: o tword, è un gruppo di 80 bit.

3.1.1 Endianness.

Nel caso (assai comune) in cui il valore da memorizzare occupi più byte e in generale superi l'ampiezza di parola del bus dati, sorge il problema dell'*ordine di memorizzazione* o *endianness*. Nel «caos primordiale» che ha circondato la nascita dei primi microprocessori, la totale assenza di regolamentazioni tecniche internazionali di riferimento e spesso anche la precisa volontà di differenziarsi dalla concorrenza con scelte radicalmente incompatibili tra loro hanno condotto al fiorire di approcci proprietari, di cui questo è un noto esempio. Molti produttori hanno infatti scelto l'ordine *little endian* (LE), una sequenza nella quale il byte *meno significativo* occupa la posizione *più bassa* in memoria («little end first», ossia il byte meno significativo si incontra per primo scorrendo la memoria per indirizzi crescenti): altri, come Motorola, hanno invece abbracciato la convenzione diametralmente opposta[1], *big endian* (BE), nella quale per contro ad indirizzi di memoria *crescenti* corrispondono byte sempre meno significativi. Vale la pena di rimarcare che la memorizzazione *little endian* ha il vantaggio di consentire l'avvio di una operazione aritmetica all'interno della CPU non appena viene letta la prima locazione (corrispondente al byte meno significativo), poiché per ragioni di efficienza e precisione l'aritmetica digitale è implementata in modo da procedere esattamente come nel calcolo manuale «per colonne», a partire dai valori meno significativi.

3.2 Basi numeriche non decimali.

Alle scuole elementari abbiamo appreso l'intuitiva (ma non scontata) **notazione posizionale** per i numeri decimali, con la nomenclatura che riguarda unità, decine, centinaia, migliaia... secondo la posizione della cifra. Ricordando che tutti i valori qui referenziati sono esclusivamente *numeri interi non negativi*, quindi naturali $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$, consideriamo il numero n composto da $k+1$ cifre e siano $d_0, d_1, \ldots, d_k$ tali cifre dove il pedice minore corrisponde alla cifra meno significativa, ossia *più a destra* nel sistema posizionale:

$$n = d_k \cdots d_1 d_0 = d_k \cdot 10^k + \cdots + d_1 \cdot 10^1 + d_0 \cdot 10^0 = \sum_{i=0}^{k} 10^i d_i \qquad \text{dove } 0 \leq d_i < 10 \tag{3.2.1}$$

Ebbene, tale espansione polinomiale è estensibile senza perdita di significato sostituendo al familiare 10 una *qualsiasi base numerica naturale* arbitraria[2] $b \geq 2$: di particolare rilevanza nel contesto computazionale sono alcune basi che risultano essere potenze intere del due, in particolare $b = 2$ (sistema binario), $b = 2^3 = 8$ (ottale) e $b = 2^4 = 16$ (esadecimale). La motivazione per la scelta di queste ultime due basi è la loro relativa espressività rispetto al binario (numero di cifre necessarie in relazione al valore rappresentato), e la facilità di eseguire mentalmente conversioni intermedie. La forma generale dell'equazione non cambia di molto:

$$n = \sum_{i=0}^{k} b^i d_i \qquad \text{dove } 0 \leq d_i < b,\ b \geq 2 \tag{3.2.2}$$

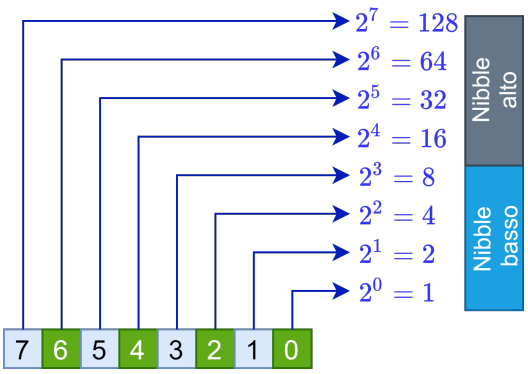


Figura 3.2.1: Posizioni dei bit e potenze del due corrispondenti.

Nel seguito useremo un pedice per indicare la base numerica, come consuetudine in questi casi, omettendolo eventualmente solo per i numeri decimali e laddove sia chiaro dal contesto di che base si tratta. L'illustrazione (3.2.1) mostra i valori delle potenze del due corrispondenti alle 8 posizioni di un byte.

[1] Volendo sono possibili ulteriori, esotiche soluzioni *middle-endian* per l'ordine di memorizzazione, adottate tuttavia da una sparutissima minoranza di progettisti per architetture decisamente molto rare.

[2] Per gli scopi del presente discorso, sarà ampiamente sufficiente considerare le più banali basi, per cui vale $b \in \mathbb{N},\ b \geq 2$. Per stimolare la curiosità nel lettore, si segnala comunque la notevole utilità computazionale di basi più esotiche, come $b = -2$ oppure $b = -1 + i$ dove i è l'unità immaginaria $i = \sqrt{-1}$ (si veda ad esempio [Knu73, War13]), e si vuole menzionare anche l'esistenza di notazioni multibase, la più nota delle quali è probabilmente costituita dai *factoradics* ([Knu97], pag. 12).

Passando ai logaritmi e confrontando gli esponenti, è immediato valutare l'espressività relativa delle varie basi, in questo caso il numero di bit necessario per rappresentare una singola cifra in base 8 e 16. Avendo $8 = 2^3$ e $16 = 2^4$, una cifra esadecimale corrisponderà a 4 cifre binarie, mentre con l'ottale si ha una ovvia corrispondenza $1:3$. Quindi, ad esempio, dato il numero binario $10100101_2 = 245_8 = A5_{16} = 165_{10}$, si avrà:

Binario	010	100	101		Binario	1010	0101
Ottale	2	4	5		Esadecimale	A	5

(3.2.3)

Come si vede, la conversione tra i tre sistemi numerici è estremamente semplice da eseguire mnemonicamente, suddividendo appropriatamente in gruppi le cifre e utilizzando l'immediata corrispondenza tabulare sotto riportata. L'uso dell'esadecimale consente una notazione compatta ed a prova di errori rispetto al binario, ed è supportato anche da una moltitudine di linguaggi di alto livello, oltre ed essere pressoché ubiquo in Assembly. La tabella seguente mostra un raffronto delle quattro rappresentazioni per i primi 16 valori decimali. Si noti come in esadecimale le cifre superiori al 9 vengono indicate con le prime lettere (maiuscole) dell'alfabeto latino, per un totale di 16 simboli distinti come richiesto dalla definizione. In tal modo, usando le 26 lettere del latino esteso (che includono J, K, W, X, Y) è possibile agevolmente rappresentare numeri fino alla base 36, effettivamente utilizzata in alcuni ambiti.

Decimale	Binario	Ottale	Esadecimale
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

La conversione da binario a decimale è parimenti immediata e si riduce, di fatto, alla *sommatoria dei pesi* relativi ai *soli bit pari a* 1. La illustreremo in modo informale, con un semplice esempio che sarà ampiamente sufficiente alla comprensione intuitiva. Sia dato il byte 11000111_2:

$$\left|\begin{array}{c|c|c|c|c|c|c|c} 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \end{array}\right| = \\ 2^7 + 2^6 + 2^2 + 2^1 + 2^0 = \\ 128 + 64 + 4 + 2 + 1 = 199_{10} \tag{3.2.4}$$

Risulta immediato desumere la regola generale semplificata: essendovi, come in ogni altra base numerica, una **moltiplicazione implicita** tra ciascuna cifra e il corrispondente peso o potenza della base, qui concorrono alla somma le sole potenze del due i cui esponenti corrispondono alle posizioni dei bit pari ad 1. La conversione di base opposta, da decimale a binario, è invece più tediosa essendo basata su una serie di divisioni intere per due, nelle quali il resto fornisce in sequenza i bit della parola binaria risultante. Si fornisce un semplice esempio anche in questo caso, dal quale il lettore potrà facilmente trarre la regola generale del semplice algoritmo con divisione *intera* e resto. Banalmente, il *quoziente intero* della divisione per due di ciascuna riga diviene il *dividendo* della riga successiva, fino all'ottenimento di un quoziente nullo. Si presti solo attenzione al peculiare ordine dei resti: il primo resto che si ottiene corrisponde al bit meno significativo, poi si procede per pesi crescenti fino al MSB. D'altro canto, il primo resto è esattamente quello che determina se il valore considerato è pari o dispari, e tale semantica viene ritenuta dal bit meno significativo del valore binario: se è nullo, il valore è *pari*, e viceversa è dispari se il LSB vale 1. Il valore binario ottenuto viene quindi letto e trascritto scorrendo la serie dei resti *dal basso verso l'alto*. Sia dato il valore decimale 184_{10}:

$$\begin{aligned} 184 \div 2 &= 92 && \text{resto } = 0 \text{ LSB} \\ 92 \div 2 &= 46 && \text{resto } = 0 \\ 46 \div 2 &= 23 && \text{resto } = 0 \\ 23 \div 2 &= 11 && \text{resto } = 1 \\ 11 \div 2 &= 5 && \text{resto } = 1 \\ 5 \div 2 &= 2 && \text{resto } = 1 \\ 2 \div 2 &= 1 && \text{resto } = 0 \\ 1 \div 2 &= 0 && \text{resto } = 1 \text{ MSB} \end{aligned} \tag{3.2.5}$$

$$184_{10} = 10111000_2$$

Disposizioni con ripetizioni. Dati n simboli distinti e un intero $0 < k \leq n$, si dicono *disposizioni con ripetizioni* di questi n elementi in classe k tutte le possibili presentazioni tali che:

- Ciascuna presentazione contenga esattamente k simboli, non necessariamente distinti;
- Ciascun simbolo possa essere ripetuto fino a k volte;
- Due presentazioni differiscano per qualche simbolo, oppure per l'ordine in cui i simboli sono disposti.

Il totale delle possibili disposizioni con ripetizioni di n elementi in classe k è pari a:

$$DR(n,k) := n^k \tag{3.2.6}$$

I $DR(2,4) = 16$ numeri binari a 4 bit $0000, 0001, \ldots, 1111$ o i $2^8 = 256$ possibili valori per un byte (8 bit) sono uno dei più classici esempi di disposizione con ripetizione, mentre per un diverso esempio le disposizioni con ripetizione $DR(3,2) = 9$ degli elementi dell'insieme $A = \{a, b, c\}$ sono tutte le coppie del prodotto cartesiano $A \times A = \{(a,a), (a,b), (a,c), (b,a), (b,b), (b,c), (c,a), (c,b), (c,c)\}$.

I valori naturali (gli usuali interi non negativi, che in questo contesto vengono definiti *numeri non segnati*) esprimibili in binario con un dato numero di bit sono limitati inferiormente dallo zero e superiormente dalla posizione del MSB: avendo un byte (8 bit), il massimo valore esprimibile sarà $2^8 - 1$, ovvero $255_{10} = 11111111_2$, per un totale di 256 valori possibili[3]. Tale totale infatti equivale combinatorialmente al numero di *disposizioni con ripetizioni* dei due soli simboli disponibili: 0 e 1. La tabella seguente illustra i limiti massimi per le varie ampiezze di parola già definite.

Nome	Esponente	Limite superiore (non segnato)
Nibble	4	$2^4 - 1 = 15$
Byte	8	$2^8 - 1 = 255$
Word	16	$2^{16} - 1 = 65.535$
Dword	32	$2^{32} - 1 = 4.294.967.295$
Qword	64	$2^{64} - 1 = 18.446.744.073.709.551.615$

A proposito di multipli, le unità di misura informatiche non seguono il convenzionale andamento del Sistema Internazionale delle unità di misura per potenze ternarie del dieci, ma fanno uso di *specifiche scale* basate sulle potenze del due. Si veda la seguente tabella comparativa:

Binario			Decimale		
Nome	Simbolo	Valore	Nome	Simbolo	Valore
kilobibyte	KiB	2^{10}	kilobyte	KB	10^3
megabibyte	MiB	2^{20}	megabyte	MB	10^6
gigabibyte	GiB	2^{30}	gigabyte	GB	10^9
terabibyte	TiB	2^{40}	terabyte	TB	10^{12}
petabibyte	PiB	2^{50}	petabyte	PB	10^{15}
exabibyte	EiB	2^{60}	exabyte	EB	10^{18}

[3] In matematica discreta e in informatica i conteggi partono *per default* da zero e, più in generale, l'insieme $\mathbb{N}$ dei naturali include sempre lo zero (interi non negativi). Questo ha una solidissima ragione: interpretando tali numeri come **ordinali**, ciascuno di essi risponde alla domanda «quanti numeri precedono il valore corrente?». Quindi il numero 1 ha esattamente un predecessore, lo zero, e così via induttivamente per ciascun naturale. D'altro canto, anche nella costruzione classica in cui si interpretano i naturali come cardinalità di insiemi risulta parimenti necessario partire dall'insieme vuoto.

Chiudiamo con un semplice esempio di memorizzazione little e big endian: data la dword (32 bit, 4 bytes) esadecimale $A07598BD_{16}$, che supponiamo memorizzata a partire dall'indirizzo esadecimale 1000_{16}, avremo la seguente situazione in memoria su una macchina *little endian* (mnemonicamente: «il valore meno significativo all'indirizzo di memoria minore»):

Indirizzo	Valore	
1000	BD	LSB
1001	98	
1002	75	
1003	A0	MSB

Simmetricamente, su una architettura *big endian* (mnemonicamente: «il valore più significativo all'indirizzo di memoria minore») si avrà:

Indirizzo	Valore	
1000	A0	MSB
1001	75	
1002	98	
1003	BD	LSB

3.3 Cenni di aritmetica in base binaria.

Esaminiamo molto rapidamente le principali operazioni aritmetiche così come eseguite nello stadio ALU (l'Unità Aritmetico Logica) di una CPU a 8 bit. Le famigliari operazioni dell'aritmetica di Peano apprese alle scuole elementari rimangono ovviamente valide anche cambiando rappresentazione, ma in linea di principio si semplificano notevolmente.

La *somma* di due bit, che chiameremo rispettivamente A e B, segue le regole elencate nella tabella seguente, nota in logica come *truth table* o tabella di verità.

#	A	B	$A+B$	Riporto
0	0	0	0	0
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1

Vale la pena di sottolineare contestualmente la convenzione universalmente invalsa nelle tabelle di verità di **ordinare** i valori per le variabili binarie (il vettore binario) in ingresso considerandoli come *un singolo valore* in base due (in questo caso una parola a due bit b_1b_0 tale che $b_1 = A, b_0 = B$), in modo da ottenere l'equivalente decimale riportato nella colonna più a sinistra per codificare comodamente i possibili casi in input.

Si noti come, in piena analogia col familiare sistema decimale, anche qui si usa il *riporto* (carry) per indicare un valore non esprimibile con una singola cifra. Tale riporto viene impiegato, in cascata, per l'usuale addizione di numeri binari multicifra in colonne, di cui segue un semplice esempio.

Si abbiano i due addendi binari a 8 bit $a_1 = 10100011_2$ e $a_2 = 00110000_2$. In rosso sono indicati i valori dei riporti: inizialmente si considera nullo tale valore.

$$\begin{array}{cccccccccc} 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & (c) \\ & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & + \\ & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & = \\ \hline 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & \end{array} \tag{3.3.1}$$

Se l'addizione tra naturali è decisamente semplificata in binario, la somma algebrica richiede invece qualche ulteriore passaggio. Occorre infatti una convenzione per rappresentare numeri segnati: il cosiddetto *complemento a due*. Tale complemento si realizza in due semplici passaggi concettuali. Consideriamo il byte $b = 00111001_2$:

- **Complemento a uno**: si ottiene semplicemente *negando* ovvero invertendo ogni singolo bit, in modo tale da sostituire ciascun valore 0 con un 1, e viceversa. Nel nostro esempio, si avrà: 11000110_2.

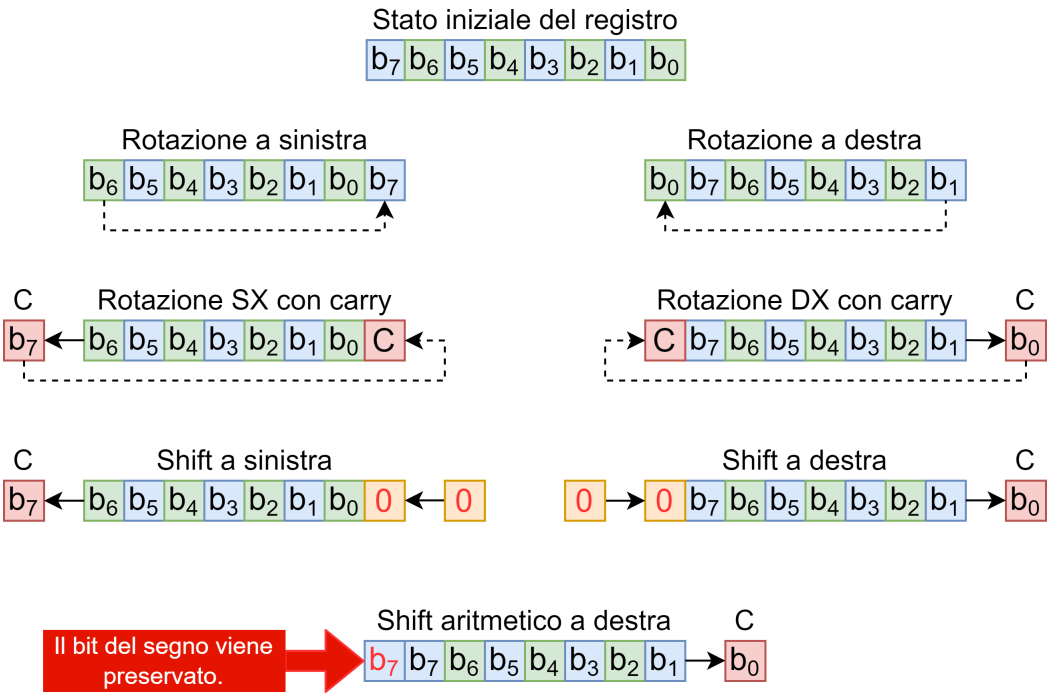


Figura 3.3.1: Sinottico delle varie versioni di shift e rotazione.

- **Incremento di una unità**: il complemento a due si ottiene ora aggiungendo 1 al valore ottenuto al passaggio precedente. Nel nostro esempio: $11000110_2 + 1 = 11000111_2$.

Il valore così ottenuto rappresenta il *complemento a due* del valore di partenza, ovvero la sua rappresentazione come *numero negativo*. Si noti che il MSB è pari ad uno: in questa rappresentazione, esso costituisce infatti il *bit del segno*, in particolare quando esso risulta **nullo** il segno è **positivo**, e viceversa.

L'operazione di complemento a due equivale in realtà a *sottrarre il valore di partenza* da 256_{10} nel caso esemplificato di un byte, e più in generale da 2^n dove n è l'ampiezza del registro considerata. Si suggerisce brevemente, senza voler complicare la trattazione, che il fatto di lavorare con un numero necessariamente *finito* di possibili valori rappresentabili per una data ampiezza (byte, word, etc.) significa, da un punto di vista algebrico, che stiamo in realtà operando su una **classe di resto** in modulo 2^n.

Supponiamo quindi di voler sottrarre il numero $b = 00101100_2 = 44_{10}$ da $a = 01100011_2 = 99_{10}$. Il complemento a due di b è pari a $11010100_2 = 212_{10} = 256_{10} - 44_{10}$, pertanto avremo:

$$\begin{array}{l} 01100011+ \\ \underline{11010100 =} \\ 00110111 \end{array} \tag{3.3.2}$$

Si noti che il riporto finale, se presente (come in questo caso) deve essere ignorato. Risulta immediato verificare il risultato: $00110111_2 = 55_{10} = 99_{10} - 44_{10}$, esattamente quanto intendevamo ottenere.

Per somme e moltiplicazioni sono altresì possibili approcci analoghi, che tuttavia qui non tratteremo, anche perché la maggioranza delle CPU di nostro interesse non dispone di uno stadio moltiplicatore interno. Occorrerà pertanto, al momento opportuno, considerare l'implementazione in Assembly di un idoneo ed efficiente algoritmo di moltiplicazione binaria.

3.3.1 Shift e rotazioni.

Una ulteriore classe di operazioni, in realtà borderline tra aritmetica e logica sequenziale, sono gli *shift* (scorrimenti) e le *rotazioni* che normalmente operano su un registro (byte). In breve, uno shift (logico) a destra di una posizione corrisponde ad una *divisione per due* del valore memorizzato, mentre a sinistra ha l'effetto di una *moltiplicazione per due*, come si può banalmente verificare. La rotazione è invece un'operazione *ciclica*: dato un registro di ampiezza n, la configurazione iniziale dei bit si ripresenta ciclicamente dopo n rotazioni (che diventano $n+1$ nel caso di rotazione attraverso il carry, una variante molto comune). Le diverse CPU prevedono nel set di istruzioni alcune varianti di tali operazioni, che saranno affrontate in dettaglio al momento opportuno. La figura (3.3.1) presenta le più diffuse versioni di shift e rotazione che è possibile trovare implementate come istruzioni atomiche.

3.4 Cenni sulle operazioni logiche combinatorie.

Visti gli scopi e la natura della trattazione, in questa sede rinunciamo a priori all'enunciazione formale dei cinque assiomi della logica booleana e delle sue numerose proprietà per concentrarci unicamente su esempi e applicazioni.

Una *funzione booleana combinatoria*, in senso generale, è una funzione $f : \{0,1\}^n \to \{0,1\}$ che prende in input un vettore finito di n variabili logiche e fornisce in output una singola variabile booleana. Le funzioni booleane, come qualsiasi altra funzione algebrica, si possono categorizzare in base al numero di variabili su cui operano: unarie, binarie, ternarie, quaternarie, vettoriali in genere, etc. ricordando comunque che gli operatori elementari previsti dall'algebra di Boole sono solamente *unari* e *binari*. Il numero totale n di possibili funzioni booleane $f_j(b_1, b_2, \ldots, b_k)$ operanti su k valori booleani in input (si veda anche l'equazione 3.2.6 a pagina 22) è dato da:

$$n = DR(2, DR(2, k)) \tag{3.4.1}$$

dove il primo parametro rappresenta la cardinalità dell'insieme di base dei simboli disponibili: in questo caso $\mathbb{B} = \{0,1\}$ e quindi $|\mathbb{B}| = 2$. Il valore del secondo parametro, in questo caso una seconda funzione $DR()$, enumera invece tutte le possibili configurazioni in input di k bit, essendo come già visto $DR(2,k) = 2^k$ e quindi, in ultima analisi:

$$n = 2^{2^k} \tag{3.4.2}$$

come riportato generalmente in letteratura. Data però la scarsa leggibilità di tale espressione, si preferisce nel seguito la più intuitiva ed immediata forma funzionale combinatoria $DR(2, 2^k)$.

3.4.1 Funzioni booleane unarie.

Consideriamo ora tutte le possibili funzioni *unarie*, ossia le $f_j : \{0,1\} \to \{0,1\}$ con un singolo valore booleano A in ingresso: in totale sono ovviamente $DR(2, 2^1) = 4$.

A	0	1	2	3
0	0	0	1	1
1	0	1	0	1

Il criterio di enumerazione arbitrario (ma pressoché universale in letteratura) qui utilizzato è molto semplice: ordinando i valori della variabile di ingresso in modo *crescente*, si enumerano sequenzialmente tutte le possibili *disposizioni con ripetizioni* di due simboli su due posizioni $DR(2,2)$ dell'uscita corrispondente. A ciascuna disposizione corrisponde una possibile funzione, che risulta codificata leggendo *come un singolo numero binario* la sequenza delle uscite, *dall'alto verso il basso*. Così, ad esempio, la funzione #1 o $f_1(A)$ è caratterizzata dall'avere in uscita la sequenza 01_2 in corrispondenza dell'ordine scelto per i valori assunti dalla variabile di ingresso.

Analizzando la tabella, si nota che le funzioni agli estremi $f_0(A) = 0$ e $f_3(A) = 1$ non risultano particolarmente utili ai fini applicativi, in quanto costanti e quindi **scorrelate** dal valore in ingresso. Le altre due funzioni sono $f_1(A) = A$ (detta anche YES[4]) e $f_2(A) = \mathrm{NOT}(A)$ (indicata con molte varianti, tra cui $\neg A$ e, in elettronica digitale, $\overline{A}$), quest'ultima di estrema importanza. Vale la pena di sottolineare subito che la logica booleana è *involutiva*, poiché applicando ricorsivamente la negazione[5] si torna ciclicamente al valore non negato di partenza: $\neg(\neg A) = A$.

Naturalmente è possibile trovare in letteratura la medesima tabella riscritta in vari modi, ad esempio come segue per enumerare ancora più esplicitamente le possibili funzioni in base all'output:

A	0	1
$f_0(A)$	0	0
$f_1(A)$	0	1
$f_2(A)$	1	0
$f_3(A)$	1	1

[4] La porta YES (buffer) è effettivamente implementata nelle logiche digitali integrate, ma la sua funzionalità logica è del tutto trasparente. Gli scopi di tale gate sono circuitalmente legati all'amplificazione di corrente e moltiplicazione del *fan-out* (buffering), alla possibilità di implementare un'uscita «open collector» e all'adattamento di livelli di tensione tra differenti sezioni circuitali.

[5] Si vuole qui solo accennare al fatto che attualmente sono in uso in logica matematica e simbolica circa **settecento** diversi sistemi formali, molti dei quali direttamente derivati dalla logica booleana tramite aggiunte di assiomi e/o per elisione di certe sue proprietà anticamente ritenute imprescindibili e «naturali»: ad esempio l'involutività della negazione ricorsiva (appena vista) o il principio del «terzo escluso» nel caso delle logiche polivalenti, si veda a tale proposito anche la nota 5 a pagina 15.

3.4.2 Funzioni booleane binarie.

Ripetiamo ora il lavoro di enumerazione esaustiva per tutte le possibili funzioni di due variabili (bit) A e B, ossia le *funzioni binarie* $f_j : \{0,1\}^2 \to \{0,1\}$, che ammontano a $DR(2,2^2) = 16$:

#	A	B	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1	0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
2	1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
3	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Come già visto in precedenza, i possibili valori di ingresso sono codificati considerandoli un unico valore binario per righe (il cui valore decimale è riportato in colore rosso nella colonna #) mentre i valori posizionali nella riga di intestazione sono ancora l'equivalente decimale del sottostante valore binario, letto dall'alto in basso, dal MSB (riga 0) al LSB (riga 3), usato anche come pedice per individuare la funzione $f_j(A,B)$. Riscriviamo la tabella in modo da evidenziarne le grandi caratteristiche di simmetria e complementarietà:

Pos.	Output	$F_j(A,B)$	$F_j(A,B)$	Output	Pos.
0	0000	$f_0(A,B) = FALSE$	$f_{15}(A,B) = TRUE$	1111	15
1	0001	$f_1(A,B) = A$ AND B	$f_{14}(A,B) = \neg(A$ AND $B)$	1110	14
2	0010	$f_2(A,B) = \neg(A \Rightarrow B)$	$f_{13}(A,B) = A \Rightarrow B$	1101	13
3	0011	$f_3(A,B) = A$	$f_{12}(A,B) = \neg A$	1100	12
4	0100	$f_4(A,B) = \neg(B \Rightarrow A)$	$f_{11}(A,B) = B \Rightarrow A$	1011	11
5	0101	$f_5(A,B) = B$	$f_{10}(A,B) = \neg B$	1010	10
6	0110	$f_6(A,B) = A$ XOR B	$f_9(A,B) = \neg(A$ XOR $B)$	1001	9
7	0111	$f_7(A,B) = A$ OR B	$f_8(A,B) = \neg(A$ OR $B)$	1000	8

Si potrebbe dedicare un intero testo alla lettura approfondita di tale tabella, ma ci limitiamo a far notare come a valori di codifica complementari (es. 2 e 13), perché a somma costante per righe, corrispondono funzioni logiche che sono l'una il *complemento* (ossia il negato) dell'altra: $f_j(A,B) = \neg f_{15-j}(A,B)$ dove $0 \leq j \leq 7$.

Analizzando velocemente la tabella sinottica delle funzioni binarie, quanto già considerato sopra per le funzioni unarie estreme si applica anche qui alle posizioni 0 e 15. Le coppie di funzioni $f_3(A,B), f_{12}(A,B)$ e $f_5(A,B), f_{10}(A,B)$ sono di fatto **unarie**, in quanto l'output è funzione di *una sola* delle variabili di ingresso, mentre per l'altra vale quella che chiamiamo *condizione di indifferenza*. Si nota come l'operatore di negazione NOT, essendo unario, si applica ad una singola variabile o all'output di un'altra funzione.

3.4.3 Funzioni booleane come istruzioni atomiche e principio di segregazione bit a bit.

Tra le restanti funzioni, assumono notevole importanza $f_1(A,B), f_7(A,B), f_6(A,B)$ ovvero rispettivamente AND, OR e XOR[6] che corrispondono universalmente ad altrettante istruzioni delle varie CPU, dette *atomiche* proprio perché realizzabili con una singola istruzione. Per comodità del lettore, si ripropongono le sole colonne relative a tali funzioni, estrapolate dal sinottico.

A	B	A AND B	A OR B	A XOR B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

[6] In letteratura tecnico-scientifica è possibile trovare indicate tali funzioni con i più disparati simbolismi, ad esempio per la congiunzione o *moltiplicazione booleana* AND $\wedge, \&, *, \cdot, \odot, \otimes, ...$, per la disgiunzione o *somma booleana* OR $\vee, |, +, \oplus, ...$, per lo XOR o *funzione di parità* $\oplus, \hat{}, \not\equiv, ...$ (non è un refuso, secondo i vari contesti viene effettivamente usato il medesimo simbolo $\oplus$ per OR o XOR!) e infine per la negazione unaria NOT $\neg, !, \sim, \overline{a}, ...$ nei vari ambiti: logica formale, logiche temporali, informatica applicativa, elettronica circuitale digitale, reti logiche.

Riguardo alla scelta dei simboli, si sono qui volutamente evitate le notazioni formali possibilmente più criptiche per il lettore, mantenendo unicamente i simboli di negazione $\neg$ e di implicazione $\Rightarrow$.

Una funzione logica multibit (tipicamente una istruzione Assembly che agisce su un registro ampio un byte) opera semplicemente applicando a ciascuna coppia di bit di peso corrispondente la tabella di verità dell'operatore prescelto, in totale isolamento rispetto ai bit adiacenti, poiché tali operazioni bitwise sono tipicamente *segregate*, ossia limitate ai due bit su cui agiscono, **senza effetti collaterali** su altri eventuali bit (non esiste il concetto di «riporto» per gli operatori logici binari). Si abbiano ad esempio i due byte $B_0 = 10010100_2$ e $B_1 = 01001101_2$, mostriamo i risultati delle tre operazioni booleane fondamentali incolonnando i bit:

$$\begin{array}{lll} 10010100 \text{ AND} & 10010100 \text{ OR} & 10010100 \text{ XOR} \\ 01001101 = & 01001101 = & 01001101 = \\ \hline 00000100 & 11011101 & 11011001 \end{array} \tag{3.4.3}$$

Nelle successive sezioni del testo, ad esempio per descrivere gli effetti degli opcode e negli esempi di codice, useremo alternativamente il nome esplicito delle funzioni oppure i simboli indicati nella tabella seguente:

AND	OR	XOR
$\wedge$	$\vee$	$\oplus$

3.4.4 Il principio di dualità e i teoremi di de Morgan.

Anche in una trattazione ridotta ai minimi termini come la presente è inevitabile notare la forte ridondanza e interdipendenza delle funzioni nell'algebra booleana. In effetti esistono numerosi modi in cui ciascuna di esse può essere equivalentemente espressa in funzione delle altre[7]. Di questo aspetto danno conto in particolare i teoremi del già menzionato Augustus de Morgan (vedi sez. 2 a pagina 14), che qui riportiamo ovviamente senza alcuna dimostrazione:

$$\begin{aligned} \neg(A \text{ OR } B) &= \neg A \text{ AND } \neg B \\ \neg(A \text{ AND } B) &= \neg A \text{ OR } \neg B \end{aligned} \tag{3.4.4}$$

Vale anche la pena di ricordare, senza complicare ulteriormente la trattazione, che le algebre booleane godono delle usuali proprietà **commutativa** e **associativa**, riconoscendo anche una **precedenza** implicita degli operatori e l'uso di parentesi per alterare tali priorità. Quindi, ad esempio, l'uso dell'operatore di negazione davanti ad una parentesi significa che l'operatore unario stesso si applica **dopo** avere svolto le operazioni indicate, e non alla variabile immediatamente seguente: tipicamente verrà complementato il risultato di una funzione binaria o di una catena di funzioni.

Per pura curiosità, si propone al lettore anche la seguente implementazione logica della somma binaria definita alla sezione precedente. Un *full adder* logico, a livello circuitale, opera su tre bit in ingresso (gli addendi A e B e il riporto dello stadio precedente C_{in}) e fornisce in uscita, oltre al bit S che rappresenta la somma $A+B$, anche il riporto C_{out}. Per lo stadio che manipola i due LSB dei byte da sommare, il riporto è inizializzato a zero.

$$\begin{aligned} S &= A \text{ XOR } B \text{ XOR } C_{in} \\ C_{out} &= A \text{ AND } B \text{ OR } C_{in} \text{ AND } (A \text{ XOR } B) \end{aligned} \tag{3.4.5}$$

La tabella di verità seguente esplicita il funzionamento combinato delle due equazioni logiche che, come è assai facile verificare, realizzano correttamente la somma binaria con riporto di A e B.

C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

[7] Dal punto di vista dell'elettronica digitale integrata, la più importante ricaduta tecnologica di tale intrinseca ridondanza è data dalla possibilità di realizzare la totalità delle funzioni logiche usando unicamente porte NAND (così è denominata in elettronica digitale la funzione AND con uscita negata), che a livello di chip sono realizzabili con esattamente *due* transistor a effetto di campo (FET), il minimo assoluto per una porta logica, che consente la massima densità di integrazione e un consumo energetico ridotto al minimo (anche se questo varia notevolmente secondo l'effettiva tecnologia di realizzazione dei transistor).

3.4.5 Forme Normali Canoniche e valutazione logica.

Per chiudere la sezione con un minimo cenno di approfondimento, volto unicamente a stimolare la curiosità del lettore e spalancare una piccola finestra sull'immenso universo delle logiche formali e sul mondo applicativo delle logiche programmabili, meritano una menzione le numerose *forme normali* utilizzate nell'ambito delle reti logiche e relativa ottimizzazione (principalmente utilizzante il metodo di Karnaugh o il forse meno noto Quine-McCluskey). Ad esempio, quelle fondamentali sono:

DNF: (Disjunctive Normal Form) o più mnemonicamente **SOP** (Sum Of Products), ossia la somma (OR) di prodotti (AND);

CNF: (Conjunctive Normal Form) o **POS** (Product Of Sums) ovvero la rappresentazione complementare come *prodotto di somme*;

PNF: (Parity Normal Form) ossia uno XOR di prodotti.

Se infatti volessimo proseguire il discorso iniziato con le tabelle presentate ad inizio sezione analizzando anche funzioni booleane arbitrarie ternarie e quaternarie, ci troveremmo subito alle prese con il problema noto come *esplosione combinatoria*: le possibili funzioni di tre variabili booleane ammontano a $DR(2, 2^3) = 256$, mentre quelle con quattro bit in input sono ben $DR(2, 2^4) = 65.536$. Vediamo a titolo puramente illustrativo una versione tabulare fortemente ridotta delle funzioni ternarie $f_j : \{0,1\}^3 \to \{0,1\}$, con evidenziate in rosso le familiari AND, XOR, OR (rispettivamente $f_1(A,B,C)$, $f_{105}(A,B,C)$ e $f_{127}(A,B,C)$):

#	A	B	C	0	1	2	...	105	...	127	...	254	255
0	0	0	0	0	0	0		0		0		1	1
1	0	0	1	0	0	0		1		1		1	1
2	0	1	0	0	0	0		1		1		1	1
3	0	1	1	0	0	0		0		1		1	1
4	1	0	0	0	0	0		1		1		1	1
5	1	0	1	0	0	0		0		1		1	1
6	1	1	0	0	0	1		0		1		1	1
7	1	1	1	0	1	0		1		1		0	1

Va premesso che nessuna CPU di normale diffusione (a fortiori gli storici 8 bit di cui ci occupiamo qui!) implementa nativamente funzioni logiche atomiche con più di due operandi, e che in tali contesti il modo in assoluto più efficiente per implementarle consiste nel precalcolarne tutte le possibili tabelle di verità e collocarle poi in un array in memoria. Se volessimo ad esempio mappare tutte le possibili funzioni quaternarie in una LUT[8] con entry a 16 bit occorrerebbe il doppio dell'intera memoria RAM di un Commodore 64. Ciascuno dei formalismi polinomiali logici sopra elencati consente di esprimere funzioni logiche arbitrarie di più variabili tramite una combinazione *minimale* di sole *coppie di funzioni elementari binarie*, in questo caso organizzate su due livelli (es. «prodotto» di «somme»), il che è elemento di base fondamentale per la sintesi logica e la configurazione di logiche programmabili, ma anche sfruttabile in ambito software e firmware[9].

Consideriamo ad esempio la funzione quaternaria il cui output è codificato dal numero decimale 23.467, la cui espressione più sintetica è $f_j(A,B,C,D) = \langle 1,3,4,6,7,8,10,12,14,15 \rangle$, ossia la lista ordinata dei soli valori in input (indicati ancora in decimale, secondo l'ordine fissato per le variabili $A...D$ come $MSB...LSB$) ai quali corrisponde un output 1:

[8] LUT è acronimo di Look-Up Table, la forma in assoluto più efficiente per realizzare una funzione tabulare sotto forma di semplice array, indicizzato direttamente con l'intero richiesto come input.

[9] Uno dei compiti più comuni per chi sviluppa firmware embedded, in special modo nel settore dell'automazione logica, è proprio quello di ottimizzare implementazioni di calcolo di funzioni booleane a più valori in modo che impieghino *il minor numero possibile* di istruzioni atomiche e maschere binarie, eventualmente appoggiandosi anche a LUT intermedie per bilanciare tra occupazione di memoria e velocità esecutiva secondo i parametri e le risorse di progetto.

#	A	B	C	D	$f(A,B,C,D)$
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	0
6	0	1	1	0	1
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	0
10	1	0	1	0	1
11	1	0	1	1	0
12	1	1	0	0	1
13	1	1	0	1	0
14	1	1	1	0	1
15	1	1	1	1	1

(3.4.6)

Le corrispondenti forme normali sono le seguenti, convenendo in questo caso per una maggiore leggibilità e compattezza dell'espressione che $\text{NOT}(X) = \overline{X}$, $\text{AND} = \wedge$ e $\text{OR} = \vee$:

$$\begin{aligned} &\text{DNF/SOP: } f(A,B,C,D) = A \wedge \overline{D} \vee \overline{A} \wedge \overline{B} \wedge D \vee B \wedge C \vee B \wedge \overline{D} \\ &\text{CNF/POS: } f(A,B,C,D) = (A \vee B \vee D) \wedge (\overline{A} \vee B \vee \overline{D}) \wedge (\overline{B} \vee C \vee \overline{D}) \end{aligned} \tag{3.4.7}$$

Naturalmente questo banalissimo esempio ha solo scopo didascalico e di immediata comprensibilità, perché tabelle di verità di dimensioni così ridotte si prestano perfettamente ad essere memorizzate sotto forma di LUT, ma deve essere ben chiaro che scalando ad applicazioni real world con decine o addirittura centinaia di variabili logiche i concetti di sintesi delle tabelle e i formalismi qui illustrati rimangono identici.

Si nota immediatamente, ad esempio, che la DNF è tale da fornire output 1 quando *almeno uno* dei prodotti indicati assume tale valore: questo ci riconduce immediatamente al concetto di *short circuit evaluation* che dovrebbe essere ben noto ai programmatori in C e in vari HLL. La valutazione dell'espressione, in questo caso, può essere interrotta non appena si trova la prima AND il cui risultato è un valore unitario, il quale diventa il valore finale dell'output *a prescindere* dal risultato di ogni altra operazione. Analoga considerazione si applica alla forma complementare del prodotto di somme, laddove il primo OR nullo interrompe la catena e porta necessariamente ad output parimenti nullo. Si vuole sottolineare che la fondamentale differenza tra la sintesi logica e la valutazione dinamica in firmware o software sta proprio nella intrinseca serialità di quest'ultima, mentre parlando di porte logiche il parallelismo è la norma (si veda anche la figura 3.4.1, che mostra con estrema chiarezza i due livelli OR e AND che caratterizzano l'implementazione circuitale corrispondente), pertanto l'ottimizzazione della valutazione di espressioni percorre vie sostanzialmente diverse nei due ambiti. Si noti in figura il classico *bus degli input*, tipico delle rappresentazioni nell'ambito delle reti logiche e dell'elettronica digitale integrata, che prevede per default tutte le variabili e tutti i loro negati.

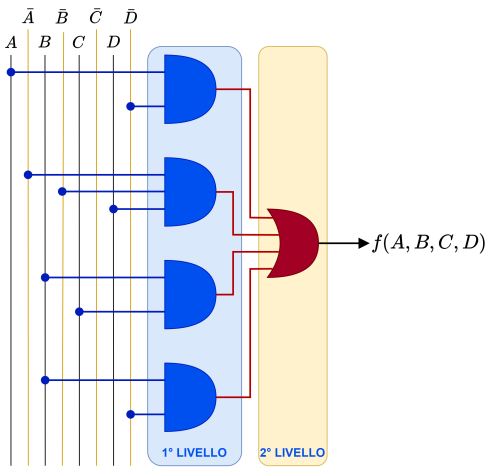


Figura 3.4.1: Implementazione circuitale della espressione DNF/SOP 3.4.7.

3.4.5.1 Forme Normali avanzate: l'operatore ternario ITE.

Nonostante la forzata stringatezza della trattazione, sembra inevitabile notare che le forme normali appena presentate operano in realtà su una combinazione tra le variabili di input così come si presentano e sulla loro *negazione*, quindi implicitamente le «coppie» di funzioni dei due livelli includono anche una **terza operazione**: la negazione unaria. Se in elettronica digitale integrata o discreta questo raramente costituisce un problema e

di solito non pare neppure degno di menzione, in alcuni frangenti (specialmente legati alla valutazione logica in software e firmware) ciò non risulta desiderabile: a tale scopo sono stati sviluppati altri formalismi, che possono essere espressi in modo da valutare esclusivamente le variabili in input *in forma nominale*, senza ricorso alla negazione. Come esempio di formalismo siffatto, tra le altre forme normali più specialistiche, la INF (If-then-else Normal Form) è quella associata alla funzione logica dalla semantica probabilmente più intuitiva in assoluto per un informatico ed è legata una funzione ternaria: la If-Then-Else (ITE), ovvero IF A THEN β ELSE γ.

$$\text{ITE}(A, \beta, \gamma) := A \text{ AND } \beta \text{ OR } \neg A \text{ AND } \gamma \tag{3.4.8}$$

La semantica della funzione può essere espressa in modo ancora più immediato come segue:

$$\text{ITE}(A, \beta, \gamma) := \begin{cases} \beta & \text{se } A = TRUE \\ \gamma & \text{se } A = FALSE \end{cases} \tag{3.4.9}$$

Appare evidente la similitudine con l'*operatore ternario* del linguaggio C, sebbene i possibili risultati siano qui limitati ai soli due valori booleani $\{F, T\}$ o $\{0, 1\}$.

Definizione di INF: se la ITE() è espressa in modo che A sia *sempre* una variabile booleana non negata mentre β e γ sono alternativamente espressioni contenenenti *variabili booleane, funzioni logiche* oppure *costanti*, allora si ha una ulteriore *forma normale*, appunto la INF, in grado di esprimere ogni altra possibile funzione logica binaria, come mostrato nella seguente tabella:

Pos.	ITE()	Funzione	Funzione	ITE()	Pos.
0	0	$FALSE$	$TRUE$	1	15
1	ITE$(A, B, 0)$	A AND B	$\neg(A$ AND $B)$	ITE$(A, \neg B, 1)$	14
2	ITE$(A, \neg B, 0)$	$\neg(A \Rightarrow B)$	$A \Rightarrow B$	ITE$(A, B, 1)$	13
3	ITE$(A, 1, 0)$	A	$\neg A$	ITE$(A, 0, 1)$	12
4	ITE$(A, 0, B)$	$\neg(B \Rightarrow A)$	$B \Rightarrow A$	ITE$(A, 1, \neg B)$	11
5	ITE$(B, 1, 0)$	B	$\neg B$	ITE$(B, 0, 1)$	10
6	ITE$(A, \neg B, B)$	A XOR B	$\neg(A$ XOR $B)$	ITE$(A, B, \neg B)$	9
7	ITE$(A, 1, B)$	A OR B	$\neg(A$ OR $B)$	ITE$(A, 0, \neg B)$	8

Naturalmente, nell'ambito della programmazione ordinaria (low level o meno) su CPU di normale diffusione, il costo implementativo di tali funzioni è quasi sempre svantaggioso rispetto all'utilizzo delle *istruzioni atomiche* corrispondenti, a meno che non si stia implementando un motore per valutazione logica, un sistema di automazione logica o un sottosistema di constraint programming, destinati a manipolare migliaia di espressioni logiche complesse. Tuttavia, si tratta di un esempio di grande impatto didattico per illustrare la intrinseca ridondanza e il dualismo delle funzioni logiche in generale.

3.4.6 Diagrammi di decisione binari.

Come ultimo, intuitivo esempio di formalismo a cui corrisponde in modo immediato una struttura dati per il calcolo booleano avanzato, si vuole semplicemente mostrare l'aspetto di un *albero di decisione binario*, in

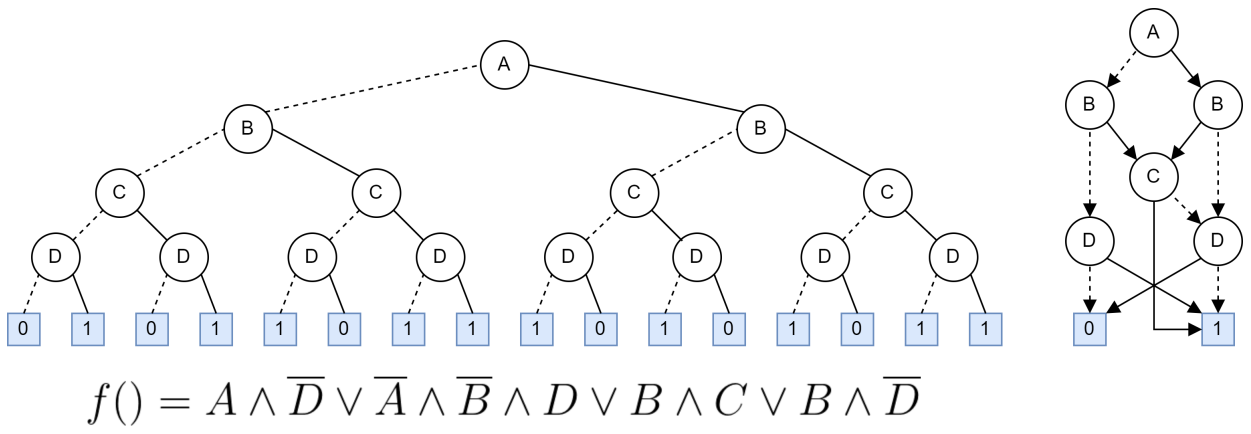


Figura 3.4.2: Binary Decision Diagram completo (sx) e ROBDD corrispondente (dx).

questo caso corrispondente alla funzione 3.4.6, nella figura (3.4.2), che illustra a sinistra l'albero *completo* e a destra il corrispondente Ordered Binary Decision Diagram ridotto (ROBDD). Atteso che, delle due linee uscenti da ciascun nodo, quella continua (che nell'albero completo porta sempre al figlio *destro*) rappresenta il percorso di valutazione da seguire se la variabile V corrispondente al nodo è pari a 1, mentre ovviamente la linea tratteggiata (sistematicamente verso il figlio *sinistro*, sempre nell'albero completo) indica il percorso complementare, abbiamo quindi ad esempio che alla quaterna di valori $A = 1, B = 0, C = 0, D = 0$ corrisponde univocamente il percorso che dal nodo radice A porta al figlio *destro* B, da qui essendo B nullo al figlio *sinistro* C, per la medesima ragione avendo $C = 0$ si va al figlio *sinistro* D, e infine essendo anche D nullo si raggiunge il suo figlio *sinistro* ossia il nodo terminale con valore 1, che è esattamente quanto indicato dalla truth table in tale caso. E così via per ogni altro possibile percorso.

La valutazione nell'albero ridotto ROBDD procede esattamente nel medesimo modo, con le uniche minimali differenze che:

- la distribuzione dei nodi figli non è rigidamente prefissata, contrariamente all'albero completo, per il quale come visto vale sempre la convenzione FALSE=left, TRUE=right;
- per ogni percorso dalla radice ad un nodo terminale corrispondente ad una data combinazione delle variabili in input, possono esservi delle *condizioni di indifferenza* (si veda anche la nota 4 a pagina 15) ossia delle variabili che non compaiono nel percorso perché il loro valore non influenza il valore della funzione.

Ad esempio, consideriamo il ROBDD in figura 3.4.2 e partendo dal nodo radice percorriamo il ramo per $A = 0, B = 0$: ci troviamo direttamente al nodo D che porta infine al nodo terminale 0 per $D = 0$. Ciò può essere scritto evidenziando la condizione di indifferenza $f(0, 0, X, 0) = 0$ la cui espansione equivale alle seguenti linee nella truth table:

#	A	B	C	D	$f(A, B, C, D)$
0	0	0	0	0	0
2	0	0	1	0	0

(3.4.10)

Si lascia come utile esercizio per il lettore interessato la **verifica completa** della truth table data 3.4.6, a partire dai percorsi presenti nel RODBB, con esplicitazione delle *condizioni di indifferenza* che sono un elemento cruciale per l'ottimizzazione e l'accorpamento dei casi.

Senza addentrarci ulteriormente nella vasta complessità di questo affascinante argomento, si affida all'intuito del lettore la valutazione dell'enorme risparmio di risorse introdotto dall'uso di ROBDD nei quali il numero di nodi si riduce drasticamente, mantenendo però intatta ogni informazione relativa alla funzione booleana rappresentata. Tali strutture dati e gli algoritmi elaborati per manipolarle efficientemente rendono ampiamente trattabili sistemi logici con un numero di stati molto rilevante: già nella prima metà degli anni Novanta si parlava di 10^{21} stati, una esplosione combinatoria di tutto rispetto. Oggi sono ampiamente gestibili in modo simbolico sistemi con 10^{32} stati o superiori, usando normalissimo ed economico hardware mainstream.

Parte II

Programmazione Assembly MCS6502/10

Capitolo 4

La famiglia di CPU MCS65xx.

La famiglia di CPU 65xx, ideata nella prima metà degli anni Settanta, consta di numerosi membri a 40 e 28 pin, i più diffusi dei quali sono senza dubbio il 6502 e il 6510. Concepiti inizialmente come snap-in replacement per il Motorola MC6800, col quale mantengono in generale la compatibilità pin-to-pin (soprattutto il capostipite MCS6500), supportano un set di 56 istruzioni e sono stati utilizzati in un numero rilevante di home computer, board didattiche, schede di controllo industriali. Il 6510 esiste in varie versioni, la più diffusa delle quali rimane quella legata al Commodore 64, l'home computer più venduto della storia.

Le differenze tra 6502 e 6510 sono minime: quest'ultima CPU offre un port di I/O integrato, con il relativo registro di controllo, mappati ai primi due indirizzi assoluti (0000h e 0001h rispettivamente per il DDR, Data Direction Register, e per il registro di I/O vero e proprio). Fin dal datasheet viene sottolineato come tale caratteristica, unita all'indirizzamento indicizzato zero-page e alla gestione degli interrupt, consenta la realizzazione di sofisticate jump tables per la gestione di segnali logici provenienti direttamente dalle periferiche: una architettura hardware e software effettivamente sfruttata in una moltitudine di sistemi di controllo e schede didattiche.

4.1 Bibliografia minimale.

Sarà utile avere a disposizione qualche testo sull'Assembly 6502/6510, oltre ai datasheet originali (già resi disponibili in PDF sul gruppo RP Italia). L'Autore suggerisce di partire da Zaks [Zak83] e Leventhal [Lev86], seguiti da Andrews [And85] e Butterfield [But86]. Dato il taglio marcatamente introduttivo del presente lavoro, saranno indicati anche testi di base come [Osb80], [Jon84], [Smi85] e magari il Sinclair [Sin84] affiancato dal classico [Dav84] e da un testo di introduzione alla logica della programmazione come il notissimo Sprankle [SH11]. Per un livello più avanzato si possono suggerire [Sut85], [Eng85] e naturalmente [Zak82].

4.2 Architettura del processore MCS6510.

I valori esadecimali saranno nel seguito contrassegnati con una lettera h finale, come in $1234h$, e nei listati Assembly con il prefisso $, come in `$1234`. Per i valori binari, si userà ove necessario il suffisso b come in $11001010b$ e nei listati il prefisso `%01011100`.

La figura (4.2.1) mostra uno schema logico semplificato della CPU MCS6510, conforme al datasheet. Si notano i bus interni, tutti rigorosamente ad 8 bit e **inaccessibili** all'utente; i registri principali a disposizione del programmatore (parimenti a 8 bit, in arancione); e una serie di registri e buffer **interni** (in celeste, anch'essi inaccessibili dall'utente) necessari al funzionamento del microprocessore. Si noti anche che il registro Program Counter, che è un singolo registro a 16 bit (peraltro *l'unico* di tale dimensione presente nella CPU), è riportato evidenziando le sue due metà alta e bassa, rispettivamente PCH e PCL, per meglio sottolineare il flusso logico dei dati verso i due bus di indirizzo interni, rispettivamente ADH, ADL (ADdress High, ADdress Low).

Si notano immediatamente i principali blocchi logico-funzionali del microprocessore: l'unità di decodifica delle istruzioni (il vero e proprio *cuore* della CPU), circondata da unità accessorie come il blocco di gestione degli interrupt e lo stadio di clock, e l'unità artimetico-logica ALU. Per ovvie ragioni di razionalizzazione dello schema si sono omesse le numerose connessioni tra l'unità di decodifica e i singoli registri: si dà per acquisito che tutti i registri e i blocchi della CPU interconnessi ai vari bus sono sempre interfacciati da buffer 3-state bidirezionali o monodirezionali (secondo la natura e funzione del registro), rappresentati nello schema tramite frecce a larghezza bus e - sul silicio - totalmente gestiti dall'unità di decodifica tramite linee individuali di abilitazione e gestione della direzione ($R/\overline{W}$). In questo modo, per un banale esempio, al momento dell'esecuzione di una istruzione che copia il contenuto dell'Accumulatore nel registro Y tali registri saranno connessi al bus interno sotto il

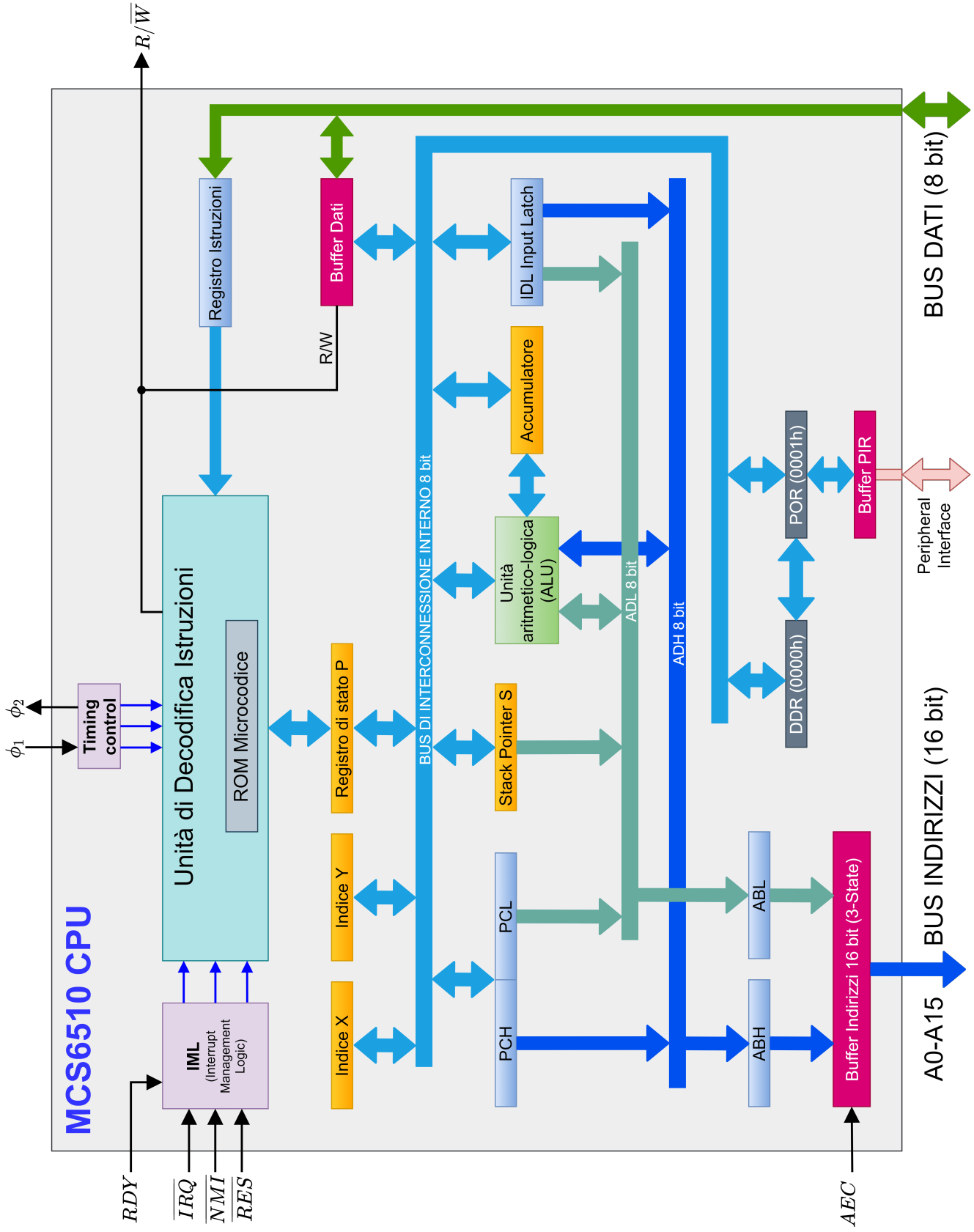


Figura 4.2.1: Schema logico semplificato della CPU 6510.

controllo dell'unità di decodifica, rispettivamente l'Accumulatore abilitato in lettura e il registro Y abilitato in scrittura mentre tutti gli altri registri e stadi rimangono in stato di alta impedenza, disconnessi dal bus interno.

Conseguenza immediata dello schema logico, e in particolare della suddivisione interna dell'indirizzamento su due registri a 8 bit ABH e ABL, è il fondamentale concetto di *pagina di memoria*. Una pagina è una sequenza contigua di 256 indirizzi, caratterizzati dall'avere il *medesimo byte di indirizzo alto* (registro interno ABH). Quindi, esprimendo gli indirizzi in esadecimale, la pagina 0 è caratterizzata dagli indirizzi $0000 \div 00FF$ estremi inclusi, la pagina 1 (riservata per design allo stack) ha indirizzi $0100 \div 01FF$ e più in generale le varie pagine di memoria sono individuate dal *byte più significativo* dell'indirizzo: 02, 03, ...FD, FE, FF. Particolare rilevanza riveste la *pagina zero*, in quanto gode di una modalità di accesso privilegiato, maggiormente efficiente rispetto all'accesso ad altre locazioni di memoria. Si noti, inoltre, che in una delle versioni più recenti del 6510 (commercializzata a partire dal 1982) le prime 256 locazioni di memoria RAM (a rigore, 254 perché le prime due sono sempre riservate al port di I/O) sono integrate nella CPU, rendendo ancora più efficiente l'accesso.

Inoltre talune istruzioni (principalmente di salto) risultano penalizzate in termini di tempi di esecuzione quando la locazione di destinazione non risiede nella *medesima pagina* di memoria di quella di partenza. Anche negli indirizzamenti indicizzati (si veda 4.2.4 a pagina 38) si può verificare condizionatamente tale salto di pagina: se supponiamo ad esempio di utilizzare l'indirizzo assoluto A077h come indirizzo base e un indirizzamento indicizzato col registro X, quando tale registro assumerà valori superiori a 88h, si avrà un indirizzo risultante superiore ad A100h e quindi appartenente alla pagina successiva.

Le affermazioni precedenti, intuitivamente, sono giustificate dal fatto che i bus interni alla CPU sono a 8 bit ed esiste un unico data latch interno IDL, quindi per un indirizzo assoluto completo (16 bit) o anche per un cambio di pagina è necessario caricare *sequenzialmente* i due registri interni di indirizzo ABH e ABL, il che richiede almeno un ciclo di clock in più rispetto alle operazioni nelle quali ABH non varia.

Le istruzioni del set e i relativi operandi occupano da un minimo di un byte ad un massimo di tre byte; i tempi di esecuzione variano tra due e sette cicli di clock: il che significa, ad esempio, che su un Commodore 64 con clock a $1,022727$ MHz (quindi con tempo di ciclo pari a $977,778$ ns) i tempi di esecuzione variano tra circa $1,96$ e $6,84$ µs per ogni singola istruzione.

4.2.1 FDE: Fetch, Decode, Execute.

Il ciclo vitale della CPU, scandito dal clock di sistema, prevede tre fasi principali che si ripetono iterativamente:

Fetch: a partire da un indirizzo iniziale *hardcoded*[1], viene prelevata la «prossima» istruzione dalla memoria, attraverso il bus di sistema. L'istruzione (un valore numerico, in questo caso a 8 bit) viene caricata nell'apposito registro istruzioni, connesso al bus dati interno, e da qui riversata nell'unità di decodifica: la parte più complessa e importante della CPU, che come già accennato occupa di gran lunga la maggior parte dell'area sul chip.

Decode: l'unità di decodifica è una complessa circuiteria composta da logiche combinatorie e sequenziali, nonché da memorie di sola lettura che contengono tabelle e istruzioni atomiche (microcodice) necessarie ad implementare istruzioni complesse presenti nel set. Essa ha lo scopo di gestire lo stato della CPU (e indirettamente del sistema connesso, in particolare della memoria esterna) come conseguenza dell'esecuzione di una data istruzione. La fase di decodifica, che nelle CPU tradizionali può richiedere numerosi cicli di clock, predispone le modifiche e abilita i singoli gate di lettura e scrittura che interconnettono i registri ai bus interni.

Execute: in questa fase vengono posti in atto gli effettivi cambiamenti di stato che coinvolgono i registri e i bus esterni. Ad esempio, in una istruzione di trasferimento tra registri interni, essi vengono messi in comunicazione (in lettura e scrittura rispettivamente) sul bus interno, evitando conflitti con altri registri e unità e la copia dei dati viene effettivamente eseguita. Viene aggiornato il Program Counter, l'unico registro a 16 bit, che contiene l'indirizzo della prossima istruzione (che può essere effettivamente consecutiva in memoria rispetto a quella eseguita, o la locazione specificata come destinazione di una istruzione di salto). A questo punto è possibile passare alla «successiva» istruzione, ricominciando quindi il ciclo dalla fase di fetching.

4.2.2 I registri accessibili dall'utente del 6502/6510A.

A registro Accumulatore. La CPU di nostro interesse, contrariamente alla maggioranza dei RISC di più moderna concezione, dispone di un solo registro Accumulatore e questo, anche rispetto a CPU 8 bit coeve

[1] Nel nostro caso, all'avvio e dopo un reset, l'indirizzo della prima istruzione da eseguire viene sempre caricato dalle due locazioni assolute di memoria FFFCh e FFFDh prefissate in fase di progetto del silicio, ovviamente in ordine *little endian*. Tale approccio è detto **vettorizzazione** ed è estremamente flessibile: i progettisti sono liberi di inserire in tali locazioni un qualsiasi indirizzo arbitrario, a seconda delle necessità del sistema operativo eventualmente presente, della possibilità di utilizzare cartucce o altri mezzi di memorizzazione non volatili per modificare con facilità le condizioni di avvio della scheda o del sistema, eccetera.

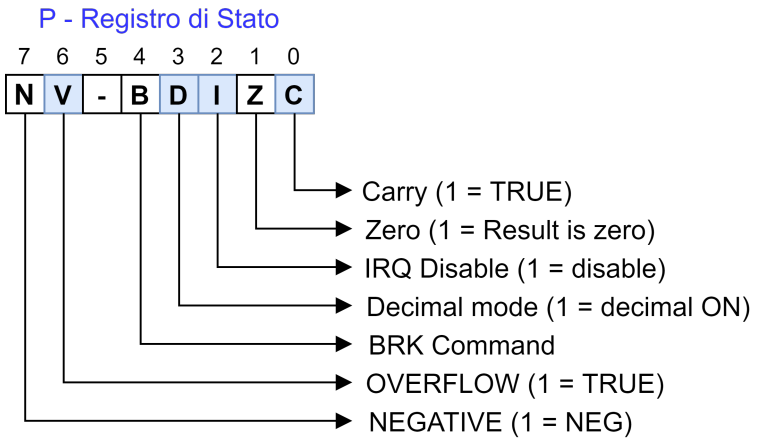


Figura 4.2.2: Il registro di stato P del processore.

(in primo luogo lo Z80), rende leggermente più difficile scrivere codice assembly efficiente. In ogni caso, l'Accumulatore è la destinazione (implicita o meno) per numerose istruzioni previste dalla ISA e in esso avviene la maggioranza delle operazioni.

Indici la coppia di registri a 8 bit **X** e **Y** è importante per tre scopi fondamentali:

1. Scambio di dati diretto con l'Accumulatore, tramite istruzioni dedicate;
2. Contatori per i loop, con istruzioni di decremento e incremento unitario specifiche (curiosamente non previste per l'Accumulatore);
3. Indicizzazione di array e tabelle in memoria, entro apposite modalità di indirizzamento indicizzato nelle quali la CPU automaticamente *aggiunge* il contenuto attuale del registro X o Y ad un dato indirizzo di memoria di base. Questo spiega, peraltro, la connessione evidenziata nello schema logico tra l'ALU e il bus interno degli indirizzi.

S il registro stack pointer. Si noti che anche tale registro consta di soli 8 bit, e quindi la profondità massima di stack è pari a 256 locazioni. Al momento del trasferimento di S nei buffer di indirizzo, il buffer **ABH** per la parte alta dell'indirizzo (pagina) viene automaticamente caricato col valore binario hardcoded 00000001b. Ne consegue che lo stack occuperà *sempre* gli indirizzi di memoria assoluti compresi tra 0100h e 01FFh in qualunque architettura di sistema: si tenga inoltre presente che, ad ogni PUSH, lo stack pointer (inizializzato a FFh dopo il reset) viene *decrementato* di una locazione, fino al limite di 0 oltre il quale una ulteriore PUSH causerebbe il *wraparound* del registro e uno stack overflow (non gestito in hardware, quindi in pratica «una ricetta per il disastro», come dicono gli amici oltremanica).

P il registro di stato del processore. Come visibile in figura (4.2.2), esso consta di 7 flag, dei quali 4 (evidenziati dallo sfondo celeste in figura) sono (parzialmente) manipolabili dall'utente tramite istruzioni dedicate. Tali flag vengono aggiornati dall'unità di decodifica dopo la maggior parte delle istruzioni: nel seguito vedremo anche in dettaglio, per ogni istruzione, quali sono i flag influenzati.

- **N** viene denominato bit del segno (sig**N** flag), ma in realtà quando viene aggiornato copia lo stato del MSB del registro interessato, indipendentemente dal fatto che il contenuto sia effettivamente un numero in complemento a due o non segnato. L'unica significativa eccezione a questa regola generale è l'istruzione di scorrimento (shift) a destra LSR, che azzera *sempre* il flag N. Tale flag non è manipolabile dall'utente tramite istruzioni dirette che possano azzerarlo o impostarne a 1 il valore.
- **V** ossia o**V**erflow flag, segnala un overflow aritmetico dopo una operazione di addizione o sottrazione, perché il risultato è tale da non poter essere contenuto in un singolo byte. Logicamente, esiste una istruzione specifica per azzerarlo, ma non una corrispondente per settarlo, al contrario degli altri tre bit di stato manipolabili dall'utente.
- **B** ovvero **B**reak flag, viene impostato quando nel flusso di esecuzione il processore incontra una istruzione specifica, denominata BRK. Ha rilevanza pratica solo nelle sequenze di interrupt e non è manipolabile dall'utente.
- **D** ovvero **D**ecimal flag, imposta la modalità di funzionamento BCD quando posto a 1. Tale modalità sarà spiegata in dettaglio nel seguito.

I ovvero Interrupt flag. Quando il microprocessore riceve una richiesta di interrupt da una periferica tramite il segnale logico $\overline{IRQ}$, esamina lo stato di questo bit: se è posto a 1, l'interrupt viene ignorato. Fondamentale per la creazione di sezioni critiche nel codice, che non possono essere interrotte per garantire tempi di esecuzione costanti e predicibili in funzione di una temporizzazione stringente (tipicamente per la comunicazione con una periferica esterna). Vista la sua natura, è chiaramente anche azzerabile e impostabile dal programmatore tramite apposite istruzioni.

Z ossia Zero flag. Si presti attenzione alla semantica invertita: viene posto a 1 se il risultato di una operazione è pari a zero, ossia nullo. Quindi è normalmente pari a zero, se il risultato dell'ultima operazione o trasferimento **non** è nullo. Come nella maggioranza delle CPU, non è manipolabile dall'utente con istruzioni specifiche.

C ovvero Carry flag, è il bit del riporto nelle operazioni aritmetiche e costituisce il «nono bit» nelle rotazioni e negli scorrimenti, come chiaramente esemplificato nella prima parte del testo (si veda la ss. 3.3.1 a pagina 24). Esistono apposite istruzioni per l'azzeramento e l'impostazione arbitrari del carry, tipicamente usate preliminarmente alle operazioni aritmetiche e agli shift/rotazioni.

Il microprocessore 6510 offre inoltre, come già accennato, un ulteriore registro mappato alla locazione assoluta 0000h: si tratta del **DDR**, Data Direction Register, che controlla il comportamento dei singoli bit del port di I/O vero e proprio, mappato alla locazione successiva. Ciascuna locazione del DDR contiene un valore 0 se il corrispondente bit del port di I/O funziona come *input*, e un valore 1 per l'output. Si noti come questa convenzione è **opposta** a quella (maggiormente mnemonica) utilizzata universalmente qualche anno dopo dalla maggioranza dei progettisti di microcontroller e CPU, tale che anche graficamente sia immediato associare 1=I (Input) e 0=O come Output.

Si presti attenzione al fatto che, nelle versioni più diffuse del 6510 (inclusa quella installata nei Commodore 64), i bit 7 e 6 del port di I/O *non sono implementati* e i relativi pin della CPU sono invece utilizzati per le funzioni di `READY` (utilizzata in hardware per la sincronizzazione e il DMA, assieme al segnale `AEC` che pone in 3-state i buffer di indirizzo isolando così la CPU dal relativo bus) e $\overline{NMI}$ (Non-Maskable Interrupt, linea di interrupt hardware ad alta priorità non soggetta al flag **I** sopra descritto).

4.2.3 Altri registri fondamentali.

PC è il registro Program Counter, rappresentato in figura (4.2.1) nelle sue due metà PCH e PCL per meglio evidenziare la logica delle sue connessioni ai due bus di indirizzo interni. Vale la pena di evidenziare nuovamente che tale registro è *l'unico registro a 16 bit effettivi contenuto nella CPU*, che consente di referenziare *qualsiasi* indirizzo nel range indirizzabile di 64KiB. Normalmente viene incrementato in automatico dalla sezione di decodifica per puntare all'istruzione successiva in memoria rispetto a quella appena eseguita, ma come già accennato viene anche modificato dalle istruzioni di salto (condizionato o assoluto) per alterare di fatto la sequenza dell'esecuzione, realizzando così indirettamente (tra l'altro) cicli e scelte come sancito dal teorema di espressività minimale dei linguaggi di programmazione [BJ66].

RI ossia Registro Istruzioni. Dopo la fase di fetch, contiene il cosiddetto *opcode* a 8 bit corrispondente all'istruzione correntemente prelevata dalla memoria. Tale valore consegna all'unità di decodifica due fondamentali informazioni:

1. Che tipo di operazione si sta per compiere;
2. Quanti *operandi* sono eventualmente richiesti, ossia quanti altri byte (zero, uno o due) è necessario prelevare ancora dalla memoria per completare l'operazione.

RD ovvero Registro Dati. Gestisce bidirezionalmente (sotto il controllo della linea logica $R/\overline{W}$) il transito dei dati a 8 bit dal bus esterno verso il bus interno.

4.2.4 Modalità di indirizzamento.

Ciascuna delle 56 istruzioni del 6502 supporta *una o più* delle 13 modalità di indirizzamento totali previste dall'architettura. Ciascuna specifica combinazione di istruzione e modalità è individuata da un *opcode* univoco, per un totale di 151 diversi opcode documentati[2]: tuttavia, di norma il programmatore Assembler non ha necessità di memorizzare tutti i possibili codici previsti dal *linguaggio macchina*. L'uso di un Assembler consente infatti l'uso di *mnemonici*, ovvero gruppi di tre lettere come `STA` o `INX` che individuano ciascuna delle 56 istruzioni a prescindere dalla modalità di indirizzamento. Tali modalità influenzano la lunghezza complessiva dell'istruzione, dovuta al numero di operandi (zero, uno, due). Gli indirizzi saranno normalmente indicati nel resto del documento come *addr8* (indirizzo ad un byte, es. pagina zero) e *addr16* (indirizzo assoluto, due byte). Sono supportate le seguenti modalità:

[2] Non affronteremo in questa sede, per vari ordini di ragioni, la questione degli opcode non documentati.

- **Indirizzamento implicito:** si tratta della forma in assoluto più semplice, che prevede operazioni unicamente sui registri interni della CPU e non richiede alcun operando. Le istruzioni con indirizzamento implicito occupano un solo byte in memoria e sono normalmente anche le più veloci in assoluto, richiedendo solo due cicli di clock (il minimo per le CPU considerate): `CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS`. Vi sono inoltre ulteriori istruzioni ad indirizzamento implicito, che tuttavia richiedono più dei due cicli di clock minimali: `BRK, PHA, PHP, PLA, PLP, RTI, RTS`. In totale sono ben 24 su 56 le istruzioni ad indirizzamento implicito. Ricordiamo che il significato di tutte le istruzioni qui elencate sarà dettagliato nella prossima sezione.

- **Indirizzamento in Accumulatore:** strettamente analogo al precedente, dal quale viene distinto formalmente solo per quelle istruzioni che supportano anche altri modi di indirizzamento. Infatti si applica unicamente alle 4 istruzioni di scorrimento e rotazione: `ASL, LSR, ROL, ROR`.

- **Indirizzamento immediato:** l'operando previsto (un singolo byte) viene prelevato dalla memoria, alla locazione immediatamente successiva all'istruzione corrente. Tutte le istruzioni con indirizzamento immediato occupano due byte in memoria: l'opcode seguito dal byte dell'operando. Tali operandi vengono universalmente denotati con un simbolo # prefisso nei sorgenti Assembly, come ad esempio `LDA #$20` che ha l'effetto di caricare in Accumulatore il valore esadecimale 20h, ossia 32 decimale. Vi sono 11 istruzioni che supportano l'indirizzamento immediato, sono tutte aritmetico-logiche o di confronto: `ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA, SBC`.

- **Indirizzamento in pagina zero:** come già ricordato, tale modalità di indirizzamento è maggiormente efficiente rispetto all'accesso a qualsiasi altra locazione di memoria: l'occupazione di memoria delle istruzioni che supportano tale indirizzamento è pari a due soli byte. Vi sono 21 istruzioni che supportano l'indirizzamento in pagina zero: `ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, STY`. Si presti attenzione al fatto che sulle architetture più diffuse (es. Commodore 64) i progettisti hanno sfruttato al massimo lo spazio disponibile in pagina zero per il sistema e il firmware residente: le locazioni ufficialmente documentate e disponibili sono decisamente poche (tipicamente quelle comprese tra FBh ed FFh, estremi inclusi, oltre a pochi altri byte sparsi).

- **Indirizzamento assoluto:** per accedere direttamente alle locazioni non in pagina zero, comprese tra 100h e FFFFh (si ricordi tuttavia che le locazioni 100h-1FFh, estremi inclusi, sono immutabilmente dedicate allo **stack** per design) si deve utilizzare l'indirizzamento assoluto, più lento e con maggiore occupazione di memoria (un byte per l'opcode e due byte per l'indirizzo little-endian). Le istruzioni che supportano tale modalità di indirizzamento sono esattamente le stesse 21 per le quali è possibile usare l'indirizzamento in pagina zero, più l'istruzione di salto `JMP` e la chiamata di subroutine `JSR`.

- **Indirizzamento relativo:** tale modalità riguarda esclusivamente le 8 istruzioni di salto condizionato (*branch*) ed è limitata come destinazione a locazioni situate entro +127 o -128 bytes rispetto a quella corrente. Naturalmente, nel caso in cui la destinazione sia necessariamente situata al di là di tali limiti, è sempre possibile inserire un salto incondizionato, con indirizzamento assoluto. Le istruzioni interessate sono `BCC, BCS, BEQ, BMI, BNE, BPL, BVC, BVS` ed hanno tutte una lunghezza pari a 2 byte.

- **Indirizzamento indicizzato in pagina zero:** in questa modalità l'operando (1 byte) specifica una locazione di pagina zero (indirizzo di base), alla quale viene sommato l'attuale contenuto del registro X o Y per ottenere l'indirizzo assoluto. Tale indirizzo risultante *deve* essere a sua volta ancora in pagina zero: quindi valgono le restrizioni $addr8 + X \leq FFh$ e $addr8 + Y \leq FFh$ rispettivamente. Le istruzioni che ammettono l'uso del registro X con tale modalità di indirizzamento, che occupa due byte (opcode e operando) e nei sorgenti viene specificata con la notazione `OPCODE addr8,X` sono `ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA, STY` mentre vi sono solamente due istruzioni che fanno uso del registro Y, in modo piuttosto ovvio: `LDX, STX`.

- **Indirizzamento indicizzato assoluto:** come sopra, ma l'indirizzo può essere situato ovunque entro la memoria indirizzabile ed è richiesto un byte in più rispetto alla precedente modalità. La sintassi è: `OPCODE addr16,X` oppure `OPCODE addr16,Y`. Anche qui, ovviamente, la somma tra l'operando (indirizzo di base) e il contenuto del registro indice X o Y utilizzato non deve superare il limite dei 64KiB, altrimenti si ha un *wraparound* potenzialmente disastroso e non intercettabile in hardware. Le istruzioni che ammettono tale forma di indirizzamento usando il registro X sono: `ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA`. Quelle che ammettono l'uso del registro Y sono invece: `ADC, AND, CMP, EOR, LDA, LDX, ORA, SBC, STA`.

- **Indirizzamento indiretto:** questa peculiare modalità vettoriale, che in realtà fa uso di un *puntatore* analogo a quelli previsti dal linguaggio C, è supportata unicamente dall'istruzione di salto incondizionato JMP. La locazione assoluta di memoria indicata dai due operandi contiene due byte che, riassemblati

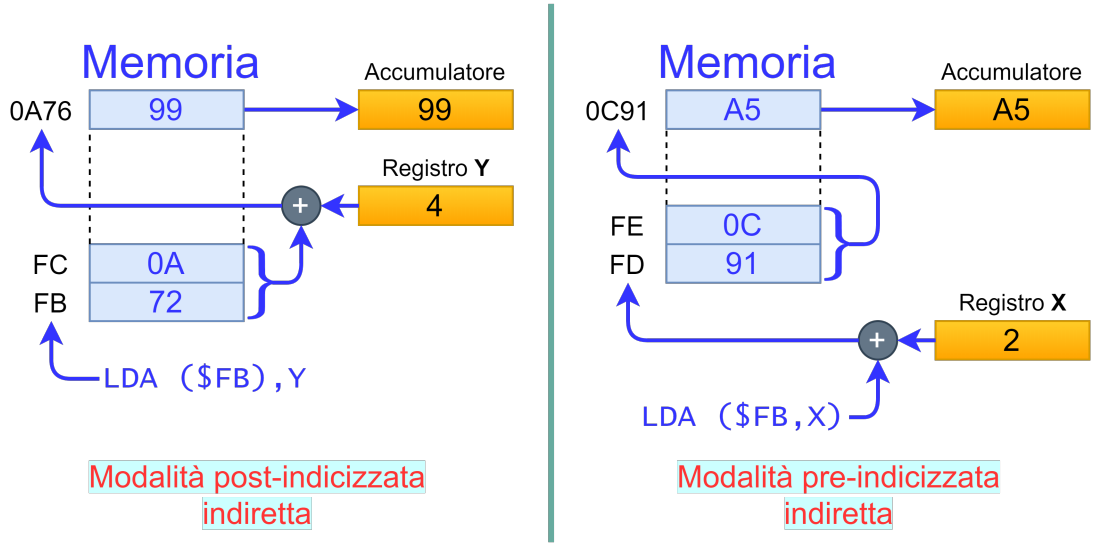


Figura 4.2.3: Modalità di indirizzamento indirette.

in ordine little-endian, indicano l'effettiva destinazione finale del salto. La sua destinazione primaria è la realizzazione di jump tables, che consentono di svincolare gli entry point principali del firmware di sistema (es. Kernal, interprete BASIC, firmware di ogni genere su cartridge...) dall'effettivo indirizzo della routine, che può variare di versione in versione secondo l'occupazione di memoria di altre subroutine, oppure può essere semplicemente sostituito a runtime con l'indirizzo di un software arbitrario scritto dall'utente. Esempio principe di questa fondamentale idea è lo stesso vettore di avvio della CPU 6502, che come già evidenziato carica al boot come prima locazione del Program Counter il **contenuto** delle locazioni di memoria FFFCh (LSB) e FFFDh (MSB) ed avvia l'esecuzione da tale indirizzo effettivo, di fatto ignoto al momento della progettazione del silicio.

- **Indirizzamento post-indicizzato:** la figura (4.2.3) mostra in modo chiaro un esempio di tale modalità di indirizzamento con puntatore in pagina zero. La similitudine più immediata per il lettore abituato ai linguaggi di alto livello è quella di un *puntatore ad array* del linguaggio C. Questa modalità di indirizzamento richiede un solo operando (indirizzo di pagina zero), fa uso del registro indice Y, viene denotata nel sorgente come `OPCODE (addr8),Y` e si articola come segue:

  1. Referenzia una locazione di pagina zero (non superiore a FEh) che, *assieme alla locazione immediatamente successiva*, **contiene** l'indirizzo assoluto di base di un array;
  2. A tale indirizzo a 16 bit viene poi aggiunto l'attuale contenuto del registro Y (offset): per tale motivo si parla di post-indicizzazione;
  3. L'indirizzo finale così ottenuto viene infine utilizzato come operando per l'istruzione corrente.

  Pertanto, nell'esempio mostrato in figura `LDA($FB),Y`:

  – Il contenuto *little endian* delle due locazioni FBh e FCh è 0A72h;
  – A tale valore si somma il contenuto corrente del registro Y, pari a 4: risultato 0A76h;
  – Il contenuto della cella di memoria all'indirizzo 0A76h viene pertanto caricato in Accumulatore.

- **Indirizzamento pre-indicizzato:** sempre con riferimento alla figura (4.2.3), tale modalità è complementare alla precedente, fa uso del registro X ed è denotata come `OPCODE (addr8, X)`. La semantica di alto livello, in questo caso, è quella di un *array di puntatori*, tenendo comunque presenti le limitazioni (spesso drastiche) di spazio disponibile in pagina zero. Supponiamo di avere una periferica seriale che risponde con un codice di stato compreso tra 0 e 7, espresso da tre bit (isolati in modo che qui non ci interessa), in funzione del quale occorre intraprendere diverse azioni. Tale valore, moltiplicato per due (per tenere conto del fatto che un puntatore a 16 bit occupa necessariamente due byte in pagina zero) con un semplice shift a sinistra e immesso nel registro X, può essere usato in modo immediato per un dispatching diretto ad una di otto diverse subroutine usando questa modalità di indirizzamento, emulando così nel modo più compatto ed efficiente uno statement strutturato simile alla `CASE` o `SWITCH` dei linguaggi di alto livello. Risulta infatti sufficiente memorizzare in fase di inizializzazione in 16 locazioni contigue di pagina zero, a coppie, gli entry point delle otto subroutine esemplificate (realizzando così, di fatto, una jump table) e usare poi la procedura descritta (uno shift in Accumulatore, un trasferimento da A ad X e

un salto pre-indicizzato usando la tabella dei puntatori). Vale la pena di sottolineare esplicitamente che le modalità di indirizzamento indiretto, sia pur penalizzate prestazionalmente, offrono due fondamentali vantaggi:

- Consentono di referenziare un indirizzo assoluto con un effettivo *risparmio di un byte* in memoria rispetto alla forma standard, e sono per questo da sempre privilegiate ad esempio nella stesura di firmware su ROM, laddove la compattezza del codice ha quasi sempre la priorità sugli aspetti prestazionali.
- Permettono di lavorare con *puntatori ad array* (post-indicizzato Y) e *array di puntatori* (pre-indicizzato X), rispettivamente, invece di costringere ad utilizzare solo indirizzi fissi, ossia *hardcoded* (memorizzati direttamente assieme all'istruzione che li referenzia) , che siano privilegiati (pagina zero) o meno (assoluti).

4.2.5 RISC o CISC?

Le informazioni fondamentali fornite riguardo l'architettura e la ISA consentono già di classificare il design (uno dei primi sul mercato, ispirato al progenitore MC6800 che è a sua volta un apripista nel mondo delle CPU a 8 bit degli anni Settanta). La famiglia MCS 65xx Rockwell/MOS Technology si potrebbe classificare come RISC sulla base del numero di istruzioni (56) e dell'architettura generale, anche se presenta una serie di peculiarità (potremmo anche definirli «difetti di gioventù») che la rendono marcatamente diversa dai tipici RISC così come concepiti a partire dalla metà degli anni Ottanta:

- **Numero di registri ridottissimo**: un solo Accumulatore e due registri indice dedicati, tutti a 8 bit. In questo differisce anche dal progenitore MC6800 e dal coevo Z80. Nei tipici RISC è normale trovare 16, 32 o anche 64 registri *general purpose*, per non dire della memoria integrata ad accesso privilegiato (in effetti implementata, sia pure limitatamente alla pagina zero, anche in una revisione del silicio del 6510, a partire dal 1982 - si vedano i datasheet in appendice).
- **Scarsa ortogonalità del set di istruzioni**, come appena visto. Nei tipici RISC invece la quasi totalità delle istruzioni supporta tutte, o quasi tutte, le modalità di indirizzamento previste.
- **Tempi di esecuzione ampiamente variabili**, uniti a **lunghezza delle istruzioni parimenti variabile**. Nei tipici core RISC si tende invece ad offrire l'esecuzione *garantita in un singolo ciclo* di tutte le istruzioni, con la sola ovvia eccezione dei salti (dovuta principalmente allo svuotamento della coda di prefetch) ed eventualmente della moltiplicazione in hardware, che tipicamente richiedeva due cicli nei design tradizionali (da circa quindici anni i core di nuova generazione hanno azzerato anche tale gap). Allo stesso modo, le architetture Harvard di microcontroller e DSP garantiscono che qualsiasi istruzione con i suoi operandi sia contenuta in una singola parola della memoria di programma (che ovviamente non coincide con i 4 o 8 bit dell'ampiezza di bus dati, ma si attesta in media sui 12-14 bit per i core midrange e può arrivare agli 80 bit dei DSP VLIW, ossia Very Long Instruction Word). Come si vede, l'evoluzione rispetto ai primi core 8 bit è stata notevole.

4.3 ISA del 6502/6510.

Il set di istruzioni (ISA: Instruction Set Architecture) consta come anticipato di 56 istruzioni, suddivisibili funzionalmente nei seguenti 8 gruppi: **ALI (5), BI (8), DTA (10), FSCI (7), RSMI (10), SOI (6), TI (4), UJR (5)** oltre a `NOP`. Di seguito forniamo i dettagli per ciascun gruppo.

- **ALI**, Arithmetical and Logical Instructions: sono le 5 istruzioni che coinvolgono più direttamente l'ALU, l'importante sezione della CPU che presiede al calcolo aritmetico e alle operazioni logiche. Il 6502/10 implementa addizione, sottrazione, AND, OR, XOR, esclusivamente usando il registro Accumulatore A come origine.

`ADC`	Aggiunge all'Accumulatore il contenuto di una locazione di memoria e il bit di riporto (Carry).
`SBC`	Sottrae dall'Accumulatore il valore di una locazione di memoria, tenendo conto anche del bit di Carry.
`AND`	Effettua l'AND logico bitwise tra il contenuto dell'Accumulatore e una locazione di memoria.
`ORA`	Effettua l'OR logico bitwise tra il contenuto dell'Accumulatore e una locazione di memoria.
`EOR`	Effettua l'OR esclusivo (XOR) bitwise tra il contenuto dell'Accumulatore e una locazione di memoria.

- **BI**, Branch Instructions. Sono le 8 istruzioni, in coppie complementari (Branch if set, Branch if clear), che consentono di operare salti condizionati dallo stato dei seguenti flag nel registro di stato P: Carry, Zero, Negative, oVerflow.

`BCC`	Salta se Carry = 0 (riporto = 0).
`BCS`	Salta se Carry = 1 (riporto = 1).
`BNE`	Salta se il flag Z = 0 (risultato non nullo).
`BEQ`	Salta se il flag Z = 1 (risultato nullo).
`BPL`	Salta se il flag N = 0 (non negativo).
`BMI`	Salta se il flag N = 1 (negativo).
`BVC`	Salta se il flag V = 0 (nessun overflow).
`BVS`	Salta se il flag V = 1 (overflow).

- **DTA**, Data Transfer Instructions. Si tratta di 10 istruzioni dedicate, come immediatamente suggerito dalla nomenclatura, al trasferimento dei dati da registro a registro e bidirezionalmente tra registri e memoria.

`LDA`	Carica in Accumulatore il contenuto di una locazione di memoria.
`LDX`	Carica nel registro X il contenuto di una locazione di memoria.
`LDY`	Carica nel registro Y il contenuto di una locazione di memoria.
`STA`	Salva in una locazione di memoria il contenuto dell'Accumulatore.
`STX`	Salva in una locazione di memoria il contenuto del registro X.
`STY`	Salva in una locazione di memoria il contenuto del registro Y.
`TAX`	Copia il contenuto dell'Accumulatore nel registro X.
`TXA`	Carica in Accumulatore il contenuto del registro X.
`TAY`	Copia il contenuto dell'Accumulatore nel registro Y.
`TYA`	Carica in Accumulatore il contenuto del registro Y.

- **FSCI**, Flag Set and Clear Instructions. Sono le 7 istruzioni con le quali si possono manipolare alcuni flag del registro di stato P: Carry, Decimal, Interrupt, oVerflow.

`CLC`	Azzera il flag di riporto (Carry).
`SEC`	Imposta a 1 il flag di riporto (Carry).
`CLD`	Azzera il flag D (modo BCD).
`SED`	Imposta a 1 il flag D.
`CLI`	Azzera il flag di interrupt.
`SEI`	Imposta a 1 il flag di interrupt.
`CLV`	Azzera il flag di overflow V.

- **RSMI**, Register Shift and Modify Instructions. Sono le 10 istruzioni delegate ad incrementi e decrementi, scorrimenti e rotazioni come già definiti nella prima parte del testo.

`INC`	Incrementa di uno il contenuto di una locazione di memoria.
`DEC`	Decrementa di uno il contenuto di una locazione di memoria.
`INX`	Incrementa di uno il contenuto del registro X.
`DEX`	Decrementa di uno il contenuto del registro X.
`INY`	Incrementa di uno il contenuto del registro Y.
`DEY`	Decrementa di uno il contenuto del registro Y.
`ASL`	Shift aritmetico a sinistra di una posizione.
`LSR`	Shift logico a destra di una posizione.
`ROL`	Rotazione a sinistra di una posizione, con Carry.
`ROR`	Rotazione a destra di una posizione, con Carry.

- **SOI**, Stack Operation Instructions. Sono le 6 istruzioni, a coppie complementari, che gestiscono PUSH e POP dallo stack hardware gestito tramite il registro di stack S, nonché caricamento e copia del registro S nel registro X, che è *l'unico* metodo di accesso a tale registro previsto nell'intera ISA.

`PHA`	Salva l'Accumulatore sullo stack (PUSH).
`PLA`	Ripristina l'Accumulatore dallo stack (POP).
`PHP`	Salva il registro di stato P sullo stack (PUSH).
`PLP`	Ripristina il registro di stato P dallo stack (POP).
`TXS`	Carica nello stack pointer S il valore del registro X.
`TSX`	Trasferisce al registro X il valore attuale dello stack pointer S.

- **TI**, Test Instructions. Si tratta di 4 istruzioni dedicate ai confronti tra memoria e registri, che alterano appropriatamente i flag nel registro di stato P e di conseguenza consentono di modificare il flusso di esecuzione tramite successive istruzioni di salto condizionato (branch), basate appunto sullo stato di tali flag.

`CMP`	Confronta il valore in Accumulatore con il contenuto di una locazione di memoria.
`CPX`	Confronta il valore nel registro X con il contenuto di una locazione di memoria.
`CPY`	Confronta il valore nel registro Y con il contenuto di una locazione di memoria.
`BIT`	Effettua un AND volatile tra il contenuto dell'Accumulatore e una locazione di memoria.

- **UJR**, Unconditional Jumps and Returns. Sono le 5 istruzioni non condizionate di controllo del flusso: salto, chiamata a subroutine, return, break, return from interrupt.

`JMP`	Effettua un salto alla locazione specificata e continua l'esecuzione da lì.
`JSR`	Richiama una subroutine.
`RTS`	Termina la subroutine e ritorna al chiamante.
`BRK`	Interrupt software o trap interrupt, salta ad un vettore predeterminato.
`RTI`	Termina un interrupt e torna al chiamante.

- Per completare il set, la singola istruzione `NOP` (presente universalmente nei set di istruzioni) che *non effettua alcun cambiamento di stato* eccetto l'incremento di una unità del registro Program Counter, e la cui esecuzione causa implicitamente (come ogni altra istruzione) un **ritardo temporale**, in questo caso pari a 2 cicli di clock.

Nel seguito analizzeremo in dettaglio ciascuna singola istruzione e i suoi effetti sui flag del processore. Saranno indicate la lunghezza complessiva di istruzione e operandi (uno, due o tre bytes) e i tempi di esecuzione, la codifica binaria ed esadecimale. Laddove il tempo sia condizionato dalla collocazione finale dell'operando (stessa pagina o pagine diverse) saranno indicati i tempi massimo e minimo, nella forma 4/5 dove il primo valore si applica quando non si verifica un cambio di pagina, e il secondo nel caso opposto. Nel seguito tale variabilità, in forma sintetica, sarà anche indicata con il tempo minimo seguito da un asterisco, come ad esempio in 5*.

Nell'indicare la *famiglia di opcode* alla quale appartiene ogni mnemonico, al fine di evidenziare i bitfield scelti dai progettisti per la codifica, faremo uso di una maschera di bit (ad esempio 011...01) che indica la *parte fissa* della codifica, e al posto delle locazioni mancanti inseriremo dei codici alfabetici nei quali a ciascuna lettera minuscola corrisponde un singolo bit, come indicato nelle tabelle che seguono. Al di là delle 24 istruzioni ad indirizzamento implicito (più le 8 istruzioni di salto condizionato relativo), le rimanenti istruzioni del set si possono suddividere in cinque gruppi principali riguardo le modalità di indirizzamento supportate. Tali gruppi supportano rispettivamente 8, 4, 5, 3 e 5 diverse modalità di indirizzamento. In generale, la ISA della famiglia 65xx non presenta una elevata *ortogonalità*: nei set di istruzioni altamente ortogonali, per definizione, la maggioranza delle istruzioni supporta invece tutti o quasi tutti i metodi di indirizzamento. Le tabelle seguenti mostrano la codifica dei bitfield adottata nelle successive sottosezioni per individuare l'effettivo opcode binario corrispondente alla specifica coppia istruzione+indirizzamento.

aaa		
000	Pre-indicizzato indiretto	(addr8, X)
001	Pagina zero	addr8
010	Immediato	#byte
011	Assoluto	addr16
100	Post-indicizzato indiretto	(addr8),Y
101	Indicizzato pagina zero, X	addr8,X
110	Assoluto indicizzato X	addr16,X
111	Assoluto indicizzato Y	addr16,Y

Dunque, ad esempio, la codifica binaria 011*aaa*01 (corrispondente allo mnemonico ADC) darà adito ai seguenti effettivi opcode generati dall'Assembly (o codificati manualmente) secondo il tipo di indirizzamento utilizzato:

Linea sorgente	Opcode		Indirizzamento	Bytes
	Binario	Hex		
`ADC (addr8,X)`	011000**01**	61	Pre-indicizzato indiretto	2
`ADC addr8`	011001**01**	65	Pagina zero	2
`ADC #byte`	011010**01**	69	Immediato	2
`ADC addr16`	011011**01**	6D	Assoluto	3
`ADC (addr8),Y`	011100**01**	71	Post-indicizzato indiretto	2
`ADC addr8,X`	011101**01**	75	Indicizzato pagina zero, X	2
`ADC addr16,X`	011110**01**	79	Assoluto indicizzato X	3
`ADC addr16,Y`	011111**01**	7D	Assoluto indicizzato Y	3

Seguono le altre quattro codifiche, con identico significato. Si noti che i codici binari non sono necessariamente sequenziali.

bb		
00	Pagina zero	addr8
01	Assoluto	addr16
10	Indicizzato pagina zero, X	addr8,X
11	Assoluto indicizzato X	addr16,X

ccc		
001	Pagina zero	addr8
010	Accumulatore	A
011	Assoluto	addr16
101[3]	Indicizzato pagina zero, X	addr8,X
111[4]	Assoluto indicizzato X	addr16,X

dd		
00	Immediato	#byte
01	Pagina zero	addr8
11	Assoluto	addr16

eee		
001	Immediato	#byte
010	Pagina zero	addr8
011	Assoluto	addr16
101[5]	Indicizzato pagina zero	addr8,X/Y
111[6]	Assoluto indicizzato	addr16,X/Y

[3] Attenzione: il medesimo codice indica invece l'indirizzamento indirizzato con registro Y per la sola istruzione STY.
[4] Vedi nota precedente.
[5] Attenzione: il medesimo codice indica addr8,Y per LDX e addr8,X per LDY.
[6] Vedi nota precedente.

Alcuni rari opcode ammettono variazioni di un singolo bit per i due diversi modi di indirizzamento supportati: tali variazioni saranno indicate in modo esplicito, elencando i diversi codici. Si noti infine che gli opcode vengono elencati in ordine numerico crescente di codifica, per fissare meglio nella memoria del lettore i criteri di formazione e codifica della ISA scelti dai progettisti: sia pure molto in anticipo rispetto alla nascita di veri e propri criteri formali per il design sistematico dei set di istruzioni e delle *best practices* del codesign[7], le decisioni adottate per la famiglia 65xx hanno comunque una loro razionalità ed omogeneità, nonostante alcune eccezioni e singolarità. Nei casi che supportano numerose modalità di indirizzamento (4 o più) si è evidenziata graficamente la modalità che consente le migliori prestazioni, riportando in rosso il numero di cicli ad essa relativo.

4.3.1 ADC

Lo mnemonico deriva da **AD**d with **C**arry. Aggiunge il contenuto di una locazione di memoria all'Accumulatore, sommando poi il bit di riporto al totale ivi contenuto: $A \leftarrow M + A + C$. Supporta 8 modalità di indirizzamento. Codifica binaria: 011*aaa*01.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*	*					*	*

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`ADC (addr8,X)`	01100001	61	Pre-indicizzato indiretto	2	6
`ADC addr8`	01100101	65	Pagina zero	2	3
`ADC #byte`	01101001	69	Immediato	2	2
`ADC addr16`	01101101	6D	Assoluto	3	4
`ADC (addr8),Y`	01110001	71	Post-indicizzato indiretto	2	5/6
`ADC addr8,X`	01110101	75	Indicizzato pagina zero, X	2	4
`ADC addr16,Y`	01111001	79	Assoluto indicizzato Y	3	4/5
`ADC addr16,X`	01111101	7D	Assoluto indicizzato X	3	4/5

4.3.2 AND

Effettua un **AND** logico bitwise tra memoria e Accumulatore, ponendo il risultato in quest'ultimo: $A \leftarrow A \wedge M$. Supporta un totale di 8 modalità di indirizzamento. Codifica binaria: 001*aaa*01.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	

A	*B*	*A* AND *B*
0	0	0
0	1	0
1	0	0
1	1	1

[7] Contrazione di «contemporaneous design», la prassi ingegneristica del design contemporaneo e *interdipendente* dei vary layer di un progetto: in particolare per quanto riguarda le CPU ci si riferisce ad hardware, microcodice e ISA portati avanti rigorosamente in parallelo. Tale modo di procedere evita note criticità ed errori, ormai ben studiate in progetti di importanza storica, nei quali miraggi di risparmio economico nell'immediato, decisioni di progetto poco meditate o false «scorciatoie» in fase di design hardware hanno poi comportato caoticità, kludge dell'ultimo minuto, eccezioni e irrazionalità negli altri layer ed aree di progetto, generando così penalità prestazionali, costi superflui a livello applicativo e gravi problemi di retrocompatibilità.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`AND (addr8,X)`	00100001	21	Pre-indicizzato indiretto	2	6
`AND addr8`	00100101	25	Pagina zero	2	3
`AND #byte`	00101001	29	Immediato	2	2
`AND addr16`	00101101	2D	Assoluto	3	4
`AND(addr8),Y`	00110001	31	Post-indicizzato indiretto	2	5/6
`AND addr8,X`	00110101	35	Indicizzato pagina zero, X	2	4
`AND addr16,Y`	00111001	39	Assoluto indicizzato Y	3	4/5
`AND addr16,X`	00111101	3D	Assoluto indicizzato X	3	4/5

4.3.3 ASL

Lo mnemonico richiama l'operazione: **A**rithmetic **S**hift to **L**eft. Effettua uno *scorrimento a sinistra* dell'Accumulatore o di una locazione di memoria di una posizione, il che equivale ad una moltiplicazione per due in caso di valori non segnati. Si veda in figura (4.3.1) e la parte introduttiva del testo (3.3.1 a pagina 24). Si noti che, a dispetto della nomenclatura scelta, *non si tratta* di un vero shift aritmetico in quanto il bit del segno *non viene preservato* nella locazione 7. Il bit 7 (MSB) viene invece spostato nel flag di Carry, mentre nel LSB viene inserito un valore 0. L'istruzione supporta 5 modalità di indirizzamento. Codifica binaria: 000*ccc*10.

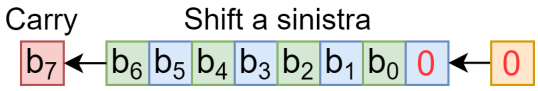


Figura 4.3.1: Effetto dell'istruzione ASL.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	*

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`ASL addr8`	00000110	06	Pagina zero	2	5
`ASL A`	00001010	0A	Accumulatore	1	2
`ASL addr16`	00001110	0E	Assoluto	3	6
`ASL addr8,X`	00010110	16	Indicizzato pagina zero, X	2	6
`ASL addr16,X`	00011110	1E	Assoluto indicizzato X	3	7

4.3.4 BCC

Branch if **C**arry **C**lear: effettua un salto relativo (massimo +127 o -128 locazioni dall'attuale Program Counter PC) se il flag di Carry è nullo, C=0. Da utilizzare dopo un test o altra operazione che alteri il flag di riporto. Impiega due cicli se il salto non viene intrapreso, tre in caso contrario (che salgono a quattro se la destinazione si trova in una pagina di memoria diversa rispetto al valore attuale di PC). Codifica binaria: 10010000. Flags modificati: nessuno.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`BCC label`	10010000	90	Relativo	2	2/4

4.3.5 BCS

Branch if **C**arry **S**et, complementare all'operazione precedente. Effettua un salto relativo (massimo +127 o -128 locazioni dall'attuale Program Counter PC) se il flag di Carry è alto, C=1. Impiega due cicli se il salto non viene intrapreso, tre in caso contrario (che salgono a quattro se la destinazione si trova in una pagina di memoria diversa rispetto al valore attuale di PC). Codifica binaria: 10110000. Flags modificati: nessuno.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`BCS label`	10110000	B0	Relativo	2	2/4

4.3.6 BEQ

Branch if **EQ**ual (to zero). Effettua un salto relativo (massimo +127 o -128 locazioni dall'attuale Program Counter PC) se il flag di Zero è alto, Z=1, per indicare un risultato nullo della precedente operazione. Come le altre istruzioni di branching, impiega due cicli se il salto non viene intrapreso, tre in caso contrario (che salgono a quattro se la destinazione si trova in una pagina di memoria diversa rispetto al valore attuale di PC). Codifica binaria: 11110000. Flags modificati: nessuno.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`BEQ label`	11110000	F0	Relativo	2	2/4

4.3.7 BIT

Lo mnemonico deriva da **BI**t **T**est. Si tratta di una istruzione con alcuni aspetti anomali, ma in linea generale non dissimile da altre istruzioni di test analoghe presenti nelle ISA di vari processori. Effettua un AND logico dei contenuti dell'Accumulatore con quelli di una locazione di memoria e imposta i flag nel registro P di conseguenza, ma *non aggiorna il contenuto dell'Accumulatore col risultato.* Si tratta quindi di una vera e propria istruzione di test, che imposta regolarmente il flag di zero se il risultato è nullo e viene normalmente utilizzata prima delle istruzioni di branch (in particolare BNE e BEQ), ma con un effetto collaterale piuttosto inusuale sul registro dei flag: infatti, in questo caso, i bit 7 e 6 della locazione di memoria indicata sono *copiati rispettivamente nei flag N e V del registro di stato.* Codifica binaria: 0010?100.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
7	6					*	

A	*B*	*A* AND *B*
0	0	0
0	1	0
1	0	0
1	1	1

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`BIT addr8`	00100100	24	Pagina zero	2	3
`BIT addr16`	00101100	2C	Assoluto	3	4

4.3.8 BMI

Branch if **MI**nus. Effettua un salto relativo (massimo +127 o -128 locazioni dall'attuale Program Counter PC) se il flag del segno N è alto, N=1, per indicare un risultato negativo della precedente operazione (semantica valida solo se si manipolano numeri in complemento a due, altrimenti trattasi semplicemente del MSB del registro interessato). Come tutte le altre istruzioni di branching, impiega due cicli se il salto non viene intrapreso, tre in caso contrario (che salgono a quattro se la destinazione si trova in una pagina di memoria diversa rispetto al valore attuale di PC). Codifica binaria: 00110000. Flags modificati: nessuno.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`BMI label`	00110000	30	Relativo	2	2/4

4.3.9 BNE

Branch if **N**ot **E**qual (to zero). Complementare a BEQ, effettua un salto relativo (massimo +127 o -128 locazioni dall'attuale Program Counter PC) se il flag di Zero è nullo, Z=0, per indicare un risultato non nullo della precedente operazione. Come ogni altra istruzioni di branching, impiega due cicli se il salto non viene intrapreso, tre in caso contrario (che salgono a quattro se la destinazione si trova in una pagina di memoria diversa rispetto al valore attuale di PC). Codifica binaria: 11010000. Flags modificati: nessuno.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`BNE label`	11010000	D0	Relativo	2	2/4

4.3.10 BPL

Branch if **PL**us. Complementare a BMI, effettua un salto relativo (massimo +127 o -128 locazioni dall'attuale Program Counter PC) se il flag del segno N è nullo, N=0, per indicare un risultato non negativo della precedente operazione (semantica valida solo se si manipolano numeri in complemento a due: altrimenti trattasi semplicemente del MSB del registro interessato). Come tutte le altre istruzioni di branching, impiega due cicli se il salto non viene intrapreso, tre in caso contrario (che salgono a quattro se la destinazione si trova in una pagina di memoria diversa rispetto al valore attuale di PC). Codifica binaria: 00010000. Flags modificati: nessuno.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`BPL label`	00010000	10	Relativo	2	2/4

4.3.11 BRK

BRea**K** invoca un interrupt software o trap interrupt. La sequenza delle azioni svolte si può semplificare come segue:

1. Il registro Program Counter è incrementato di due;
2. Il flag Break è posto a 1;
3. Il Program Counter e il registro di stato P sono salvati sullo stack (PUSH) nel seguente ordine:

Cima dello stack	Nibble alto PC
-1	Nibble basso PC
-2	Registro di stato P
$SP \rightarrow$	

4. Il flag di Interrupt è posto a 1 per disabilitare gli interrupt esterni;
5. L'indirizzo contenuto nelle locazioni del vettore di BREAK (FFFEh e FFFFh) viene caricato nel Program Counter;
6. Il flusso di esecuzione continua da tale locazione, la prima dell'interrupt software/trap interrupt.

Tale istruzione è estremamente flessibile e consente l'uso di breakpoint a scopo di debugging, oppure il trasferimento del controllo ad un monitor o DOS wegde. Codifica binaria: 00000000[8].

Flags modificati:

N	V	$-$	B	D	I	Z	C
			1		1		

[8] La scelta dell'opcode nullo è stata a suo tempo dettata da ragioni tecnologiche: tale codice è l'unico che può essere sicuramente riprogrammato nelle celle di una memoria PROM (di sola scrittura, non cancellabili), le quali constavano di una serie di veri e propri fusibili che venivano interrotti durante la programmazione, in modo tale che la PROM vergine riportava valori FFh in ciascuna cella (analogamente alle EPROM, che tuttavia sono cancellabili e riscrivibili per un numero molto elevato di cicli). In questo modo era possibile anche a posteriori, sul campo, modificare un firmware inserendo uno zero al posto di un opcode (punto di iniezione) per poi completare la routine altrove, revettorizzandola sul BREAK interrupt. Questa è solo una delle innumerevoli forme di flessibilità operativa consentite da tale istruzione.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`BRK`	00000000	0	Implicito	1	7

4.3.12 BVC

Branch if o**V**erflow **C**lear. Effettua un salto relativo (massimo +127 o -128 locazioni dall'attuale Program Counter PC) se il flag di overflow V è basso, V=0, per indicare l'assenza di overflow dalla precedente operazione aritmetica. Come tutte le altre istruzioni di branching, impiega due cicli se il salto non viene intrapreso, tre in caso contrario (che salgono a quattro se la destinazione si trova in una pagina di memoria diversa rispetto al valore attuale di PC). Codifica binaria: 01010000. Flags modificati: nessuno.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`BVC label`	01010000	50	Relativo	2	2/4

4.3.13 BVS

Branch if o**V**erflow **S**et. Effettua un salto relativo (massimo +127 o -128 locazioni dall'attuale Program Counter PC) se il flag di overflow V è alto, V=1, per indicare un overflow della precedente operazione. Come tutte le altre istruzioni di branching, impiega due cicli se il salto non viene intrapreso, tre in caso contrario (che salgono a quattro se la destinazione si trova in una pagina di memoria diversa rispetto al valore attuale di PC). Codifica binaria: 01110000. Flags modificati: nessuno.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`BVS label`	01110000	70	Relativo	2	2/4

4.3.14 CLC

CLear **C**arry. Azzera il flag di riporto, C=0. Codifica binaria: 00011000.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
							0

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`CLC`	00011000	18	Implicito	1	2

4.3.15 CLD

CLear **D**ecimal. Azzera il flag del modo decimale (BCD), D=0. Codifica binaria: 11011000.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
				0			

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`CLD`	11011000	D8	Implicito	1	2

4.3.16 CLI

CLear **I**nterrupt. Azzera il flag di interrupt, I=0. Codifica binaria: 01011000.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
					0		

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`CLI`	01011000	58	Implicito	1	2

4.3.17 CLV

CLear o**V**erflow. Azzera il flag di overflow, V=0. Codifica binaria: 10111000.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
	0						

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`CLV`	10111000	B8	Implicito	1	2

4.3.18 CMP

Tipica istruzione di confronto senza memorizzazione del risultato, **CoMP**are esegue una sottrazione implicita tra il contenuto dell'Accumulatore e una locazione di memoria, senza però salvare il risultato e modificando solamente i flag del registro di stato P, come preparazione per una istruzione di salto condizionato (branch). Supporta un totale di 8 modalità di indirizzamento. Codifica binaria: 110*aaa*01.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	*

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`CMP (addr8,X)`	11000001	C1	Pre-indicizzato indiretto	2	6
`CMP addr8`	11000101	C5	Pagina zero	2	3
`CMP #byte`	11001001	C9	Immediato	2	2
`CMP addr16`	11001101	CD	Assoluto	3	4
`CMP (addr8),Y`	11010001	D1	Post-indicizzato indiretto	2	5/6
`CMP addr8,X`	11010101	D5	Indicizzato pagina zero, X	2	4
`CMP addr16,Y`	11011001	D9	Assoluto indicizzato Y	3	4/5
`CMP addr16,X`	11011101	DD	Assoluto indicizzato X	3	4/5

4.3.19 CPX

Analoga a CMP, l'istruzione **C**om**P**are **X** esegue una sottrazione implicita tra il contenuto del registro X e una locazione di memoria, modificando solamente i flag del registro di stato P, come preparazione per una istruzione di salto condizionato (branch). Supporta un totale di 3 modalità di indirizzamento. Codifica binaria: 1110*dd*00.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	*

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`CPX #byte`	11100000	E0	Immediato	2	2
`CPX addr8`	11100100	E4	Pagina zero	2	3
`CPX addr16`	11101100[9]	EC	Assoluto	3	4

4.3.20 CPY

Analoga a CPX, l'istruzione **C**om**P**are **Y** esegue una sottrazione implicita tra il contenuto del registro Y e una locazione di memoria, modificando solamente i flag del registro di stato P, come preparazione per una istruzione di salto condizionato (branch). Supporta un totale di 3 modalità di indirizzamento. Codifica binaria: 1100*dd*00.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	*

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`CPY #byte`	11000000	C0	Immediato	2	2
`CPY addr8`	11000100	C4	Pagina zero	2	3
`CPY addr16`	11001100[10]	CC	Assoluto	3	4

4.3.21 DEC

Lo mnemonico corrisponde a **DEC**rement. Diminuisce di una unità il contenuto di una locazione di memoria: se al momento dell'esecuzione la locazione contiene uno zero, si ha un wraparound e il contenuto passa a FFh, impostando opportunamente i flag nel registro di stato P. Curiosamente, non può operare sull'Accumulatore, né esiste una diversa istruzione dedicata per farlo. Questa istruzione supporta infatti solo 4 modalità di indirizzamento: due assolute e due indicizzate. Codifica binaria: 110*bb*110.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`DEC addr8`	11000110	C6	Pagina zero	2	5
`DEC addr16`	11001110	CE	Assoluto	3	6
`DEC addr8,X`	11010110	D6	Indicizzato pagina zero, X	2	6
`DEC addr16,X`	11011110	DE	Assoluto indicizzato X	3	7

4.3.22 DEX

DEcrement **X**. Decrementa di una unità il contenuto del registro X, con wraparound a FFh in caso di contenuto nullo al momento dell'esecuzione. Codifica binaria: 11001010.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`DEX`	11001010	CA	Implicito	1	2

[9] Si noti che la codifica dd=10 non viene qui utilizzata ed è impiegata per un diverso opcode.
[10] Analogamente a CPX, si noti che anche qui la codifica dd=10 non viene utilizzata ed è impiegata per un diverso opcode.

4.3.23 DEY

DEcrement **Y**. Decrementa di una unità il contenuto del registro Y, con wraparound a FFh in caso di contenuto nullo al momento dell'esecuzione. Codifica binaria: 10001000.

Flags modificati:

N	V	–	B	D	I	Z	C
*						*	

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`DEY`	10001000	88	Implicito	1	2

4.3.24 EOR

Exclusive **OR** esegue uno XOR logico bitwise tra il contenuto dell'Accumulatore e una locazione di memoria: il risultato viene salvato nell'Accumulatore. $A \leftarrow A \oplus M$. Supporta un totale di 8 modalità di indirizzamento. Codifica binaria: 010*aaa*01.

Flags modificati:

N	V	–	B	D	I	Z	C
*						*	

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`EOR (addr8,X)`	01000001	41	Pre-indicizzato indiretto	2	6
`EOR addr8`	01000101	45	Pagina zero	2	3
`EOR #byte`	01001001	49	Immediato	2	2
`EOR addr16`	01001101	4D	Assoluto	3	4
`EOR (addr8),Y`	01010001	51	Post-indicizzato indiretto	2	5/6
`EOR addr8,X`	01010101	55	Indicizzato pagina zero, X	2	4
`EOR addr16,Y`	01011001	59	Assoluto indicizzato Y	3	4/5
`EOR addr16,X`	01011101	5D	Assoluto indicizzato X	3	4/5

4.3.25 INC

Lo mnemonico corrisponde a **INC**rement. Aumenta di una unità il contenuto di una locazione di memoria: se al momento dell'esecuzione la locazione contiene il valore limite FFh, si ha un wraparound e il contenuto passa a 0, impostando opportunamente i flag nel registro di stato P. Curiosamente, non può operare sull'Accumulatore, né esiste una diversa istruzione dedicata per farlo. Questa istruzione supporta infatti solo 4 modalità di indirizzamento: due assolute e due indicizzate. Codifica binaria: 111*bb*110.

Flags modificati:

N	V	–	B	D	I	Z	C
*						*	

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`INC addr8`	11100110	E6	Pagina zero	2	5
`INC addr16`	11101110	EE	Assoluto	3	6
`INC addr8,X`	11110110	F6	Indicizzato pagina zero, X	2	6
`INC addr16,X`	11111110	FE	Assoluto indicizzato X	3	7

4.3.26 INX

INcrement **X**. Incrementa di una unità il contenuto del registro X, con wraparound a 0 in caso di contenuto pari a FFh al momento dell'esecuzione. Codifica binaria: 11101000.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`INX`	11101000	E8	Implicito	1	2

4.3.27 INY

INcrement **Y**. Incrementa di una unità il contenuto del registro Y, con wraparound a 0 in caso di contenuto pari a FFh al momento dell'esecuzione. Codifica binaria: 11001000.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`INY`	11001000	C8	Implicito	1	2

4.3.28 JMP

Esegue un salto incondizionato (**J**u**MP**) che altera il flusso di esecuzione. Può operare con un indirizzo assoluto o con un puntatore (16 bit) ad un indirizzo. Codifica binaria: 01?01100. Flags modificati: nessuno.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`JMP addr16`	01001100	4C	Assoluto	3	3
`JMP (addr16)`	01101100	6C	Indiretto	3	5

4.3.29 JSR

Richiama una subroutine (**J**ump to **S**ub**R**outine) ad un dato indirizzo assoluto. Salva sullo stack il valore corrente di Program Counter, aumentato di due unità per puntare all'istruzione successiva. In questo modo, al termine della subroutine, l'istruzione RTS riporterà il flusso di esecuzione al punto in cui è stato interrotto. Codifica binaria: 00100000. Flags modificati: nessuno.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`JSR addr16`	00100000	20	Assoluto	3	6

4.3.30 LDA

LoaD Accumulator carica un valore (byte) in Accumulatore da una locazione di memoria: $A \leftarrow M$. Supporta un totale di 8 modalità di indirizzamento. Codifica binaria: 101*aaa*01.

Flags modificati:

N	V	$-$	B	D	I	Z	C
*						*	

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`LDA (addr8,X)`	10100001	A1	Pre-indicizzato indiretto	2	6
`LDA addr8`	10100101	A5	Pagina zero	2	3
`LDA #byte`	10101001	A9	Immediato	2	2
`LDA addr16`	10101101	AD	Assoluto	3	4
`LDA (addr8),Y`	10110001	B1	Post-indicizzato indiretto	2	5/6
`LDA addr8,X`	10110101	B5	Indicizzato pagina zero, X	2	4
`LDA addr16,Y`	10111001	B9	Assoluto indicizzato Y	3	4/5
`LDA addr16,X`	10111101	BD	Assoluto indicizzato X	3	4/5

4.3.31 LDX

LoaD X carica un valore (byte) nel registro X da una locazione di memoria. Supporta un totale di 5 modalità di indirizzamento: si noti come le modalità indicizzate fanno necessariamente uso del registro Y. Codifica binaria: 101*eee*10.

Flags modificati:

N	V	$-$	B	D	I	Z	C
*						*	

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`LDX #byte`	10100010	A2	Immediato	2	2
`LDX addr8`	10100110	A6	Pagina zero	2	3
`LDX addr16`	10101110	AE	Assoluto	3	4
`LDX addr8,Y`	10110110	B6	Indicizzato pagina zero, Y	2	4
`LDX addr16,Y`	10111110	BE	Assoluto indicizzato Y	3	4/5

4.3.32 LDY

LoaD Y carica un valore (byte) nel registro Y da una locazione di memoria. Supporta un totale di 5 modalità di indirizzamento: si noti come le modalità indicizzate fanno necessariamente uso del registro X. Codifica binaria: 101*eee*00.

Flags modificati:

N	V	$-$	B	D	I	Z	C
*						*	

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`LDY #byte`	10100000	A0	Immediato	2	2
`LDY addr8`	10100100	A4	Pagina zero	2	3
`LDY addr16`	10101100	AC	Assoluto	3	4
`LDY addr8,X`	10110100	B4	Indicizzato pagina zero, X	2	4
`LDY addr16,X`	10111100	BC	Assoluto indicizzato X	3	4/5

4.3.33 LSR

Logical **S**hift to **R**ight. Effettua uno *scorrimento a destra* dell'Accumulatore o di una locazione di memoria di una posizione, il che equivale ad una *divisione per due* in caso di valori non segnati. Si veda in figura (4.3.2) e la parte introduttiva del testo (3.3.1 a pagina 24). Nel bit 7 (MSB) viene inserito un valore nullo, mentre il LSB viene spostato nel flag di Carry. Si noti che questa istruzione azzera **sempre** il flag di segno N. L'istruzione supporta 5 modalità di indirizzamento. Codifica binaria: 010*ccc*10.

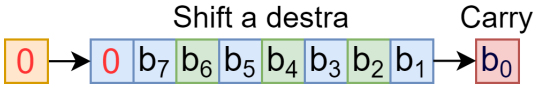


Figura 4.3.2: Effetto dell'istruzione LSR.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
0						*	*

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`LSR addr8`	01000110	46	Pagina zero	2	5
`LSR A`	01001010	4A	Accumulatore	1	2
`LSR addr16`	01001110	4E	Assoluto	3	6
`LSR addr8,X`	01010110	56	Indicizzato pagina zero, X	2	6
`LSR addr16,X`	01011110	5E	Assoluto indicizzato X	3	7

4.3.34 NOP

No OPeration. Non altera lo stato del processore, eccetto per l'incremento di una unità del registro interno PC Program Counter. La sua esecuzione richiede comunque due cicli di clock, quindi 2 µs con un tipico quarzo da 1,000000 MHz. Codifica binaria: 11101010. Flags modificati: nessuno.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`NOP`	11101010	EA	Implicito	1	2

4.3.35 ORA

OR Accumulator esegue un OR logico bitwise tra il contenuto dell'Accumulatore e una locazione di memoria: il risultato viene salvato nell'Accumulatore. $A \leftarrow A \vee M$. Supporta un totale di 8 modalità di indirizzamento. Codifica binaria: 000*aaa*01.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	

A	*B*	*A* OR *B*
0	0	0
0	1	1
1	0	1
1	1	1

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`ORA (addr8,X)`	00000001	01	Pre-indicizzato indiretto	2	6
`ORA addr8`	00000101	05	Pagina zero	2	3
`ORA #byte`	00001001	09	Immediato	2	2
`ORA addr16`	00001101	0D	Assoluto	3	4
`ORA (addr8),Y`	00010001	11	Post-indicizzato indiretto	2	5/6
`ORA addr8,X`	00010101	15	Indicizzato pagina zero, X	2	4
`ORA addr16,Y`	00011001	19	Assoluto indicizzato Y	3	4/5
`ORA addr16,X`	00011101	1D	Assoluto indicizzato X	3	4/5

4.3.36 PHA

PusH Accumulator. Salva l'Accumulatore sullo stack, decrementando poi di una unità il registro S, Stack Pointer. Codifica binaria: 01001000. Flags modificati: nessuno.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`PHA`	01001000	48	Implicito	1	3

4.3.37 PHP

PusH Processor Status Register. Salva il registro di stato P sullo stack, decrementando poi di una unità il registro S, Stack Pointer. Normalmente utilizzata per salvare lo stato dei flag prima di richiamare una subroutine (codice utente o firmware di sistema), il suo utilizzo è implicito in tutti i meccanismi di chiamata a interrupt, sia in risposta ai segnali esterni $\overline{IRQ}$ e $\overline{NMI}$ che in caso di interrupt software ossia trap interrupt invocato tramite l'istruzione BRK: in ambedue i casi, il registro di stato viene salvato e ripristinato automaticamente. Codifica binaria: 00001000. Flags modificati: nessuno.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`PHP`	00001000	08	Implicito	1	3

4.3.38 PLA

PulL Accumulator. Ripristina (pop) il contenuto dell'Accumulatore dalla cima dello stack, dopo avere incrementato di una unità il contenuto del registro S Stack Pointer. Si noti che, analogamente al normale caricamento di un valore in Accumulatore, anche qui i flags di segno N e zero Z vengono modificati secondo i contenuti finali del MSB e dell'intero registro A rispettivamente. Codifica binaria: 01101000.

Flags modificati:

N	V	$-$	B	D	I	Z	C
*						*	

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`PLA`	01101000	68	Implicito	1	4

4.3.39 PLP

PulL Processor Status Register. Ripristina (pop) il contenuto del registro P dalla cima dello stack, dopo avere incrementato di una unità il contenuto del registro S Stack Pointer. Codifica binaria: 00101000.

Flags modificati:

N	V	$-$	B	D	I	Z	C
*	*		*	*	*	*	*

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`PLP`	00101000	28	Implicito	1	4

4.3.40 ROL

ROtate **L**eft. Effettua una rotazione a sinistra dell'Accumulatore o di una locazione di memoria di una posizione, attraverso il Carry. Si veda in figura (4.3.3) e la parte introduttiva del testo (3.3.1 a pagina 24). Nel bit 0 (LSB) viene inserito il valore attuale del Carry, mentre il MSB (bit 7) viene poi a sua volta spostato nel flag di Carry. L'istruzione supporta 5 modalità di indirizzamento. Codifica binaria: 001*ccc*10.

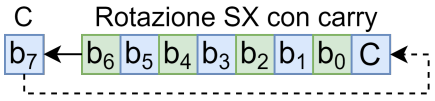


Figura 4.3.3: Effetto dell'istruzione ROL.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	*

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`ROL addr8`	00100110	26	Pagina zero	2	5
`ROL A`	00101010	2A	Accumulatore	1	2
`ROL addr16`	00101110	2E	Assoluto	3	6
`ROL addr8,X`	00110110	36	Indicizzato pagina zero, X	2	6
`ROL addr16,X`	00111110	3E	Assoluto indicizzato X	3	7

4.3.41 ROR

ROtate **R**ight. Effettua una rotazione a destra dell'Accumulatore o di una locazione di memoria di una posizione, attraverso il Carry. Si veda in figura (4.3.4) e la parte introduttiva del testo (3.3.1 a pagina 24). Nel bit 7 (MSB) viene inserito il valore attuale del Carry, mentre il LSB (bit 0) viene poi a sua volta spostato nel flag di Carry. L'istruzione supporta 5 modalità di indirizzamento. Codifica binaria: 011*ccc*10.

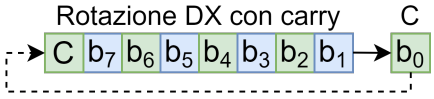


Figura 4.3.4: Effetto dell'istruzione ROR.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	*

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`ROR addr8`	01100110	66	Pagina zero	2	5
`ROR A`	01101010	6A	Accumulatore	1	2
`ROR addr16`	01101110	6E	Assoluto	3	6
`ROR addr8,X`	01110110	76	Indicizzato pagina zero, X	2	6
`ROR addr16,X`	01111110	7E	Assoluto indicizzato X	3	7

4.3.42 RTI

Re**T**urn from **I**nterrupt. Torna al flusso principale di esecuzione dopo un interrupt. Ripristina il registro di stato P e il registro PC Program Counter dalla cima dello stack. Codifica binaria: 01000000.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*	*		*	*	*	*	*

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`RTI`	01000000	40	Implicito	1	6

4.3.43 RTS

Re**T**urn from **S**ubroutine. Torna al flusso principale di esecuzione dopo una chiamata a subroutine, avviata con l'istruzione JSR. Ripristina il registro PC Program Counter dalla cima dello stack. Codifica binaria: 01100000. Flags modificati: nessuno.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`RTS`	01100000	60	Implicito	1	6

4.3.44 SBC

Lo mnemonico deriva da **S**u**B**tract with **C**arry. Implementa la sottrazione con prestito, usando opportunamente il bit di Carry. Sottrae dall'Accumulatore il contenuto di una locazione di memoria e il complemento del Carry, ponendo il risultato in Accumulatore. $A \leftarrow A - M - \overline{C}$. Supporta 8 modalità di indirizzamento. Codifica binaria: 111*aaa*01.

Flags modificati:

N	V	$-$	B	D	I	Z	C
*	*					*	*

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`SBC (addr8,X)`	11100001	E1	Pre-indicizzato indiretto	2	6
`SBC addr8`	11100101	E5	Pagina zero	2	3
`SBC #byte`	11101001	E9	Immediato	2	2
`SBC addr16`	11101101	ED	Assoluto	3	4
`SBC (addr8),Y`	11110001	F1	Post-indicizzato indiretto	2	5/6
`SBC addr8,X`	11110101	F5	Indicizzato pagina zero, X	2	4
`SBC addr16,Y`	11111001	F9	Assoluto indicizzato Y	3	4/5
`SBC addr16,X`	11111101	FD	Assoluto indicizzato X	3	4/5

4.3.45 SEC

SEt **C**arry. Imposta il flag di riporto, C=1. Codifica binaria: 00111000.

Flags modificati:

N	V	$-$	B	D	I	Z	C
							1

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`SEC`	00111000	38	Implicito	1	2

4.3.46 SED

SEt **D**ecimal. Imposta il flag del modo decimale (BCD), D=1. Codifica binaria: 11111000.

Flags modificati:

N	V	$-$	B	D	I	Z	C
				1			

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`SED`	11111000	F8	Implicito	1	2

4.3.47 SEI

SEt **I**nterrupt. Imposta il flag di interrupt, I=1. Codifica binaria: 01111000.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
					1		

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`SEI`	01111000	78	Implicito	1	2

4.3.48 STA

STore **A**ccumulator copia il contenuto dell'Accumulatore in una locazione di memoria. Si noti che, come nella maggioranza delle CPU a 8 bit, questa istruzione non modifica lo stato dei flag. Pur appartenendo al gruppo di istruzioni con bitfield di indirizzamento *aaa*, supporta solo 7 modalità di indirizzamento, perché (per ovvi motivi) la destinazione non può essere un operando immediato. Codifica binaria: 100*aaa*01. Flags modificati: nessuno.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`STA (addr8,X)`	10000001	81	Pre-indicizzato indiretto	2	6
`STA addr8`	10000101	85	Pagina zero	2	3
	10001001	89	Non utilizzato		
`STA addr16`	10001101	8D	Assoluto	3	4
`STA (addr8),Y`	10010001	91	Post-indicizzato indiretto	2	6
`STA addr8,X`	10010101	95	Indicizzato pagina zero, X	2	4
`STA addr16,Y`	10011001	99	Assoluto indicizzato Y	3	5
`STA addr16,X`	10011101	9D	Assoluto indicizzato X	3	4/5

4.3.49 STX

Lo mnemonico corrisponde a **ST**ore **X**. Salva il contenuto del registro X in una locazione di memoria. Si noti che, tra le modalità di indirizzamento supportate, mancano (per ragioni piuttosto ovvie) quelle indicizzate con X. Codifica binaria: 100*bb*110. Flags modificati: nessuno.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`STX addr8`	10000110	86	Pagina zero	2	3
`STX addr16`	10001110	8E	Assoluto	3	4
`STX addr8,Y`	10010110	96	Indicizzato pagina zero, Y	2	4
	10011110	9E	Non utilizzato		

4.3.50 STY

Lo mnemonico corrisponde a **ST**ore **Y**. Salva il contenuto del registro Y in una locazione di memoria. Si noti che, tra le modalità di indirizzamento supportate, mancano (per ragioni piuttosto ovvie) quelle indicizzate con Y. Codifica binaria: 100*bb*100. Flags modificati: nessuno.

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`DEC addr8`	10000100	84	Pagina zero	2	3
`DEC addr16`	10001100	8C	Assoluto	3	4
`DEC addr8,X`	10010100	94	Indicizzato pagina zero, X	2	4
	10011100	9C	Non utilizzato		

4.3.51 TAX

Transfer **A**ccumulator to **X**. Copia nel registro X il contenuto dell'Accumulatore. I flags di segno N e zero Z vengono modificati secondo i contenuti di A. Codifica binaria: 10101010.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`TAX`	10101010	AA	Implicito	1	2

4.3.52 TAY

Transfer **A**ccumulator to **Y**. Copia nel registro Y il contenuto dell'Accumulatore. I flags di segno N e zero Z vengono modificati secondo i contenuti di A. Codifica binaria: 10101000.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`TAY`	10101000	A8	Implicito	1	2

4.3.53 TSX

Transfer **S**tack Pointer to **X**. Copia il contenuto dello Stack Pointer nel registro X. Si noti che questa è *l'unica istruzione* che consente l'accesso diretto al puntatore dello stack: per portarlo in Accumulatore o copiarlo in una locazione arbitraria di memoria occorre una seconda istruzione, ad esempio TXA o STX. Codifica binaria: 10111010.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`TSX`	10111010	BA	Implicito	1	2

4.3.54 TXA

Transfer **X** to **A**ccumulator. Copia nell'Accumulatore il contenuto del registro X. I flags di segno N e zero Z vengono modificati secondo i contenuti di A. Codifica binaria: 10001010.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`TXA`	10001010	8A	Implicito	1	2

4.3.55 TXS

Transfer **X** to **S**tack Pointer. Copia il contenuto del registro X nello Stack Pointer S. Si noti che questa è *l'unica istruzione* che consente l'impostazione diretto del puntatore dello stack. Codifica binaria: 10011010.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`TXS`	10011010	9A	Implicito	1	2

4.3.56 TYA

Transfer **Y** to **A**ccumulator. Copia nell'Accumulatore il contenuto del registro Y. I flags di segno N e zero Z vengono modificati secondo i contenuti di A. Codifica binaria: 10011000.

Flags modificati:

N	*V*	–	*B*	*D*	*I*	*Z*	*C*
*						*	

Linea sorgente	Opcode		Indirizzamento	Bytes	Cicli
	Binario	Hex			
`TYA`	10011000	98	Implicito	1	2

4.4 Sinottico dei tempi di esecuzione.

Si propone, per la massima comodità del lettore, una tabella completa che riassume i tempi di esecuzione per tutti gli opcode documentati. Sarà così estremamente semplice ricercare le istruzioni che garantiscono le migliori prestazioni: sono quelle i cui tempi sono riportati nelle prime tre colonne a sinistra della tabella, corrispondenti alle tre modalità di indirizzamento più vantaggiose in assoluto, eccetto ovviamente per le istruzioni dei gruppi **BI** (Branch Instructions), **UJR** (Unconditional Jump and Returns) e **SOI** (Stack Operation Instructions).

In appendice, come ultima pagina del presente testo, è riportata la stampa in formato A3 della ISA in formato esteso, con codifica, tempi di esecuzione e lunghezza dell'istruzione.

Legenda:
* Aggiungere **1 ciclo** se l'indirizzo destinazione oltrepassa il limite di pagina del Program Counter attuale.
** Aggiungere **1 ciclo** se il salto viene eseguito. Aggiungere **2 cicli** se il salto viene eseguito verso una locazione in una pagina diversa.

	Implicito	Immediato	Pag. zero	Pag. zero, X	Assoluto	Assoluto, X	Assoluto, Y	(zero, X)	(zero),Y	Relativo	Indiretto
ADC		2	3	4	4	4*	4*	6	5*		
AND		2	3	4	4	4*	4*	6	5*		
ASL	2		5	6	6	7					
BCC										2**	
BCS										2**	
BEQ										2**	
BIT			3		4						
BMI										2**	
BNE										2**	
BPL										2**	
BRK	7										
BVC										2**	
BVS										2**	
CLC	2										
CLD	2										
CLI	2										
CLV	2										
CMP		2	3	4	4	4*	4*	6	5*		
CPX		2	3		4						
CPY		2	3		4						
DEC			5	6	6	7					
DEX	2										
DEY	2										
EOR		2	3	4	4	4*	4*	6	5*		
INC			5	6	6	7					
INX	2										
INY	2										
JMP					3						5
JSR					6						
LDA		2	3	4	4	4*	4*	6	5*		

Legenda:
* Aggiungere **1 ciclo** se l'indirizzo destinazione oltrepassa il limite di pagina del Program Counter attuale.
** Aggiungere **1 ciclo** se il salto viene eseguito. Aggiungere **2 cicli** se il salto viene eseguito verso una locazione in una pagina diversa.

	Implicito	Immediato	Pag. zero	Pag. zero, X	Assoluto	Assoluto, X	Assoluto, Y	(zero, X)	(zero),Y	Relativo	Indiretto
LDX		2	3	4	4		4*				
LDY		2	3	4	4	4*					
LSR	2		5	6	6	7					
NOP	2										
ORA		2	3	4	4	4*	4*	6	5*		
PHA	3										
PHP	3										
PLA	4										
PLP	4										
ROL	2		5	6	6	7					
ROR	2		5	6	6	7					
RTI	6										
RTS	6										
SBC		2	3	4	4	4*	4*	6	5*		
SEC	2										
SED	2										
SEI	2										
STA			3	4	4	4*	4*	6	5*		
STX			3	4	4						
STY			3	4	4						
TAX	2										
TAY	2										
TSX	2										
TXA	2										
TXS	2										
TYA	2										

Capitolo 5

Esempi di programmazione Assembly.

5.1 Strumenti di lavoro.

Per chi volesse procedere nel modo tradizionale, sarà necessario fare uso di una macchina d'epoca o di un emulatore adeguato, dotati di un Assembler e magari di una cartridge con monitor/debugger. Per il 6510, con particolare riguardo al Commodore 64, tra gli oltre sedici ambienti di sviluppo *low level* commercializzati nell'arco di due decenni i principali tools a cui fare riferimento sono:

- Abacus 64 Development System
- Commodore Assembler
- Merlin 64
- Mikro 64
- Panther 64
- Turbo Assembler
- Zeus 64

Tali ambienti (eccetto forse il Mikro Assembler) vanno molto oltre le esigenze di un principiante e richiedono l'attento studio dei rispettivi manuali per l'apprendimento della particolare sintassi utilizzata. In alternativa, lasciando a margine i vari storici Assembler 6502 CLI[1] per DOS, la soluzione più semplice sarà fare uso di un ambiente come CBM PRG Studio, che è quello effettivamente utilizzato per la maggior parte degli esempi di questo testo. L'Autore incoraggia comunque la modalità di sviluppo in cross-development più tradizionale, basata su un editor avanzato per programmatori e su un Assembler command line, come ad esempio 64Tass (da non confondere con l'omonimo ambiente nativo per C64, né tantomeno con il glorioso TASM DOS della Borland) o Win2C64 sviluppato da Aart Bik. Altri fondamentali siti di riferimento per il 65xx sono:

- 6502.org
- CBM8bit.com
- Soft6502.

Tutti gli esempi di codice qui forniti sono stati assemblati con CBM Prg Studio e provati con l'emulatore VICE.

5.2 Lavorare senza un Assembler.

Esistono sostanzialmente due strade per inserire in memoria e mandare in esecuzione codice macchina senza l'uso di un Assembler: sfruttare un linguaggio di alto livello (es. BASIC, FORTH, Pascal, ...) oppure fare uso di un programma dedicato, un monitor LM, disponibile sia su cartridge che come software autonomo, che come minimo consente l'immissione diretta di valori esadecimali in memoria e quasi sempre offre anche una funzione Assemble per codificare «al volo» singoli opcode seguiti dai relativi operandi.

Il presente testo è volutamente svincolato da ogni particolare architettura o macchina, anche se è sempre sotteso il riferimento al più venduto home computer basato su 6510: il Commodore 64. Tuttavia, anche

[1] CLI è acronimo di Command Line Interface: si riferisce in generale al software privo di una interfaccia utente interattiva, lanciato da linea di comando con eventuali parametri e/o file di configurazione per modificarne il comportamento e comunicare i dati necessari all'elaborazione.

Indirizzo	Istruzione			Etichetta	Mnem.	Operando	Commento
				START	LDA	#$17	Converte maiuscolo/min.
					STA	$D018	Carica il valore nel registro
					RTS		Ritorno al BASIC

Figura 5.2.1: Listato Assembly per l'esercizio di codifica manuale.

molte schede didattiche[2] (tra cui gli innumerevoli derivati dei giustamente famosi KIM-1 e SYM-1) e varie schede industriali supportano un interprete per linguaggi di alto livello su EPROM, il che consente l'uso di metodi semplificati per l'incorporazione di brevi routine in linguaggio macchina, codificate manualmente. Quasi tutti i testi citati dedicano almeno qualche sezione (quando non un intero capitolo) a queste tecniche, e si possono facilmente formare pile ideali di riviste di programmazione che raggiungono il quinto o sesto piano di un edificio scegliendo quelle che nelle decadi Ottanta e Novanta dello scorso secolo hanno presentato listati BASIC letteralmente infarciti di statement DATA contenenti appunto routine in codice macchina[3].

Presentiamo qui rapidamente un esempio di tale possibilità per Commodore 64 (adattabile senza sforzo a molti altri home computer analoghi), ovviamente basato su una codifica manuale delle istruzioni, senza dilungarci eccessivamente a causa delle innumerevoli limitazioni di tale tecnica, la quale tuttavia mantiene un elevatissimo valore didattico. Chiunque voglia fregiarsi del titolo di «Programmatore Assembly» non può prescindere da una buona pratica nella codifica e decodifica manuale di dump esadecimali, tra esercizi di encoding e ML monitor.

La figura (5.2.1) mostra un classico esempio di *coding sheet*, il modulo di codifica Assembly sul quale ha invariabilmente iniziato a lavorare chi è stato abbastanza «fortunato» generazionalmente da evitare in tutto o in parte le schede perforate con codifica Hollerith. Tabulati simili esistevano anche per COBOL e FORTRAN, per la cronaca. Si inizia scrivendo le istruzioni e gli operandi necessari all'operazione da compiere, in questo caso banale ma ben visibile: la conversione dei caratteri a video dal set maiuscolo a quello minuscolo. Le operazioni richieste si articolano in sole tre fasi:

1. Caricamento di un valore immediato in Accumulatore;
2. Copia di tale valore in un registro del VIC, il chip di controllo video;
3. Ritorno al BASIC.

Al lettore basti per il momento sapere che il valore caricato esegue l'azione richiesta, a carico del controller video. Il nostro brevissimo programma verrà caricato ed eseguito a partire dalla locazione C000h, come peraltro suggerito dai manuali Commodore.

A questo punto, usando l'elenco completo delle istruzioni fornito nel precedente capitolo, si verifica che l'opcode esadecimale corrispondente a `LDA` in modalità immediata è A9h, e questo consente già di completare l'encoding della prima istruzione, che occupa due byte:

Indirizzo	Valore
C000h	A9h
C001h	17h

Riportiamo questi dati nel foglio di codifica, negli appositi campi, e passiamo alla linea successiva. L'istruzione `STA` accetta in questo caso un indirizzo assoluto a 16 bit, ed è quindi codificata come 8Dh. Avremo quindi:

[2] Al giorno d'oggi la realizzazione di tali schede è più che mai a portata di hobbista, grazie all'universale disponibilità di schemi elettrici collaudati e files gerber/excellon già pronti o addirittura sbrogli editabili prerealizzati per i CAD EDA amatoriali più diffusi, e grazie soprattutto alla possibilità di fare realizzare a prezzi irrisori in uno dei tanti service del Far East dei PCB professionali a doppia faccia con fori metallizzati, piste stagnate, serigrafia e solder livellato, anche in singolo esemplare.

[3] Listati quasi sempre generati da appositi tools a partire dal codice binario già assemblato, in realtà, e spesso comprensivi di checksum di riga.

Indirizzo	Istruzione			Etichetta	Mnem.	Operando	Commento
C000	A9	17		Start	LDA	#$17	Converte maiuscolo/min.
C002	8D	18	D0		STA	$D018	Carica il valore nel registro
C005	60				RTS		Ritorno al BASIC

Figura 5.2.2: Codifica completa dell'esempio.

Indirizzo	Valore
C002h	8Dh
C003h	18h
C004h	D0h

Si noti ancora una volta come la convenzione little endian impone di porre il byte meno significativo all'indirizzo di memoria più basso.

L'ultima istruzione presente è `RTS`, codificata univocamente con 60h (indirizzamento implicito, nessun operando) e questo completa la codifica, come si vede in figura (5.2.2).

L'ultimo passaggio necessario è la conversione da esadecimale a decimale, poiché il BASIC V2 non accetta numeri nel primo formato (a meno di un piccolo sforzo aggiuntivo di programmazione). Si tratta comunque di un banale lavoretto, per pochi byte.

Hex	A9	17	8D	18	D0	60
Dec	169	23	141	24	208	96

L'indirizzo C000h corrisponde a 49.152 in decimale. A questo punto, il gioco è fatto: non resta che inserire tali codici in memoria e poi mandare in esecuzione la subroutine, tramite un semplicissimo programmino BASIC.

```
10 DATA 169,23,141,24,208,96
20 AD=49152
30 FOR I=0 to 5: READ CO
40 POKE AD+I,CO:NEXT
50 SYS 49152
```

Molti testi e riviste iniziano un intero percorso da qui, dove noi invece volutamente ci fermiamo. In BASIC è facile infatti realizzare con poche decine di linee di codice ogni genere di supporto all'immissione di codici esadecimali, fino a sofisticati monitor che consentono immissione diretta dei valori (anche con checksum per gruppi), dump di memoria formattati su video e stampante, patching di singoli byte, ricerche e sostituzioni. Un mondo che ha il suo fascino e un sapore «retro» del tutto particolare, sempre valido quando si parli di pochi byte di linguaggio macchina, ma che storicamente ha poi lasciato il posto agli Assembler e alla loro evoluzione: da grezzi codificatori di singole istruzioni a sofisticati strumenti multipasso, multifile, con parser avanzati, gestione di include e potenti sistemi di macro.

5.3 Lavorare con l'Assembler.

Con le figure (5.2.1) e (5.2.2) abbiamo già anticipato, rispettivamente, il formato di una linea sorgente in Assembly e anche il formato del relativo listato prodotto dall'Assembler. Vi sono dei campi fissi, la cui struttura esatta dipende strettamente dall'ambiente di lavoro utilizzato: in linea generale, più si va indietro nel tempo (e in basso nella scala dei costi di licenza, all'epoca), più rigido sarà il formato e minore la flessibilità sui delimitatori.

Molti Assembler primitivi hanno problemi se si inseriscono spazi o tabulazioni dopo le virgole, le parentesi o altri delimitatori. Vale quindi a fortiori il suggerimento di studiare con attenzione la manualistica (spesso poche paginette, composte con un primitivo editor DOS o direttamente su un home computer) del proprio ambiente preferito.

Poiché un testo come il presente impone delle scelte, si è preferito un cammino di minima resistenza: la sintassi generica e destrutturata dell'Assembler incluso in CBM Prg Studio, che risulta blandamente compatibile con quasi tutti i più diffusi ambienti nativi, ed è comunque facilmente convertibile tramite qualche search&replace col proprio editor preferito.

Tornando ai campi della linea Assembly, abbiamo la seguente struttura generale:

Label	Opcode	Operando/i	Commento

Il campo Label, opzionale, serve principalmente come riferimento per salti (assoluti o condizionati), salvo altri usi che vedremo in seguito. Deve essere separato da almeno uno spazio dal campo successivo, Opcode, che ovviamente non è opzionale su una generica linea di istruzioni. Gli Operandi, se presenti (il lettore ricordi che esistono nella ISA un totale di 24 istruzioni su 56 che ammettono solo l'indirizzamento implicito, senza operandi di sorta), devono seguire l'opcode, separati da almeno uno spazio, e rispettare il formato universale per gli indirizzamenti indiretti (virgola, parentesi) e immediati (cancelletto) già presentato durante l'esame della ISA, assieme ai caratteri letterali A, X, Y che identificano rispettivamente Accumulatore e registri indice X e Y.

Segue il commento, anch'esso opzionale, delimitato in genere da un punto e virgola (semicolon): una regola che ammette varie eccezioni. Si noti che in realtà una linea può contenere il solo commento, il che è peraltro un'ottima prassi di programmazione[4] per documentare il codice in maniera meno smozzicata e telegrafica rispetto a quanto consentono i commenti riga per riga, che troppo spesso si riducono a banali descrizioni testuali dell'istruzione eseguita, prive di vero contenuto informativo sulla effettiva semantica di quanto sta avvenendo.

Naturalmente, oltre alla possibilità di immettere una mera sequenza ordinata di linee di istruzioni, gli Assembler hanno bisogno anche di altre informazioni (una su tutte, l'indirizzo iniziale a cui assemblare il codice!), ed offrono alcune caratteristiche aggiuntive. A tale scopo esistono le *direttive* o pseudo-op, ed è qui che la fantasia dei progettisti si è sbizzarrita per creare interi sistemi di direttive e macro proprietari e totalmente incompatibili gli uni con gli altri, nella più caotica mancanza di standard e unificazioni (che caratterizza da sempre la variopinta galassia degli Assembler, a dire il vero).

Per non trasformare questo breve testo in una improponibile trattazione enciclopedica in parallelo sulla trentina di (cross-)Assembler 6502 esistenti (molti dei quali introvabili o irrimediabilmente obsoleti), ci limitiamo unicamente a qualche indicazione relativa al CBM Prg Studio, come anticipato, rimandando al relativo help in linea per ulteriori dettagli.

5.3.1 Sintassi e direttive CBM Prg Studio.

CBM Prg Studio (nel seguito anche CPS, per brevità) usa un Assembler a tre passi[5] che supporta macro, assemblaggio condizionale e alcuni operatori avanzati. Si ponga sempre attenzione al fatto che la sintassi rimane piuttosto rigida dal punto di vista posizionale: le labels in particolare *devono sempre* essere essere collocate nella prima colonna, come in tutti gli esempi che forniremo nel seguito.

La specifica dell'indirizzo iniziale deve sempre rigidamente *precedere ogni altra istruzione* nel sorgente. Può essere preceduta unicamente dalla dichiarazione di «variabili», ossia labels alle quali viene esplicitamente assegnato un valore. L'indirizzo viene determinato nel sorgente nei seguenti due modi alternativi:

```
*=$0800  oppure  *=2048
@=$0800  oppure  @=2048
```

Questo introduce direttamente il secondo argomento: il *formato di immissione dei valori numerici.* Nel seguito n rappresenta un singola cifra.

[4] A maggior ragione quando si usano ambienti nativi, limitati a 40 caratteri per riga.

[5] Questo significa, brevemente, che il sorgente viene analizzato tramite un parser che effettua **tre** distinti cicli di lettura sequenziale per risolvere le «forward references» alle label e migliorare la generazione del codice.

Formato	Base
nn	Decimale
$nn	Esadecimale
@nn	Ottale
%nnnnnnnn	Binario

I commenti sono introdotti da un singolo ';'. Di fatto, tutto ciò che segue tale carattere sarà **ignorato** dall'Assembler fino al termine della riga (CR/LF o equivalenti). Come da tradizione nel mondo degli Assembly Commodore, gli operatori '<' e '>' (minore e maggiore, rispettivamente) sono utilizzati per referenziare *il byte meno significativo ed il più significativo*, rispettivamente, da un valore a 16 bit. Esempio:

```
LDA #<$FB2A      ; Carica in accumulatore 2Ah
LDX #>$55AA      ; Carica in X 55h
```

Si presti attenzione al fatto che la precedenza degli operatori **varia notevolmente** secondo il tipo di Assembler. Per questo motivo CPS supporta due varianti fondamentali:

HiLo Prima separa il byte (alto o basso rispettivamente) e poi applica l'offset;

Calc Prima calcola l'indirizzo e solo come ultimo passaggio estrae il suo byte alto (o basso).

La prima opzione è il default. Quindi, dato il codice seguente:

```
    *=$1000
1000 loop    NOP
1001         LDA #>loop+$20
```

Se è attiva l'opzione **HiLo**, le operazioni eseguite saranno nell'ordine

1. Viene estratto il byte più significativo dall'indirizzo corrispondente alla label «loop»;
2. Viene aggiunto a tale valore un offset esadecimale pari a 20h (32_{10}).

Il risultato sarà quindi pari a 30h.

Gli operatori logici supportati sono `AND, OR, NOT, XOR, <<, >>`. CBM Prg Studio supporta inoltre **solo espressioni semplici** per gli indirizzi, nelle quali compare in via esclusiva uno e un solo simbolo scelto tra '+', '-', '*' o '/'.

Tra le direttive supportate, le più rilevanti ai fini del presente testo sono le seguenti:

- **IncAsm**

Include un altro file sorgente, con profondità teoricamente illimitata, purché non sussistano riferimenti circolari.

- **Operator Calc | HiLo**

Imposta la precedenza degli operatori, come spiegato poco sopra. Il default, come già indicato, è **HiLo**.

- **Relocate «address» | OFF**

La direttiva `Relocate` viene utilizzata per modificare l'indirizzo di memoria a cui viene assemblato il codice. Tale direttiva modifica l'indirizzo effettivo in cui viene assemblato il codice oggetto, in particolare modifica tutte le label e le destinazioni delle istruzioni di salto facendo corrispondere la prima istruzione che segue al nuovo indirizzo impostato. Ad esempio, su Commodore 64 è del tutto normale assemblare codice dotato di un loader che sarà caricato all'indirizzo di default del BASIC (per non dover specificare LOAD «Program», 8, 1 ed evitare poi di dover digitare manualmente un indirizzo per la SYS che manda in esecuzione il codice). Il loader provvederà a copiare e poi eseguire una sostanziale porzione di codice ad un diverso indirizzo di memoria, ad esempio `$C000`. Usando la direttiva Relocate si istruisce allora l'Assembler a produrre codice destinato alla rilocazione dinamica a runtime. Come esempio generale di tale strategia, che sarà molto utile in seguito, presentiamo il listato assemblato di un loader BASIC realizzato interamente tramite direttive Assembly con CBM Prg Studio e facilmente adattabile ad altri ambienti. Il loader offre l'enorme vantaggio di consentire il «`LOAD and RUN`» in modo totalmente automatico, senza dover ricordare l'indirizzo di avvio e senza dover manualmente digitare un comando `SYS`. Ciò rende estremamente professionale l'applicazione, al pari di software commerciali e della stragrande maggioranza dei giochi, e ne facilita grandemente la gestione e la distribuzione.

```
Line   Addr  Code         Source
-------------------------------------------------------------------------------
00001  0000               ;************************************
00002  0001               ; CODICE "UNIVERSALE" PER STARTUP BASIC
00003  0001               ;************************************
00004  0001                         * = $0801
00005  0801               ;-------------------------------------------------
00006  0801  0B 08                  WORD BASEND ; INDIRIZZO DELLA PROSSIMA LINEA
00007  0803  E4 07                  WORD 2020   ; NUMERO DI LINEA -> ANNO DI STESURA
00008  0805  9E                    BYTE $9E    ; TOKEN PER "SYS"
00009  0806  32 33 30               TEXT "2304" ; INDIRIZZO DI AVVIO: $0900
00010  080A  00                    BYTE 0      ; TERMINATORE DI LINEA
00011  080B  00 00       BASEND    WORD 0      ; TERMINATORE DI PROGRAMMA
00012  080D               ;-------------------------------------------------
00013  080D               ;************************************
00014  080D               ;** INIZIO CODICE APPLICAZIONE      **
00015  080D               ;************************************
00016  080D
00017  080D                         * = $0900
00018  0900               ;** Qui inizia il codice utente in Assembly
00019  0000               ...
000xx  yyyy               Relocate $C000
```

Il codice che segue la direttiva nel listato esemplificativo sarà assemblato per essere eseguito alla locazione indicata, ma l'immagine binaria del sorgente sarà compatta. Purtroppo questo aspetto tende a confondere i neofiti: manipolare direttamente il program counter con l'operatore *, impostando `* = $C000` prima del codice da rilocare produrrebbe per poche decine di locazioni un file binario di oltre 47kib esteso dall'indirizzo `$0801` a `$Cxxx` (in funzione della lunghezza del codice), vale a dire un file di dimensioni eccessive e con un enorme «buco» che creerebbe enormi problemi all'atto del caricamento, andando tra l'altro in questo caso ad interferire con la ROM del BASIC allocata a partire da $A000. La soluzione è, appunto, l'uso della direttiva e la stesura di un apposito codice (nel nostro esempio, con inizio a `$0900`) che si occupi di copiare i byte di codice binario che seguono al corretto indirizzo previsto per l'esecuzione. Supponendo, solo per fissare le idee, che il codice preliminare e di copia richieda in tutto 64 bytes e che il resto del codice eseguibile occupi effettivamente due pagine di memoria, ossia 512 bytes, il codice di copia non farà altro che copiare le due pagine di RAM da `$0940` a `$0B40` nella memoria tra `$C000` e `$C200`, trasferendo poi il controllo all'indirizzo iniziale con una istruzione di salto `JMP`. Ecco come si effettua (manualmente!) la rilocazione effettiva a runtime, mentre a tempo di assemblaggio provvede la direttiva qui illustrata.

La direttiva si applica dal punto in cui compare nel sorgente fino a che non si incontra un'altra direttiva `Relocate` oppure la fine del file.

- **Target «target name»**

Serve a specificare il tipo di macchina di destinazione. I valori ammissibili per l'espressione «target_name» sono i seguenti:

1. TGT_C128
2. TGT_C16
3. TGT_C64
4. TGT_PETBV2
5. TGT_PETBV4
6. TGT_PLUS4
7. TGT_VIC20
8. TGT_VIC20_3K
9. TGT_VIC20_8K

5.3.2 La gestione delle variabili.

Come già in più punti anticipato, in linguaggio Assembly le «variabili» non sono altro che mere *locazioni di memoria etichettate* ed usate come sorgente o destinazione per varie istruzioni. La convivenza di dati e istruzioni imposta dall'architettura Von Neumann, se da un lato offre una notevole flessibilità, obbliga anche a compiere delle scelte di design che possono confondere il programmatore di alto livello e produrre errori estremamente difficili da tracciare per chi è agli inizi, tipicamente causati da: sovrascrittura involontaria di codice, tentativi di esecuzione di dati, errato ordine di memorizzazione dei byte.

Ricordiamo al lettore che nelle architetture MMIO l'etichetta spesso corrisponde ad un registro (tipicamente ampio un byte) di un chip di I/O specializzato, che sia il controller video o piuttosto un chip seriale, oppure ad una locazione di sistema (possibilmente in pagina zero). Questo è il caso più banale e intuitivo. Ma allo stesso modo si possono etichettare zone di memoria usate poi come «variabili» multibyte: qui occorre ricordare che le CPU a 8 bit di nostro interesse **non gestiscono in alcun modo tali variabili**, se non per ciò che concerne l'unico registro interno a 16 bit, il Program Counter (in particolare con il fetch automatico della istruzione successiva e la manipolazione indiretta del PC tramite le istruzioni di salto e gli interrupt). Questo significa che quanto già considerato in ordine alla memorizzazione little endian, nel caso di variabili numeriche, *rimane interamente a carico del programmatore Assembly.* Quindi occorre gestire *manualmente* la successione dei byte nelle rispettive locazioni di memoria per tutte le istruzioni della categoria DTI, Data Transfer.

Nella struttura dei più semplici programmi Assembly a singolo file sorgente che avremo modo di esemplificare, la collocazione ideale delle variabili generiche utilizzate (quindi non le eventuali locazioni di pagina zero referenziate, o altre locazioni di sistema) sarà in coda o in testa al codice stesso. Si tratta di una consuetudine di codifica sbrigativa e inerentemente poco robusta, ma adattissima al contesto didattico e facilissima da monitorare.

Qualunque ambiente Assembler (e CPS non fa certo eccezione) consente, oltre alla banale etichettatura delle singole linee di codice, l'**assegnazione esplicita** di un indirizzo ad una label, che crea una *costante* usando una qualsiasi tra le basi numeriche supportate e ricordando che le etichette sono obbligatoriamente collocate a colonna 1:

```
BASICSTART = $0801
SCREENRAM  = $0400
BASICROM   = $A000
VIC        = $D000
SID        = $D400
COLORRAM   = $D800
CIA1       = $DC00
CIA2       = $DD00
KERNALROM  = $E000
```

Inoltre sono supportate direttive specifiche per l'inizializzazione di intere aree di memoria e di singole «variabili» di tipo byte, word, long (24 bit, ma solo su 65816) e floating point, oltre al supporto di vari tipi di stringhe[6].

```
var1    byte $A5
var2    byte %10100101
var3    word $F1CA
var4    word %1111000010101010
var5    fltp 1.414213562373             ; Il separatore decimale è il punto '.'
; Sono ammessi valori multipli, ciascuno occuperà 1 byte o 1 word etc.
array1  byte 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233
```

Altre eventuali direttive avanzate saranno illustrate al momento dell'utilizzo negli esempi.

[6] Le tipologie di stringhe possono essere:

1. BASIC/Pascal ossia con lunghezza prefissa (in un byte, nel nostro caso);
2. Null-terminated o ASCII-zero (C-style) con terminatore nullo finale;
3. CBM table, nelle quali l'ultimo carattere ha il MSB posto ad uno per indicare la terminazione della stringa senza fare uso di un carattere aggiuntivo. Le tabelle delle keyword BASIC, ad esempio, sono realizzate in questo modo.

CBM Prg Studio non supporta quest'ultima tipologia tramite una direttiva specifica. Inoltre le stringhe possono contenere caratteri PETSCII (se delimitate dalle doppie virgolette) oppure codici schermo (delimitate da apici singoli). Si veda l'help in linea dell'ambiente di sviluppo per ulteriori dettagli.

5.4 Esempi di codice Assembly.

In questa sezione presentiamo una serie di esempi elementari che consentiranno progressivamente di padroneggiare i costrutti e gli idiomi di base dell'Assembly 6502.

Nota importante riguardo i listati. Al fine di contenere la lunghezza dei sorgenti e dei listati, e limitare così nel complesso il numero di pagine dell'edizione cartacea, si è fatto ovunque un uso limitato di commenti *single-line* multilinea, riducendolo al minimo indispensabile e privilegiando ove possibile brevi commenti *inline* a margine del codice. Questo comporta alcune limitazioni, soprattutto nel listing dopo l'assemblaggio, a sua volta uno strumento fondamentale per verificare la lunghezza delle istruzioni e i tempi di esecuzione. Dove opportuno, vengono quindi presentati sia il mero sorgente, sia il listing in modo da evitare la perdita di contenuti conseguente il troncamento dei commenti nel listing stesso: una consuetudine peraltro molto diffusa nella manualistica di ogni livello.

5.4.1 Costrutti di controllo del flusso e idiomi caratteristici.

Il primo aspetto del linguaggio Assembly che disorienta il programmatore HLL è sicuramente la mancanza di costrutti come IF, FOR, WHILE in tutte le loro forme. Se da un lato i macroAssembler più evoluti forniscono un arsenale di macro e pseudo-istruzioni in tale senso, è essenziale padroneggiare quanto prima i numerosi metodi di controllo del flusso di programma offerti dalle istruzioni atomiche della ISA e i relativi costrutti più frequenti. Per comodità del lettore, si forniscono dei *template* Assembly per alcuni costrutti fondamentali, riportando l'equivalente in BASIC. Si ricordi che esistono in realtà innumerevoli forme equivalenti, quelle qui riportate sono solamente le più comuni per una prima familiarizzazione, sovente anche tra le più efficienti, ma assolutamente lontano da qualsiasi pretesa di esaustività.

5.4.1.1 Alcune forme di `IF...THEN`.

Per semplificare, nel seguito assumiamo di voler controllare i valori assunti da variabili ampie 1 byte, presenti in memoria come risultato di altre generiche operazioni a monte, che non ci interessa specificare.

```
BASIC:
IF N=0 THEN GOTO 10000
```

```
Assembly:
        LDA N           ; Il caricamento imposta il flag Z se N è nullo
        BEQ L10000
        ...             ; Esegue le azioni previste per il caso N<>0
        RTS

;** Si arriva qui solo se N=0
L10000  ...
```

Questo esempio ci consente di chiarire immediatamente un concetto. L'uso di `BEQ` oppure di `BNE`, e più in generale di una logica diretta o negata inerente il flag interessato, dipende esclusivamente da due fattori:

1. **Lunghezza del codice** presente nei due rami dell'albero di esecuzione.
2. **Frequenza** con la quale viene intrapreso il salto.

Focalizziamo al momento la nostra attenzione sul primo punto[7]. Poiché i salti relativi sono limitati rispettivamente a +128 e -127 locazioni di distanza dall'attuale Program Counter, e per giunta risultano penalizzati di un ciclo (passando così da 3 a 4) se la destinazione risiede in una pagina di memoria diversa dall'attuale, è responsabilità del programmatore minimizzare i costi ed evitare il rischio che l'Assembler segnali un errore in caso di salto relativo ad una locazione che risulta irraggiungibile. Se le «azioni previste» sono davvero poche istruzioni Assembly, questo template rimane perfettamente valido e applicabile (non a caso in molti ambienti questa è una vera e propria macro), purché ci si accerti leggendo il listing finale che non venga superato il limite di pagina nel codice.

[7] Per il secondo punto, si ricordi che ogni volta che il salto viene intrapreso si incorre in una penalità di un ciclo, che può salire a ben due cicli se l'indirizzo di destinazione si trova in una pagina di memoria diversa da quella in cui risiede l'istruzione di branch.

Le soluzioni alternative standard sono diverse, secondo le specifiche esigenze del codice. Quando siamo ragionevolmente certi di dover eseguire un numero rilevante di operazioni, una delle idee migliori è quella di creare una subroutine (o invocarne una esistente in firmware), che viene richiamata condizionatamente:

```
Assembly:
          LDA N
          BNE NotZero
          JSR Subroutine ; Esegue una subroutine solo se N=0
NotZero   ...
          ...
Subroutine
          ...
          RTS
```

In alternativa, la soluzione più conservativa e sicura in assoluto (sebbene con un costo complessivo in cicli non trascurabile) si configura come la seguente:

```
Assembly:
          LDA N
          BNE NotZero
          JMP Label      ; Esegue il salto solo se N=0
NotZero   ...
          ...
Label     ...
```

In questo modo il salto relativo sarà sicuramente entro i limiti: di nuovo, una lettura del listing ci garantirà che non vi sia sconfinamento di pagina, caso sempre possibile (in tale evenienza non rimane che spostare il codice prima o dopo altre operazioni, se possibile). In ambedue i casi si è fatto uso di una logica negata: si ponga attenzione alla coerenza mantenuta tra il tipo di controllo effettuato (`BNE`: Branch if Not Equal to Zero) e la nomenclatura della label `NotZero`, il che è la prima forma di auto-documentazione del codice.

Naturalmente questi pochi e semplici esempi sono ben lontani dell'esaurire le innumerevoli possibili casistiche. Sicuramente sono orientati alla strutturazione del codice e alla programmazione difensiva, perché in Assembly il confine tra flessibilità & performance vs spaghetti code è una linea estremamente sottile, a maggior ragione per chi inizia.

Tenendo presente quanto appena asserito, continuiamo con qualche altro utile esempio. Ovviamente, il caso simmetrico per `N<>0` è strettamente analogo:

```
BASIC:
IF N<>0 THEN GOTO 10000
```

```
Assembly:
          LDA N
          BEQ IsZero
          JMP L10000     ; Esegue il salto solo se N<>0
IsZero    ...
          ...
L10000    ...
```

Allo stesso modo, sempre operando in logica negata, è possibile condizionare l'esecuzione al controllo del segno (in generale, del MSB) di un valore a 8 bit:

```
BASIC:
IF N<0 THEN GOTO 10000
```

```
Assembly:
          LDA N
```

```
          BPL  IsPos
          JMP  L10000       ; Esegue il salto solo se N<0
IsPos     ...
          ...
L10000    ...
```

```
BASIC:
IF N>=0 THEN GOTO 10000
```

```
Assembly:
          LDA  N
          BMI  IsNeg
          JMP  L10000       ; Esegue il salto solo se N>=0
IsNeg     ...
          ...
L10000    ...
```

Ovviamente il confronto con lo zero non è che un caso particolare di una situazione più generale, come illustrato nel seguito. Confrontare due byte A e B può avere banalmente tre distinti esiti, che corrispondono ad altrettante diverse configurazioni dei flag fondamentali N, Z e C dopo l'esecuzione di una istruzione `CMP` (o equivalenti `CPX, CPY`):

A::B	N	Z	C
A<B	1	0	0
A=B	0	1	1
A>B	0	0	1

Come si può notare, il flag N qui assume sempre valori complementari rispetto al Carry. Si tenga sempre presente che la `CMP` ed equivalenti operano, in realtà, una **sottrazione implicita**. La tabella sopra riportata è quindi lo strumento fondamentale per costruire *qualunque tipo di idioma di confronto*, con qualsiasi semantica, ordinando opportunamente i byte da verificare (laddove possibile) e controllando a valle lo stato di uno dei flag interessati con una istruzione di salto condizionato.

```
Assembly:
;** Codice didattico da eseguire direttamente nel monitor LM
;** verificando dopo ogni istruzione lo stato dei flag N,Z,C
          LDA  #32          ; A = 32
          CMP  #33          ; A < B, N=1, Z=0, C=0
          CMP  #32          ; A = B, N=0, Z=1, C=1
          CMP  #31          ; A > B, N=0, Z=0, C=1
```

```
BASIC:
IF A>35 THEN GOTO 10000
```

```
Assembly:
          LDA  #35
          CMP  A            ; 35 < A: N=1, Z=0, C=0
          BCC  L10000       ; Esegue il salto se 35<A
          ...               ; altrimenti esegue queste istruzioni
L10000    ...
```

```
BASIC:
IF A>=B THEN GOTO 10000
```

```
Assembly:
```

```
          LDA A
          CMP B
          BCS L10000      ; Esegue il salto se A>=B
          ...
L10000    ...
```

Un altro idioma tipico implementa l'istruzione di alto livello `IF...THEN...ELSE` o `ITE`. Questo costrutto richiede ovviamente una label aggiuntiva con funzione di *landing point* finale, equivalente dal punto di vista logico allo statement `ENDIF` tipico di molti linguaggi strutturati. Nel caso favorevole in cui le istruzioni da eseguire siano poche, o meglio ancora incorporate in una subroutine come già visto in precedenza, il template da implementare può essere il seguente:

```
BASIC:
IF A=10 THEN B=0 ELSE B=C*C-2*A
```

```
Assembly:
          LDA A
          CMP #10
          BEQ A_IS_TEN  ; Salta se A=10
          JSR Calc      ; Esegue il calcolo solo se A<>10
          JMP Label     ; Aggira il codice seguente, relativo al caso opposto

A_IS_TEN  LDA #0
          STA B
Label     ...
          ...
;** Implementa il calcolo di B=C*C-2*A per il caso A<>10
Calc      ...
          RTS
```

Per inciso, la chiamata a subroutine `JSR` equivale concettualmente alla `GOSUB` del BASIC: il flusso elaborativo principale viene temporaneamente interrotto, il controllo passa alla subroutine fino a quando non si incontra una istruzione `RTS`, la quale fa riprendere il flusso principale dall'istruzione immediatamente successiva alla chiamata a subroutine appena effettuata.

Vediamo ora due costrutti con le condizioni logiche che più spesso mettono in difficoltà i programmatori HLL. Si noti che, per la massima semplicità, viene implementata la valutazione in *short-circuit* (tipica ad esempio del linguaggio C), a cui abbiamo già accennato alla sottosez. 3.4.5 a pagina 28: se la prima condizione è falsa, l'AND sarà necessariamente falsa a prescindere dalla seconda condizione, che quindi non viene valutata. Rispettivamente, è sufficiente che la prima condizione valutata sia vera per rendere vero il risultato dell'OR.

Per una migliore comprensione, riprendiamo il concetto più volte richiamato di *condizione di indifferenza* e riscriviamo esplicitamente la tabella di verità compatta di ciascuna funzione, già vista in aggregato alla sezione 3.4 a pagina 25:

A	B	A OR B
0	X	B
1	X	1

Come già noto, la X indica che la variabile corrispondente può assumere indifferentemente uno dei valori booleani $\{0,1\}$. Esaminando la tabella per la funzione OR risulta lapalissiano che, quando $A = 0$, l'output dipende unicamente dal valore logico di B (che pertanto **deve** essere valutata), mentre nel caso opposto $A = 1$ il valore della funzione risulta già determinato dalla sola A, ed è pari ad 1 *per qualsiasi* valore di B.

A	B	A AND B
0	X	0
1	X	B

Considerazioni speculari valgono ovviamente per la funzione AND: è sufficiente *un solo valore nullo* in input a rendere parimenti nullo l'output della funzione, ed in tal caso non è necessario valutare anche l'espressione corrispondente a B.

Il lettore è esortato a notare come tutto ciò venga espresso con la medesima chiarezza dall'operatore ternario ITE (vedi sottosez. 3.4.5.1 a pagina 29) e in particolare dall'equazione 3.4.9:

$$\mathrm{ITE}(A, \beta, \gamma) := \begin{cases} \beta & \text{se } A = 1 \\ \gamma & \text{se } A = 0 \end{cases}$$

Le espressioni seguenti rappresentano AND e OR, come già visto in tabella nella sottosezione richiamata, e - di nuovo - rendono particolarmente evidenti i casi in cui rispettivamente $A = 0$ implica output 0 (AND) e $A = 1$ implica output 1 (OR), indipendentemente dal valore di B:

$$\begin{aligned} A \text{ AND } B &\quad \mathrm{ITE}(A, B, 0) \\ A \text{ OR } B &\quad \mathrm{ITE}(A, 1, B) \end{aligned}$$

Naturalmente, nel caso di condizioni complesse e con *side effect*, ovvero la cui valutazione ha effetti su altre variabili e più in generale influenza lo stato complessivo del sistema, sarà cura del programmatore fare in modo che tale eventuale espressione critica *venga valutata sempre*, ponendola per prima in questi idiomi o (più raramente) modificando opportunamente il codice per eseguire in qualsiasi caso *ambedue* le valutazioni.

```
BASIC:
IF N=12 AND M=65 THEN GOTO 10000
```

```
Assembly:
          LDA N
          CMP #12     ; Se la prima condizione è falsa, la seconda non viene valutata
          BNE Next    ; Short Circuit Logic Evaluation
          LDA M
          CMP #65
          BEQ L10000
Next      ...
          ...
L10000    ...
```

```
BASIC:
IF N=22 OR M=96 THEN GOTO 10000
```

```
Assembly:
          LDA N
          CMP #22
          BEQ Label   ; Short Circuit Logic Evaluation
          LDA M
          CMP #96
          BEQ L10000
          ...
          ...
L10000    ...
```

Per concludere, vediamo un semplice esempio di confronti a 16 bit, con variabili collocate in generici indirizzi assoluti in memoria. Come si vede, il confronto di due byte procede con la logica implicita di un AND: se il primo confronto fallisce, non è necessario eseguire il secondo. Nel secondo esempio, si presume che l'indirizzo destinazione sia irraggiungibile per una branch.

```
BASIC:
IF N=12345 THEN GOTO 10000
```

```
Assembly:
          LDA N           ; Carica il byte meno significativo
          CMP #$39        ; 12345 = $3039
          BNE Next        ; Se non è uguale, inutile proseguire nel confronto
          LDA N+1         ; Carica il byte più significativo
          CMP #$30
          BEQ L10000      ; Si presume che Label sia entro le 128 locazioni
Next      ...
          ...
          RTS
L10000    ...
          ...
```

```
BASIC:
IF A=B THEN GOTO 10000
```

```
Assembly:
          LDA A
          CMP B
          BNE Next
          LDA A+1
          CMP B+1
          BNE Next
          JMP L10000
Next      ...
          ...
          RTS
L10000    ...
          ...
```

5.4.1.2 Salvataggio e ripristino dei registri.

In numerose situazioni applicative è essenziale preservare il contenuto dei registri principali prima di eseguire determinate subroutine, ad esempio quando si interagisce col BASIC o altri HLL, oppure nella gestione di interrupt.

Normalmente è possibile fare uso dello stack, con una semplicissima sequenza di PUSH all'inizio del lavoro e una di POP alla fine in ordine inverso. Per le situazioni ordinarie, va benissimo un banale template come il seguente (in quasi ogni ambiente di programmazione è possibile farne una macro, sempre che non esista già nella libreria del sistema di sviluppo):

```
;****************************************
;** Salvataggio registri CPU
;****************************************

Start     PHP
          PHA
          TXA
          PHA
          TYA
          PHA

;** Corpo della routine
;**       ...
;**       ...

Exit      PLA
          TAY
          PLA
          TAX
```

```
        PLA
        PLP

        RTS
;****************************************
```

L'unica nota, alquanto ovvia, è che occorrono due passaggi per i registri indice X e Y, in quanto la ISA manca decisamente di ortogonalità e non prevede istruzioni di PUSH e POP dedicate. Ovviamente il gioco può diventare molto più farraginoso se si intende inserirlo all'interno di subroutine separate, a maggior ragione se si desidera salvare anche il valore dello stack pointer assieme agli altri registri, perché l'indirizzo di ritorno da una `JSR` viene appunto salvato sullo stack stesso e le manipolazioni del registro SP, per definizione, non sono argomento per principianti - neppure su queste CPU. Ma lasciamo a margine queste situazioni assai specifiche.

Salvataggio dei registri in RAM. Bisogna ricordare che lo spazio di stack era una risorsa preziosa negli anni Ottanta, specialmente sui 65xx, e quindi occorreva usarlo con molta parsimonia: a maggior ragione vista la penalità prestazionale associata alle operazioni di PUSH e POP. Quasi sempre la soluzione ottimale, anche in termini di prestazioni, era fare uso di un banale buffer in memoria per salvare almeno i registri principali: Accumulatore, X, Y e il registro di stato P.

L'occupazione di memoria è decisamente molto modesta e il relativo codice di gestione decisamente intuitivo, se solo si tiene conto degli ovvi vincoli: ad esempio, il registro P è inaccessibile direttamente, non esistendo un ipotetico equivalente della coppia `TSX/TXS` per tale registro. L'unica opzione prevista dalla ISA è quella di salvare P sullo stack e poi trasferirne indirettamente il contenuto, con una istruzione `PLA`, nell'Accumulatore, e ovviamente eseguire la manovra opposta in fase di ripristino. Inoltre, le istruzioni di caricamento come `LDA` notoriamente modificano alcuni flag (in particolare N e Z) nello stesso registro P, e di questo occorre tenere debitamente conto in fase di ripristino dei valori.

Di seguito il codice relativo alle semplicissime routine di salvataggio e ripristino: per comodità operativa si assume di poter utilizzare la pagina zero per i salvataggi. Si noti solo che il ripristino dell'Accumulatore in `SAVEREG`, a rigore, non è strettamente necessario se stiamo passando il controllo ad una subroutine che certamente non ha necessità di tener conto del suo valore iniziale.

```
;****************************************
;** Salvataggio registri CPU
;****************************************

        *=$C000

;****************************************
;** Variabili in pagina zero
;****************************************

A_save = $F8
X_save = $F9
Y_save = $FA
P_save = $FB

;****************************************
;** SAVEREG
;** Salva A, X, Y, P in pagina zero
;****************************************

SAVEREG
        STA A_SAVE ; Copia l'Accumulatore
        PHP        ; Copia P in A
        PLA        ; usando lo stack
        STA P_SAVE ; Salva P
        LDA A_SAVE ;** Ripristina l'Accumulatore
        STX X_SAVE ; Copia X
        STY Y_SAVE ; Copia Y
        RTS
```

```
;**************************************
;** GETREG
;** Ripristina A, X, Y, P
;**************************************

GETREG
        LDA P_SAVE ; Precarica il registro P
        PHA        ; P su stack per evitare i side effects
        LDA A_SAVE ; Ripristina A, X, Y dalla memoria
        LDX X_SAVE
        LDY Y_SAVE
        PLP        ; Ripristina P
End     RTS
;**************************************
```

5.4.1.3 Loop e dintorni.

Il più semplice tipo di loop è quello **a decremento**, che risulta anche essere il più efficiente. Vale la pena di utilizzare sistematicamente i registri indice X e Y come variabili di induzione, ovunque ciò risulti possibile. Il codice seguente non richiede molte spiegazioni.

BASIC:
FOR I=10 TO 1 STEP −1:...:NEXT

```
Assembly:
        LDX #10    ; Il registro X è usato come variabile di induzione a decremento
Loop    ...
        DEX
        BNE Loop
```

Invio seriale di un buffer. Ricordando che anche le modalità di indirizzamento indicizzato fanno a loro volta uso dei registri indice X e Y, il passo più ovvio è quello di **abbinare le due cose**: ad esempio, per inviare ad un chip seriale tutti i byte di un buffer. Per arricchire un po' l'esempio, realizziamo un loop ad *incremento* e supponiamo che l'indirizzo assoluto in memoria di tale buffer sia noto solo a runtime e memorizzato in pagina zero, alla coppia di indirizzi `$FB` e `$FC`. Faremo quindi uso di una post-indicizzazione tramite il registro Y (idioma assolutamente **tipico** per l'uso di puntatori definiti a runtime), che sarà anche la variabile di induzione del loop con un conteggio incrementale.

```
        LDY #0
Loop    LDA ($FB),Y
        STA WBUF       ; Registro buffer di uscita di un generico chip seriale
        NOP
        INY
        CPY #$10       ; Il buffer contiene 16 byte
        BNE Loop
```

L'istruzione `NOP` dopo la scrittura è esemplificativa di molte situazioni reali, nelle quali il timing dei chip di periferica impone un ciclo di attesa con una **durata minima** prima di effettuare la scrittura successiva. In altri casi, occorre invece rileggere un registro del chip e/o controllare uno o più flag prima di scrivere il byte successivo.

A proposito di limite del loop: cosa succede se continuiamo ad incrementare un registro a 8 bit come Y quando ha già raggiunto il massimo valore esprimibile, ossia `$FF`? Lo abbiamo già specificato ampiamente nella sezione descrittiva sulla ISA (4.3), e ne lasciamo la verifica come istruttivo esercizio per il lettore, eseguendo le istruzioni qui elencate con un monitor e verificando anche i contenuti del registro di stato P:

```
        LDY #$FF
        INY
```

Ricerca dell'elemento massimo in un array. Come esempio classico presentiamo la semplice ricerca dell'elemento massimo in un array non ordinato di byte. Anche qui, il registro X è utilizzato sia come variabile di induzione per il loop, sia come indice (in modalità assoluta indicizzata).

```
;**************************************
;** Ricerca elemento massimo in un array
;**************************************

        *=$C000

;**************************************
Start   LDA #0          ; Massimo in Accumulatore
        TAX             ; Azzera indice
Loop    CMP Array,X     ; Assoluto ind. X
        BCS Next        ; Se minore, salta
        LDA Array,X     ; Nuovo massimo temporaneo
Next    INX             ; Prossimo elemento
        CPX #12
        BNE Loop
Exit    RTS             ; A contiene il massimo
;**************************************
;** Array di byte:
Array   byte $79, $B3, $91, $32, $A0, $27
        byte $84, $55, $2B, $4E, $11, $CF
```

Il funzionamento del codice è assolutamente intuitivo: si scorre l'array dal primo all'ultimo elemento e si confronta l'elemento attuale col massimo (inizializzato correttamente a zero e mantenuto nell'Accumulatore). Se l'elemento attuale risulta maggiore, diventa il nuovo massimo temporaneo e si procede con gli ulteriori confronti. Al termine il valore in accumulatore sarà quindi il massimo assoluto dell'array. Proponiamo anche il listing dopo l'assemblaggio, per completezza.

```
Line    Addr ## Code        Source
-------------------------------------------------------------------------
00001   0000                ;**************************************
00002   0001                ;** RICERCA ELEMENTO MASSIMO IN UN ARRAY
00003   0001                ;**************************************
00004   0001
00005   0001                        *=$C000
00006   C000
00007   C000                ;**************************************
00008   C000 2  A9 00       START   LDA #0        ; MASSIMO IN ACCUMULATORE
00009   C002 2  AA                 TAX           ; AZZERA INDICE
00010   C003 4* DD 11 C0    LOOP    CMP ARRAY,X   ; ASSOLUTO IND. X
00011   C006 2* B0 03               BCS NEXT      ; SE MINORE, SALTA
00012   C008 4* BD 11 C0            LDA ARRAY,X   ; NUOVO MASSIMO TEMPORANEO
00013   C00B 2  E8          NEXT    INX           ; PROSSIMO ELEMENTO
00014   C00C 2  E0 0C               CPX #12
00015   C00E 2* D0 F3               BNE LOOP
00016   C010 6  60          EXIT    RTS           ; A CONTIENE IL MASSIMO
00017   C011                ;**************************************
00018   C011                ;** ARRAY DI BYTE:
00019   C011    79 B3 91    ARRAY   BYTE $79, $B3, $91, $32, $A0, $27
00020   C017    84 55 2B            BYTE $84, $55, $2B, $4E, $11, $CF
```

Le colonne del listing prodotto dall'Assembler indicano rispettivamente:

1. il numero di linea a cinque cifre;
2. l'indirizzo di memoria, con quattro cifre esadecimali;

3. i byte ivi assemblati, sempre in esadecimale (fino ad un massimo di tre);

4. la linea sorgente corrispondente.

Abbiamo aggiunto ad ogni listing la colonna ## dei tempi di esecuzione, indicati ovviamente in cicli, per avere un'idea del costo complessivo del codice a runtime (prassi aurea, da seguire regolarmente: molti ambienti di sviluppo nativi forniscono per default tale informazione nel listato dell'output assemblato).

Merging di due array ordinati. Un esempio immancabile in una pletora di testi di ogni livello e aspirazione è quello della copia di due array numerici (di byte) ordinati in un terzo array, mantenendo l'ordine iniziale (tipicamente ascendente: su questo si basano i confronti che costituiscono il vero punto centrale del banale algoritmo).

Questa implementazione *naïf* è normalmente tra i primi esempi di codice presentati, perché tutto sommato non fa altro che annidare una `IF...THEN...ELSE` all'interno di un loop, due esempi elementari che anche in questo contesto abbiamo mostrato poco sopra.

Il funzionamento è assolutamente banale. Dopo avere inizializzato opportunamente tutti gli indici, si confronta l'elemento `Array1[X]` con `Array2[Y]`. Il minore di essi viene copiato nella locazione corrente dell'array destinazione, il cui indice viene poi incrementato di una unità. Anche l'indice X o Y relativo all'elemento appena copiato viene incrementato, per poi ricominciare il ciclo dall'inizio fino al raggiungimento del totale prefissato di scritture, dato dalla somma delle lunghezze dei due array.

L'unica «difficoltà», a questo livello, consiste nel gestire tre indici distinti nel modo meno dispersivo possibile: i primi due sono utilizzati per la scansione indipendente degli array dati, il terzo indice è impiegato per le scritture sull'array destinazione. Per fare ciò, avendo a disposizione solo X e Y, utilizziamo in modo opportuno degli swap e due locazioni in pagina zero, una gestione nettamente più efficiente rispetto all'uso di stack o altre locazioni assolute.

Si noti che il registro X viene salvato subito prima di entrare nel loop, e ripristinato come prima operazione: questa apparente duplicazione del codice è in realtà una sorta di *unrolling* che semplifica la fase di controllo di permanenza nel loop, quando il contenuto del registro X viene alterato per fungere da indice destinazione e variabile di induzione con limite noto a priori. Il codice, in nome dell'estrema semplificazione e della massima linearità, fa infatti uso di una dimensione precalcolata per il numero totale di byte da scrivere. Allo stesso modo, il terzo indice non sarebbe strettamente necessario, come vederemo indirettamente al prossimo paragrafo. Ma poiché siamo appena all'inizio del nostro percorso, ci atteniamo strettamente alla regola universale K.I.S.S. (Keep It Simple, Stupid).

Il risultato finale, al termine dell'elaborazione, potrà essere facilmente verificato tramite monitor esaminando la locazione `$C048` e successive.

```
Line   Addr ## Code        Source
-----------------------------------------------------------------------
00001  0000               ;**************************************
00002  0001               ;** MERGE DI DUE ARRAY ORDINATI
00003  0001               ;**************************************
00004  0001
00005  0001                        *=$C000
00006  C000
00007  C000               ;**************************************
00008  C000               SAVE_X   = $FB
00009  C000               IDX      = $FC
00010  C000               SIZE     = 31   ; DIMENSIONE NOTA A PRIORI
00011  C000               ;**************************************
00012  C000               START
00013  C000 2  A2 00              LDX #0          ; AZZERA GLI INDICI
00014  C002 2  A0 00              LDY #0
00015  C004 3  86 FC              STX IDX         ; INDICE DESTINAZIONE
00016  C006 3  86 FB              STX SAVE_X      ; SALVA INDICE ARRAY1
00017  C008
00018  C008 3  A6 FB       LOOP   LDX SAVE_X      ; RIPRISTINA INDICE ARRAY1
00019  C00A 4* BD 29 C0           LDA ARRAY1,X    ; CONFRONTA ELEMENTI
00020  C00D 4* D9 3A C0           CMP ARRAY2,Y
00021  C010 2* 10 04              BPL NEXT        ; SE A1[X]>=A2[Y], SALTA
00022  C012
00023  C012 2  E8                INX             ; A2[Y]>A1[X]
```

```
00024   C013 3   4C 1A C0              JMP COPY
00025   C016
00026   C016 4*  B9 3A C0    NEXT      LDA ARRAY2,Y
00027   C019 2   C8                     INY
00028   C01A
00029   C01A                  ;** ORA IN A ABBIAMO IL MINORE DEI DUE VALORI
00030   C01A 3   86 FB        COPY      STX SAVE_X       ; SALVA INDICE ARRAY1
00031   C01C 3   A6 FC                  LDX IDX          ; INDICE DESTINAZIONE
00032   C01E 5   9D 48 C0               STA ARRAY,X
00033   C021 2   E8                     INX
00034   C022 3   86 FC                  STX IDX
00035   C024 2   E0 1F                  CPX #SIZE
00036   C026 2*  D0 E0                  BNE LOOP
00037   C028
00038   C028 6   60           EXIT      RTS
00039   C029                  ;***************************************
00040   C029                  ;** ARRAY NUMERICI DA FONDERE:
00041   C029     01 01 02     ARRAY1    BYTE 1, 1, 2, 3, 5, 7, 11, 15, 22
00042   C032     1E 2A 38                BYTE 30, 42, 56, 77, 101, 135, 176
00043   C039     E7                      BYTE 231
00044   C03A     00 01 01     ARRAY2    BYTE 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
00045   C044     37 59 90                BYTE 55, 89, 144, 233
00046   C048                  ;** ARRAY DESTINAZIONE:
00047   C048     00 00 00     ARRAY     BYTES 128, $FF
00048   C1C7                  ;***************************************
```

Vale la pena di notare incidentalmente che, tra le possibili scelte per la sequenza delle operazioni nella `IF...THEN...ELSE` implementata alle linee `00019-00027`, si è optato per una distribuzione che rende meno disastroso possibile il *worst case*, avendo sostanzialmente un costo di 7 cicli nel ramo più efficiente (`00021, 00026-00027`) e 9 nell'altro, il che sposta di poco la prestazione media anche in caso di grandi sbilanciamenti numerici tra i due array.

Si noti inoltre che, sempre in ottemperanza al principio K.I.S.S., in caso di uguaglianza `Array1[X] == Array2[Y]`, la priorità viene arbitrariamente data ad `Array2` e non al valore (identico) già contenuto in Accumulatore: ciò comporterà mediamente alcuni scambi non necessari, ma consente di risparmiare il costo di un ulteriore salto condizionato per ogni iterazione. Questi compromessi in Assembly sono la norma, specialmente nell'ambito qui discusso.

Copia efficiente di grandi blocchi di memoria. Per terminare presentiamo il listato di uno dei loop più sofisticati, che risulta anche **in assoluto il più efficiente** nella sua categoria: questo prevede un uso accorto delle capacità di auto-modifica del codice Assembly. In questo caso si tratta di un utilizzo del tutto lecito e controllato, che semplifica la gestione di indirizzi a 16 bit. Le costanti `_ORIGIN=$7000` e `_DEST=$A000` si assumono definite a monte.

```
Line    Addr ##  Code        Source
-------------------------------------------
00001   085C 2   A0 00                 LDY #$00
00002   085E 2   A2 C0                 LDX #$C0
00003   0860 5   B9 00 70    BSTART    LDA _ORIGIN,Y
00004   0863 5   99 00 A0    BDEST     STA _DEST,Y
00005   0866 2   C8                    INY
00006   0867 2*  D0 F7                 BNE BSTART
00007   0869 6   EE 62 08              INC BSTART+2
00008   086C 6   EE 65 08              INC BDEST+2
00009   086F 2   CA                    DEX
00010   0870 2*  D0 EE                 BNE BSTART
```

Il registro Y è azzerato, mentre il registro X contiene il numero totale di pagine da copiare. Al termine della copia della prima pagina, Y è nuovamente nullo (*wraparound*): a questo punto si modificano i byte alti degli indirizzi di origine e di destinazione *direttamente nelle istruzioni che li referenziano*! Ciò implica, ovviamente,

che il codice deve risiedere in RAM e non su memorie di sola lettura (dettaglio non trascurabile se parliamo di sistemi embedded basati su 6510). Alla linea 00007 infatti viene incrementato il byte posto alla locazione `BSTART+2=$0862`, vale a dire il byte inizialmente pari a `$70` nell'istruzione `LDA $7000,Y` che occupa i seguenti indirizzi:

0860	0861	0862
B9	00	70

Il byte all'indirizzo `$0862`, ossia la *pagina dell'indirizzo di partenza*, assumerà quindi in sequenza i valori `$70, $71, $72, ...` , trasformando di volta in volta l'istruzione in `LDA $7100,Y` poi `LDA $7200,Y` e così via per le pagine successive. Lo stesso si applica all'indirizzo di destinazione. Di fatto, il codice **modifica sé stesso** agendo direttamente sulle locazioni di memoria che contengono il programma. Come evidenziato nell'apposita colonna del costo in cicli, questo loop è in grado di movimentare una intera pagina di memoria tra due indirizzi assoluti con un costo complessivo pari a $256 \cdot 15 + 17 = 3.857$ cicli, vale a dire ad esempio su Commodore 64 NTSC (977,778 ns per ciclo) un tempo per pagina di circa 3,77 ms. Qualunque altra soluzione alternativa (necessariamente basata su indirizzamento indiretto) risulta misurabilmente **molto più costosa**. Si noti che i branch sono connotati da un costo pari a 2∗ cicli: occorre infatti aggiungere la penalità di un ciclo quanto il salto viene eseguito, e di un ulteriore ciclo nel caso in cui la destinazione risieda in una pagina di memoria diversa da quella del Program Counter corrispondente all'istruzione di branch stessa. Nel nostro calcolo abbiamo ovviamente considerato il caso favorevole, in cui l'intero codice del loop annidato risiede nella medesima pagina di memoria.

5.4.2 Esempi aritmetici di base.

Proseguiamo con gli esempi presentando alcune brevi **routine aritmetiche**, che peraltro andranno a colmare le lacune del set di istruzioni e dell'architettura[8].

I possibili risultati della somma di due byte ricadono nell'intervallo $[0, 510]$, il valore massimo è esprimibile con 9 bit in quanto banalmente $2^8 < 510 < 2^9$, in altri termini $\lceil \log_2 510 \rceil = 9$. Le possibili somme di due valori $a + b$ (e vale più in generale per tutte le operazioni algebriche *binarie*, ovvero applicate a due operandi) a 8 bit sono in totale $256^2 = (2^8)^2 = 2^{16} = 65.536$, ed è molto didattico divertirsi a costruire una *addition table* completa estendendo lo schema parziale seguente con uno spreadsheet, un ambiente di calcolo o al limite un qualsiasi linguaggio di programmazione di alto livello. In questo caso, poniamo in ciascuna cella $a_{r,c}$ il risultato della somma dei corrispondenti valori nella prima riga e prima colonna[9].

+	0	1	···	254	255
0	0	1	···	254	255
1	1	2	···	255	256
···	···	···	···	···	···
254	254	255	···	508	509
255	255	256	···	509	510

Nel caso delle operazioni **commutative** come somma e moltiplicazione, possiamo trascurare l'ordine degli addendi e limitarci a considerare le 32.896 coppie uniche, incluse quelle in cui $a = b$, considerando in sostanza la matrice sopra come triangolare inferiore o superiore. Anche in questa ottica, rimane il fatto che circa la metà (16.384, per l'esattezza) delle possibili somme $s = a + b$ forniscono un totale s superiore a 255 e quindi non rappresentabile con soli 8 bit.

Queste considerazioni, oltre a sottolineare la banale necessità di fare ricorso ad una word per contenere il risultato di una generica somma di due addendi a 8 bit, acquistano notevole rilevanza nel caso in cui sia indispensabile limitare uno o ambedue gli addendi per contenere tassativamente entro soli 8 bit il risultato, per i motivi più svariati, come avviene sovente nell'ambito del controllo embedded, del condizionamento di segnali, della trasmissione seriale etc.

[8] Si ricorda brevemente al lettore che i core a 8 bit concepiti un paio di decenni dopo le CPU di nostro interesse sono in prevalenza dotati di uno speciale stadio aritmetico a 16 bit, che (oltre alle banali somme algebriche) prima del volgere del millennio poteva già garantire lo svolgimento di una moltiplicazione intera 8x8 in due o tre cicli di clock, un risultato già assolutamente inavvicinabile per qualsiasi procedura in Assembly: prestazione portata poi in modo generalizzato, nel giro di un lustro o due, al traguardo attuale del singolo ciclo di clock.

[9] Il lettore noti che, con un semplice passaggio aggiuntivo, possiamo costruire la vera tabella additiva della classe di resto in modulo 2^8 e più in generale 2^n ponendo nelle celle la somma s $(mod\ 2^n)$. Tuttavia, in questo momento ci interessa maggiormente evidenziare che **circa metà delle possibili somme di due byte fornisce un risultato non esprimibile con 8 bit**.

Non volendo complicare la trattazione, si affidano all'intuito e all'osservazione del lettore le rilevanti proprietà e simmetrie della tabella in questione.

5.4.2.1 Somma a 8 bit.

Dovendo scrivere una generica routine Assembly per la somma di due byte, in base a quanto appena considerato sarà opportuno e necessario tenere conto del riporto (Carry) e fare uso di un totale a 16 bit, occupando di fatto un ulteriore byte di memoria per il risultato. Altro accorgimento di *programmazione difensiva*, arma vincente del programmatore in qualsiasi linguaggio e contesto operativo, è quello di assicurarsi che il carry sia azzerato prima dell'inizio della somma e che il microprocessore si trovi in modalità binaria. Ne consegue il listato seguente: si noti l'uso dei commenti *inline*, preceduti da un carattere ';', per rendere maggiormente ordinato e leggibile il sorgente.

```
;***************************************
;** Somma 8+8 bit con risultato a 2 byte
;***************************************

        *=$C000

;***************************************
Start   CLC         ; Azzera il carry
        CLD         ; Imposta il modo binario
        LDA OP1     ; Carica il primo addendo
        ADC OP2     ; Esegue la somma in A
        STA RES     ; Memorizza il risultato
        LDA #0      ; Addizione fittizia
        ADC #0      ; per tenere conto del carry
        STA RES+1   ; Aggiorna il byte alto
End     RTS         ; Fine lavoro
;***************************************
;** Variabili inizializzate:
OP1 byte $69
OP2 byte $A5
RES word 0
```

Il risultato, una volta assemblato con l'ambiente CPS, deve essere il seguente:

```
Line    Addr ## Code        Source
-------------------------------------------------------------------
00001   0000               ;***************************************
00002   0001               ;** SOMMA 8+8 BIT CON RISULTATO A 2 BYTE
00003   0001               ;***************************************
00004   0001
00005   0001                       *=$C000
00006   C000
00007   C000               ;***************************************
00008   C000 2  18         START    CLC         ; AZZERA IL CARRY
00009   C001 2  D8                  CLD         ; IMPOSTA IL MODO BINARIO
00010   C002 4  AD 13 C0            LDA OP1     ; CARICA IL PRIMO ADDENDO
00011   C005 4  6D 14 C0            ADC OP2     ; ESEGUE LA SOMMA IN A
00012   C008 4  8D 15 C0            STA RES     ; MEMORIZZA IL RISULTATO
00013   C00B 2  A9 00                LDA #0      ; ADDIZIONE FITTIZIA
00014   C00D 2  69 00                ADC #0      ; PER TENERE CONTO DEL CARRY
00015   C00F 4  8D 16 C0            STA RES+1   ; AGGIORNA IL BYTE ALTO
00016   C012 6  60         END      RTS         ; FINE LAVORO
00017   C013               ;***************************************
00018   C013               ;** VARIABILI INIZIALIZZATE:
00019   C013 2  69         OP1      BYTE $69
00020   C014 3  A5         OP2      BYTE $A5
00021   C015 7  00 00      RES      WORD 0
```

Si invita il lettore a porre attenzione ai numerosi dettagli evidenziati dal listing, in particolare:

- le **variabili inizializzate** poste in coda al codice (è lì che avverranno le letture e scritture);
- i **due byte** riservati dall'Assembler per il risultato `RES`, grazie all'uso della direttiva `WORD`;
- la **codifica** degli mnemonici con i differenti tipi di indirizzamento.

L'espressione semplice `RES+1` viene risolta dall'Assembler, come si vede nei tre byte di codifica, con l'indirizzo abbinato all'etichetta `RES` (che è `$C015`, come si evince chiaramente dall'ultima riga del listato) aumentato di una unità, ed è dunque pari a `$C016` i cui byte in ordine little endian costituiscono rispettivamente il secondo e il terzo byte nell'encoding dell'istruzione `STA RES+1`: `8D 16 C0`. Come si nota, si tratta di un approccio alla gestione delle variabili radicalmente diverso rispetto a qualsiasi linguaggio di alto livello!

5.4.2.2 Somma a 8 bit: una soluzione alternativa.

Tre considerazioni scaturiscono immediatamente dalla lettura del sorgente al punto precedente.

1. Si può ottimizzare un po' il codice facendo uso di locazioni in pagina zero?
2. L'addizione fittizia è un po' farraginosa. Esiste un'alternativa più razionale?
3. Alternativamente, si può sfruttare tale codice per eseguire una vera somma di addendi a 16 bit?

Per la 1. la risposta è «ovviamente sì», purché tali locazioni siano disponibili. Poiché la maggioranza dei lettori proverà il codice su una macchina reale o emulatore, è noto che lo spazio in pagina zero scarseggia in quanto risorsa preziosa per il programmatori di sistema Commodore e altri. Si lascia la stesura del codice modificato come utile esercizio per il lettore.

Per la 2. sappiamo che, al caso peggiore, il risultato della somma non potrà avere più di 9 bit. Quindi il byte alto di `RES` (memorizzato alla locazione `RES+1`, si ricordi) potrà in definitiva valere solamente 0 o 1. La nostra CPU, come la totalità delle altre in commercio, consente di eseguire un salto condizionato in base al valore del carry. Tutto ciò che occorre è quindi incrementare di una unità `RES+1` nel caso in cui il carry sia alto dopo la somma, dal momento che abbiamo saggiamente inizializzato a zero ambedue i byte di tale variabile. Presentiamo qui direttamente il listato dopo l'assemblaggio.

```
Line    Addr ## Code        Source
-------------------------------------------------------------------------
00001   0000                ;****************************************
00002   0001                ;** SOMMA 8+8 BIT MIGLIORATA
00003   0001                ;****************************************
00004   0001
00005   0001                        *=$C000
00006   C000
00007   C000                ;****************************************
00008   C000 2  18          START   CLC         ; AZZERA IL CARRY
00009   C001 2  D8                  CLD         ; IMPOSTA IL MODO BINARIO
00010   C002 4  AD 11 C0            LDA OP1     ; CARICA IL PRIMO ADDENDO, OP1
00011   C005 4  6D 12 C0            ADC OP2     ; ESEGUE LA SOMMA IN A CON OP2
00012   C008 4  8D 13 C0            STA RES     ; MEMORIZZA IL RISULTATO
00013   C00B 2* 90 03                BCC END     ; NIENTE CARRY? FINITO.
00014   C00D 6  EE 14 C0            INC RES+1   ; AGGIORNA IL BYTE ALTO
00015   C010 6  60          END     RTS         ; FINE LAVORO
00016   C011                ;****************************************
00017   C011                ;** VARIABILI INIZIALIZZATE:
00018   C011    69          OP1     BYTE $69
00019   C012    A5          OP2     BYTE $A5
00020   C013    00 00       RES     WORD 0
```

Il programma così ottenuto è più breve (di due byte) e mantiene la stessa robustezza, quindi non genera errori in caso di overflow del risultato rispetto agli 8 bit. Si noti, grazie alla colonna ## del costo in cicli, come variano i tempi di esecuzione, ricordando che il costo del branch `BCC` è pari a 3 cicli se il salto viene intrapreso (salirebbero a 4 se la destinazione non si trovasse nella stessa pagina di memoria).

L'istruzione `INC RES+1` potrebbe essere sostituita teoricamente da una coppia di istruzioni come `LDA #1` e `STA RES+1`, ma una veloce analisi del footprint e dei tempi di esecuzione (che viene lasciata come proficuo esercizio per il lettore) è sufficiente a dissuaderci dal modificare quanto già scritto. In estrema sintesi, la forma

mentis del programmatore Assembly è completamente incentrata su questi aspetti: dopo aver verificato a monte le caratteristiche dell'algoritmo scelto, in primis ottimalità come pure correttezza e robustezza, ci si deve poi concentrare sugli aspetti di micro-ottimizzazione, tra cui footprint, ordine di esecuzione e costo in cicli delle singole istruzioni.

5.4.2.3 Somma a 16 bit.

Riguardo al punto 3., le modifiche richieste per adattare il codice alla somma di addendi a 16 bit sono decisamente minime. Naturalmente anche in questo caso il risultato può eccedere i 16 bit, quindi occorre prevedere tre byte per il risultato: sulle normali architetture risulta impossibile frazionare l'accesso alla memoria[10], quindi anche un solo bit in più richiede una intera locazione, in questo caso un byte. La soluzione è data dal listato seguente:

```
;**************************************
;** Somma a 16 bit, risultato su 3 byte
;**************************************

        *=$C000

;**************************************
Start   CLC         ; Azzera il carry
        CLD         ; Imposta il modo binario
        LDA OP1     ; Carica in A il byte basso del primo addendo
        ADC OP2     ; Esegue la somma in A con il byte basso di OP2
        STA RES     ; Memorizza il risultato parziale
        LDA OP1+1   ; Ripete per il byte alto
        ADC OP2+1   ; includendo il carry
        STA RES+1   ;
        BCC End     ; No Carry? No party...
        INC RES+2   ; Aggiorna il terzo byte del risultato
End     RTS         ; Fine lavoro
;**************************************
;** Variabili inizializzate:
OP1 word $F069
OP2 word $AA55
RES word 0,0
```

Per riscontro, ecco il listing completo dopo l'assemblaggio. Come già ricordato, per motivi tipografici la lunghezza delle linee risulta limitata a 80 caratteri, con ovvie conseguenze sui commenti: per tale motivo si presenta separatamente anche il sorgente nella maggioranza degli esempi. Si noti come la somma dei byte alti includa anche il carry eventualmente generato dall'operazione precedente: il riporto è appunto stato progettato per questo tipo di **propagazione automatica**.

```
Line    Addr ## Code        Source
-------------------------------------------------------------------
00001   0000                ;**************************************
00002   0001                ;** SOMMA A 16 BIT, RISULTATO SU 3 BYTE
00003   0001                ;**************************************
00004   0001
00005   0001                        *=$C000
00006   C000
00007   C000                ;**************************************
00008   C000 2  18          START   CLC         ; AZZERA IL CARRY
00009   C001 2  D8                  CLD         ; IMPOSTA IL MODO BINARIO
00010   C002 4  AD 1A C0            LDA OP1     ; CARICA IN A IL BYTE BASSO DEL P
00011   C005 4  6D 1C C0            ADC OP2     ; ESEGUE LA SOMMA IN A CON IL BYT
00012   C008 4  8D 1E C0            STA RES     ; MEMORIZZA IL RISULTATO PARZIALE
00013   C00B 4  AD 1B C0            LDA OP1+1   ; RIPETE PER IL BYTE ALTO
```

[10] Per mera curiosità del lettore, si rammenta che su molti DSP e su alcune architetture di microcontroller avanzate è possibile configurare (spesso anche dinamicamente, a runtime) l'ampiezza di parola di determinate zone della memoria dati, creando così dei bit array (accessibili singolarmente per bit) o segmenti con accesso differenziato (8, 16, 32, 64, 80 bit e oltre).

```
00014   C00E 4   6D 1D C0            ADC OP2+1   ; INCLUDENDO IL CARRY
00015   C011 4   8D 1F C0            STA RES+1   ;
00016   C014 2*  90 03               BCC END     ; NO CARRY? NO PARTY...
00017   C016 6   EE 20 C0            INC RES+2   ; AGGIORNA IL TERZO BYTE DEL RISU
00018   C019 6   60          END     RTS         ; FINE LAVORO
00019   C01A                 ;*************************************
00020   C01A                 ;** VARIABILI INIZIALIZZATE:
00021   C01A     69 F0       OP1     WORD $F069
00022   C01C     55 AA       OP2     WORD $AA55
00023   C01E     00 00 00    RES     WORD 0,0
```

Subroutine modulare per somma a sedici bit. A questo punto è possibile proporre anche una soluzione leggermente più avanzata, che fa un uso più esteso delle caratteristiche dell'Assembler CPS e mostra tecniche di notevole importanza: l'inizializzazione dinamica di variabili multibyte e l'uso di una subroutine, elemento fondante della programmazione modulare e strutturata. Sono vari e importanti i sostanziali elementi di novità introdotti in questo sorgente, in particolare si sottolineano:

1. L'uso di costanti a 16 bit definite in testa al sorgente e poi caricate dinamicamente nelle locazioni di memoria `OP1, OP2` usando i tipici operatori di estrazione dei byte < e >;
2. Uso di una specifica subroutine per la somma, richiamata in più punti con diversi valori delle variabili, il che consente anche di verificarne con facilità il comportamento nei casi limite.

```
;*************************************
;** Somma a 16 bit, risultato su 3 byte
;** Versione modularizzata
;*************************************

        *=$C000

;*************************************
;** Costanti di esempio:
Add1 = $F069
Add2 = $55AA
Add3 = $0178
Add4 = $AA55
Add5 = $FFFF
Add6 = $FFFF

;*************************************
Start   CLD         ; Imposta il modo binario
;** Inizializzazione dinamica
;** RES = Add1 + Add2
        LDA #<Add1 ; Least Significant Byte
        STA OP1    ; Indirizzo più basso
        LDA #>Add1 ; Most Significant Byte
        STA OP1+1

        LDA #<Add2 ; Come sopra, per OP2
        STA OP2
        LDA #>Add2
        STA OP2+1
        JSR Add_16 ; Effettua la somma

;** RES = Add3 + Add4
        LDA #<Add3
        STA OP1
        LDA #>Add3
        STA OP1+1
```

```
        LDA #<Add4
        STA OP2
        LDA #>Add4
        STA OP2+1
        JSR Add_16

;** RES = Add5 + Add6
        LDA #<Add5
        STA OP1
        LDA #>Add5
        STA OP1+1

        LDA #<Add6
        STA OP2
        LDA #>Add6
        STA OP2+1
        JSR Add_16

        RTS         ; Fine lavoro
;****************************************

;****************************************
;** Subroutine di somma 16+16, con
;** risultato su 3 byte
;****************************************
Add_16  CLC         ; Azzera il Carry
        LDA #0
        STA RES+2   ; Azzera il terzo byte del risultato
        LDA OP1     ; Somma dei due byte bassi
        ADC OP2     ;
        STA RES     ; Memorizza il risultato parziale
        LDA OP1+1   ; Somma dei due byte alti
        ADC OP2+1   ;
        STA RES+1   ; Aggiorna il byte alto della somma
        BCC END     ; NO CARRY? NO PARTY...
        INC RES+2   ; Aggiorna il terzo byte del risultato
END     RTS         ; Fine subroutine

;****************************************
;** Variabili:
OP1     WORD 0      ; Primo addendo, 16 bit
OP2     WORD 0      ; Secondo addendo, 16 bit
RES     WORD 0,0    ; Risultato, 24 bit
```

Come si vede, il caricamento dei valori costanti definiti all'inizio del sorgente è facilitato dall'uso dei tipici operatori per l'estrazione dei byte alti e bassi, e rende evidente ancora una volta la sequenza di memorizzazione **little endian** gestita interamente a carico del programmatore. Si propone, per comodità del lettore, anche il listing completo prodotto dall'Assembler di CPS:

```
Line    Addr ## Code        Source
-------------------------------------------
00001   0000                ;****************************************
00002   0001                ;** SOMMA A 16 BIT, RISULTATO SU 3 BYTE
00003   0001                ;** VERSIONE MODULARIZZATA
00004   0001                ;****************************************
00005   0001
00006   0001                        *=$C000
00007   C000
00008   C000                ;****************************************
```

```
00009  C000                ;** COSTANTI DI ESEMPIO:
00010  C000                ADD1 = $F069
00011  C000                ADD2 = $55AA
00012  C000                ADD3 = $0178
00013  C000                ADD4 = $AA55
00014  C000                ADD5 = $FFFF
00015  C000                ADD6 = $FFFF
00016  C000
00017  C000                ;***************************************
00018  C000                START
00019  C000 2  D8                   CLD          ; IMPOSTA IL MODO BINARIO
00020  C001                ;** INIZIALIZZAZIONE DINAMICA
00021  C001                ;** RES = ADD1 + ADD2
00022  C001 2  A9 69                LDA #<ADD1   ; LEAST SIGNIFICANT BYTE
00023  C003 4  8D 65 C0             STA OP1      ; INDIRIZZO PIU' BASSO
00024  C006 2  A9 F0                LDA #>ADD1   ; MOST SIGNIFICANT BYTE
00025  C008 4  8D 66 C0             STA OP1+1
00026  C00B
00027  C00B 2  A9 AA                LDA #<ADD2   ; COME SOPRA, PER OP2
00028  C00D 4  8D 67 C0             STA OP2
00029  C010 2  A9 55                LDA #>ADD2
00030  C012 4  8D 68 C0             STA OP2+1
00031  C015 6  20 47 C0             JSR ADD_16   ; EFFETTUA LA SOMMA
00032  C018
00033  C018                ;** RES = ADD3 + ADD4
00034  C018 2  A9 78                LDA #<ADD3
00035  C01A 4  8D 65 C0             STA OP1
00036  C01D 2  A9 01                LDA #>ADD3
00037  C01F 4  8D 66 C0             STA OP1+1
00038  C022
00039  C022 2  A9 55                LDA #<ADD4
00040  C024 4  8D 67 C0             STA OP2
00041  C027 2  A9 AA                LDA #>ADD4
00042  C029 4  8D 68 C0             STA OP2+1
00043  C02C 6  20 47 C0             JSR ADD_16
00044  C02F
00045  C02F                ;** RES = ADD5 + ADD6
00046  C02F 2  A9 FF                LDA #<ADD5
00047  C031 4  8D 65 C0             STA OP1
00048  C034 2  A9 FF                LDA #>ADD5
00049  C036 4  8D 66 C0             STA OP1+1
00050  C039
00051  C039 2  A9 FF                LDA #<ADD6
00052  C03B 4  8D 67 C0             STA OP2
00053  C03E 2  A9 FF                LDA #>ADD6
00054  C040 4  8D 68 C0             STA OP2+1
00055  C043 6  20 47 C0             JSR ADD_16
00056  C046
00057  C046 6  60                   RTS          ; FINE LAVORO
00058  C047                ;***************************************
00059  C047
00060  C047                ;***************************************
00061  C047                ;** SUBROUTINE DI SOMMA 16+16, CON
00062  C047                ;** RISULTATO SU 3 BYTE
00063  C047                ;***************************************
00064  C047 2  18          ADD_16   CLC          ; AZZERA IL CARRY
00065  C048 2  A9 00                LDA #0
00066  C04A 4  8D 6B C0             STA RES+2    ; AZZERA IL TERZO BYTE DEL RISULT
00067  C04D 4  AD 65 C0             LDA OP1      ; SOMMA DEI DUE BYTE BASSI
00068  C050 4  6D 67 C0             ADC OP2      ;
00069  C053 4  8D 69 C0             STA RES      ; MEMORIZZA IL RISULTATO PARZIALE
```

```
00070   C056 4  AD 66 C0            LDA OP1+1   ; SOMMA DEI DUE BYTE ALTI
00071   C059 4  6D 68 C0            ADC OP2+1   ;
00072   C05C 4  8D 6A C0            STA RES+1   ; AGGIORNA IL BYTE ALTO DELLA SOM
00073   C05F 2* 90 03               BCC END     ; NO CARRY? NO PARTY...
00074   C061 6  EE 6B C0            INC RES+2   ; AGGIORNA IL TERZO BYTE DEL RISU
00075   C064 6  60          END     RTS         ; FINE SUBROUTINE
00076   C065
00077   C065                ;*************************************
00078   C065                ;** VARIABILI:
00079   C065    00 00       OP1     WORD 0      ; PRIMO ADDENDO, 16 BIT
00080   C067    00 00       OP2     WORD 0      ; SECONDO ADDENDO, 16 BIT
00081   C069    00 00 00    RES     WORD 0,0    ; RISULTATO, 24 BIT
```

Ovviamente quanto fin qui discusso ha valore generale puramente didattico ed esemplificativo. Vi sono numerose occasioni nel *real world* nelle quali il risultato di una simile operazione aritmetica *non può e non deve* in alcun caso eccedere i 16 bit: esempio banale, quando si sta calcolando un indirizzo di memoria, che per tutte le CPU della famiglia 65xx è limitato in hardware ad un massimo assoluto di 65.536 locazioni. In tal caso, la logica non cambia: rimane comunque responsabilità del programmatore intraprendere l'azione più opportuna in presenza di un carry dopo la somma dei due byte più significativi, usando le istruzioni proposte.

5.4.2.4 Sommatoria di un array di byte.

Totalizzare il contenuto di un array di byte è, a questo punto, un lavoretto banale: si tratta semplicemente di fondere due esempi già presentati. In linea di principio, un array gestito in Assembly potrebbe estendersi fino ai limiti della memoria installata, ma si tratta di un caso che qui non contempleremo, limitandoci al classico array delle dimensioni di una pagina di memoria. Nel caso peggiore l'array sarà quindi composto da 256 elementi (il massimo indirizzabile direttamente con uno dei registri indice), ciascuno contenente il valore massimo 255: la sommatoria sarà in tale caso limite pari a $256 \cdot 255 = 65.280 < 2^{16}$. Possiamo quindi tranquillamente dimensionare il nostro totalizzatore come una word.

```
;*************************************
;** Sommatoria degli elementi di un array
;*************************************

        *=$C000

;*************************************
;** Totalizzatore a 16 bit
Sum = $FB

;*************************************
Start   CLD             ; Modo decimale
        CLC             ; Carry = 0
        LDA #0
        TAY             ; Azzera l'indice
        TAX             ; Byte alto del totalizzatore
Loop    ADC Array,Y     ; Assoluto ind. Y
        BCC Next        ; Se c'e' carry, aggiorna X
        INX             ; Incrementa il byte alto
Next    INY             ; Prossimo elemento
        CPY #12         ; Lunghezza nota a priori
        BNE Loop
        STA Sum         ; Memorizza il byte basso
        STX Sum+1       ; ...e quello alto
Exit    RTS
;*************************************
;** Array di byte (somma = $04E8):
Array   byte $79, $B3, $91, $32, $A0, $27
        byte $84, $55, $2B, $4E, $11, $CF
```

Il funzionamento è del tutto intuitivo. Si parte con il totalizzatore inizializzato a zero e vi si aggiungono, uno alla volta, tutti gli elementi dell'array in ordine di visita per indice crescente. Per ovvi motivi prestazionali, il byte meno significativo del totalizzatore è mantenuto nell'Accumulatore, mentre il byte più significativo nel registro X. Questo conferisce al nostro codice la massima efficienza possibile, sfruttando un principio fondamentale nella codifica con ISA RISC: l'uso intensivo dei registri, sebbene con i drastici limiti imposti dall'architettura del 6502. L'ultima operazione trasferisce semplicemente il contenuto dei due registri nelle locazioni di pagina zero designate a contenere il totalizzatore, rendendolo così disponibile per ulteriori elaborazioni. Si noti che tali locazioni non vengono preventivamente azzerate, in quanto comunque sovrascritte al termine dell'elaborazione dal contenuto di A e X rispettivamente. Si propone di seguito anche il listato assemblato.

```
 Line    Addr ## Code         Source
-------------------------------------------------------------------------------
00001   0000                 ;**************************************
00002   0001                 ;** SOMMATORIA DEGLI ELEMENTI DI UN ARRAY
00003   0001                 ;**************************************
00004   0001
00005   0001                          *=$C000
00006   C000
00007   C000                 ;**************************************
00008   C000                 ;** TOTALIZZATORE A 16 BIT
00009   C000                 SUM      = $FB
00010   C000
00011   C000                 ;**************************************
00012   C000 2   D8          START    CLD             ; MODO DECIMALE
00013   C001 2   18                   CLC             ; CARRY = 0
00014   C002 2   A9 00                LDA #0
00015   C004 2   A8                   TAY             ; AZZERA L'INDICE
00016   C005 2   AA                   TAX             ; BYTE ALTO DEL TOTALIZZATORE
00017   C006 4*  79 16 C0    LOOP     ADC ARRAY,Y     ; ASSOLUTO IND. Y
00018   C009 2*  90 01                BCC NEXT        ; SE C'E' CARRY, AGGIORNA X
00019   C00B 2   E8                   INX             ; INCREMENTA IL BYTE ALTO
00020   C00C 2   C8          NEXT     INY             ; PROSSIMO ELEMENTO
00021   C00D 2   C0 0C                CPY #12
00022   C00F 2*  D0 F5                BNE LOOP
00023   C011 3   85 FB                STA SUM         ; MEMORIZZA IL BYTE BASSO
00024   C013 3   86 FC                STX SUM+1       ; ...E QUELLO ALTO
00025   C015 6   60          EXIT     RTS
00026   C016                 ;**************************************
00027   C016                 ;** ARRAY DI BYTE:
00028   C016     79 B3 91    ARRAY    BYTE $79, $B3, $91, $32, $A0, $27
00029   C01C     84 55 2B             BYTE $84, $55, $2B, $4E, $11, $CF
```

5.4.2.5 Somma segnata (complemento a due).

Qui è doveroso invitare il lettore a riflettere su un aspetto fondamentale. Alcuni dei numeri esadecimali utilizzati negli esempi corrispondono a valori *diversi* a seconda che si interpretino come numeri senza segno o in complemento a due (ciò è del tutto trasparente per la CPU). In particolare, `$F069` rappresenta sia 61.542 che -3.994 e allo stesso modo `$AA55` è la rappresentazione sia di 43.605 che di -21.931, ma la `ADC` (unica istruzione prevista indistintamente per valori *signed* e *unsigned*) deve fornire il risultato corretto **in qualsiasi caso**. Dunque, la prima somma deve esprimere sia $61.542 + 21.930 = 83.472$ che $-3.994 + 21.930 = 17.396$ ed in effetti questo è esattamente ciò che avviene, a meno del carry finale che - come ricordiamo - deve essere *ignorato* quando si sommano algebricamente valori in complemento a due. Il risultato, infatti, è sempre pari a `$4610` ma solo nel primo caso si dovrà tenere conto del carry per comporre l'effettivo valore a 17 bit, pari a `$14610`. Il lettore è invitato a sperimentare con vari valori per gli operandi, usando opportunamente un monitor LM (a partire da quello disponibile in VICE[11]).

[11] Risulta possibile, e molto comodo, lanciare direttamente il progetto corrente da CBM Prg Studio nell'emulatore VICE. Prima di avviare il codice prodotto, tipicamente con una `SYS 49152`, è sufficiente invocare il monitor dall'ambiente VICE e digitare nella relativa finestra `break $C000`. A questo punto si può tornare a VICE e lanciare il comando `SYS`: il breakpoint così impostato causerà l'apertura automatica della finestra del monitor. Si rimanda il lettore all'help in linea per qualsiasi ulteriore approfondimento.

Per stimolare ulteriori riflessioni nel lettore, presentiamo una tabella comparativa delle rappresentazioni binarie su 8 bit. Si sottolinea ancora una volta come il bit più significativo MSB rappresenta *il bit del segno* in complemento a due:

Binario	Esadecimale	Decimale	
		Naturale	Relativo
0000 0000	0	0	0
0000 0001	1	1	+1
...	...	...	...
0111 1111	7F	127	+127
1000 0000	80	128	-128
1000 0001	81	129	-127
...	...	...	...
1111 1110	FE	254	-2
1111 1111	FF	255	-1

5.4.2.6 Sottrazione a 16 bit.

Presentiamo un esempio di codice modulare per la sottrazione di valori a 16 bit. Le differenze rispetto al codice per la somma sono decisamente minime: in questo caso, il complemento del Carry indica il prestito (borrow). Per contro, l'eventuale Carry al termine della sottrazione non è significativo e lo sconfinamento viene in questo caso effettivamente gestito come overflow: tramite il flag V nel registro P.

Si noti che è stato predisposto un risultato a soli 16 bit: si lascia infatti come utile esercizio per il lettore il completamento del codice per la gestione del caso di overflow, usando opportunamente l'istruzione `BVC`.

Ovviamente valgono le medesime considerazioni fatte appena sopra riguardo al complemento a due e all'overflow (che qui sostituisce il carry finale), con una sostanziale differenza concettuale: la sottrazione $a - b = n$ in condizioni normali non ha chiusura algebrica[12] in $\mathbb{N}$ quando $a < b$. Dunque, ad esempio, il risultato della prima sottrazione `$2269-$7703` ovvero $8.809 - 30.467$ sarà necessariamente *negativo* e quindi significativo solo se interpretato come complemento a due, anche se in questo caso ambedue gli operandi rappresentano *sempre* numeri positivi - quale che sia la rappresentazione inizialmente prescelta. Il lettore è caldamente invitato a sperimentare e fare esercizi con valori numerici diversi, calcolando manualmente in base decimale i risultati attesi in rappresentazione non segnata e in complemento a due, ponendo ogni volta attenzione alla presenza ed eventuale significatività di overflow tramite esecuzione in ambiente LM Monitor.

```
Line    Addr ## Code        Source
-------------------------------------------------------------------
00001   0000                ;*****************************************
00002   0001                ;** SOTTRAZIONE A 16 BIT, RISULTATO SU
00003   0001                ;**  2 BYTE - VERSIONE MODULARIZZATA
00004   0001                ;*****************************************
00005   0001
00006   0001                          *=$C000
00007   C000
00008   C000                ;*****************************************
00009   C000                ;** COSTANTI DI ESEMPIO:
00010   C000                ADD1 = $2269
00011   C000                ADD2 = $7703
00012   C000                ADD3 = $8000
00013   C000                ADD4 = $7FFF
00014   C000
00015   C000                ;*****************************************
00016   C000                START
00017   C000 2   D8                   CLD         ; IMPOSTA IL MODO BINARIO
00018   C001                ;** INIZIALIZZAZIONE DINAMICA
00019   C001                ;** RES = ADD1 - ADD2
00020   C001 2   A9 69                LDA #<ADD1  ; LEAST SIGNIFICANT BYTE
00021   C003 4   8D 44 C0             STA OP1     ; INDIRIZZO PIU' BASSO
```

[12] Senza scomodare le definizioni algebriche di anello e semianello, ricordiamo dalle scuole dell'obbligo che $n \in \mathbb{N} \Leftrightarrow n \geq 0$, quindi sostituendo abbiamo $a - b \geq 0 \Leftrightarrow a \geq b$ affinché la loro differenza sia un numero naturale, ossia intero non negativo. Per garantire incondizionatamente la chiusura algebrica rispetto alla sottrazione, occorre passare agli interi relativi $\mathbb{Z}$.

```
00022  C006 2  A9 22        LDA #>ADD1 ; MOST SIGNIFICANT BYTE
00023  C008 4  8D 45 C0     STA OP1+1
00024  C00B
00025  C00B 2  A9 03        LDA #<ADD2 ; COME SOPRA, PER OP2
00026  C00D 4  8D 46 C0     STA OP2
00027  C010 2  A9 77        LDA #>ADD2
00028  C012 4  8D 47 C0     STA OP2+1
00029  C015 6  20 30 C0     JSR SUB_16 ; EFFETTUA LA PRIMA SOTTRAZIONE
00030  C018
00031  C018               ;** RES = ADD3 - ADD4
00032  C018 2  A9 00        LDA #<ADD3
00033  C01A 4  8D 44 C0     STA OP1
00034  C01D 2  A9 80        LDA #>ADD3
00035  C01F 4  8D 45 C0     STA OP1+1
00036  C022
00037  C022 2  A9 FF        LDA #<ADD4
00038  C024 4  8D 46 C0     STA OP2
00039  C027 2  A9 7F        LDA #>ADD4
00040  C029 4  8D 47 C0     STA OP2+1
00041  C02C 6  20 30 C0     JSR SUB_16 ; EFFETTUA LA SECONDA SOTTRAZIONE
00042  C02F
00043  C02F 6  60           RTS        ; FINE LAVORO
00044  C030
00045  C030               ;***************************************
00046  C030               ;** SUBROUTINE DI SOTTRAZIONE 16 BIT,
00047  C030               ;** CON RISULTATO SU 2 BYTE
00048  C030               ;***************************************
00049  C030 2  38        SUB_16   SEC        ; AZZERA IL PRESTITO (C=1)
00050  C031 4  AD 44 C0           LDA OP1    ; DIFFERENZA DEI DUE BYTE BASSI
00051  C034 4  ED 46 C0           SBC OP2    ;
00052  C037 4  8D 48 C0           STA RES    ; MEMORIZZA IL RISULTATO PARZIALE
00053  C03A 4  AD 45 C0           LDA OP1+1  ; DIFFERENZA DEI DUE BYTE ALTI
00054  C03D 4  ED 47 C0           SBC OP2+1  ;
00055  C040 4  8D 49 C0           STA RES+1  ; AGGIORNA IL BYTE ALTO DELLA SOMMA
00056  C043 6  60        END      RTS        ; FINE SUBROUTINE
00057  C044
00058  C044               ;***************************************
00059  C044               ;** VARIABILI:
00060  C044     00 00     OP1      WORD 0     ; MINUENDO, 16 BIT
00061  C046     00 00     OP2      WORD 0     ; SOTTRAENDO, 16 BIT
00062  C048     00 00     RES      WORD 0     ; DIFFERENZA, 16 BIT
```

5.4.2.7 Aritmetica decimale (BCD).

Il Binary Coded Decimal (BCD) è un metodo antico, semplice ed economico per l'esecuzione di calcoli a precisione arbitraria, utilizzato spesso in ambito gestionale come alternativa alle ben più sofisticate implementazioni di *floating point decimale*[13] previste dallo standard **IEEE 854-1987** [IEE87]. Si tratta inoltre di un formato che facilita grandemente la *comunicazione tra sottosistemi*: ad esempio è ancora fondamentale in ambito embedded per i protocolli inter-IC come la memorizzazione di formati data e ora in chip datario specializzati, o la comunicazione con unità display con logica a bordo, in quanto facilissimo da convertire in ASCII (come anche in PETSCII, nel nostro caso).

La tabella seguente illustra l'idea fondamentale alla base del Binary Coded Decimal: il *sottoutilizzo* della rappresentatività binaria. Infatti si usa una rappresentazione diretta di una cifra decimale sfruttando un nibble, realizzando così una *codifica ridondante*.

[13] Telegraficamente, tale formato non soffre di alcuno dei problemi di arrotondamento e troncamento (con relative conseguenze sulla stabilità dei metodi numerici) che invece affliggono il floating point *binario*, assai più diffuso perché di più economica implementazione anche in hardware (ad esempio sui comuni PC mainstream e su tutte le architetture simili), soprattutto in riferimento alla versione precedente dello standard: IEEE 754-1985. Ad oggi implementazioni di FP decimale sono presenti come software in numerose librerie commerciali e compilatori, ma sono realizzate in hardware solo su processori di fascia altissima come gli IBM z9-z15 per il supercalcolo, vere e proprie astronavi al confronto degli storici precursori a 8 bit di cui trattiamo qui.

Decimale	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Non vengono utilizzati gli altri 6 possibili codici binari. Questo implica un fattore di sottoutilizzo molto elevato: nel BCD «unpacked» si usano solo 4 bit su 8, lasciando inutilizzati ben 246 possibili codici (praticamente il 96%). Nel packed BCD abbiamo invece due cifre decimali rappresentate dai due nibbles di un byte, che quindi può contenere valori compresi tra 0 e 99: il che esclude dall'utilizzo 156 codici, pari al 61% circa. In ambedue i casi abbiamo un'ampia ridondanza.

Le tabelle seguenti riassumono tutti i 100 codici BCD validi per un byte: le colonne rappresentano il **nibble basso**. Se cerchiamo la codifica del decimale 62 la troveremo a **riga 6, colonna 2**, e così via.

	0	1	2	3	4
0	0000 0000	0000 0001	0000 0010	0000 0011	0000 0100
1	0001 0000	0001 0001	0001 0010	0001 0011	0001 0100
2	0010 0000	0010 0001	0010 0010	0010 0011	0010 0100
3	0011 0000	0011 0001	0011 0010	0011 0011	0011 0100
4	0100 0000	0100 0001	0100 0010	0100 0011	0100 0100
5	0101 0000	0101 0001	0101 0010	0101 0011	0101 0100
6	0110 0000	0110 0001	0110 0010	0110 0011	0110 0100
7	0111 0000	0111 0001	0111 0010	0111 0011	0111 0100
8	1000 0000	1000 0001	1000 0010	1000 0011	1000 0100
9	1001 0000	1001 0001	1001 0010	1001 0011	1001 0100

	5	6	7	8	9
0	0000 0101	0000 0110	0000 0111	0000 1000	0000 1001
1	0001 0101	0001 0110	0001 0111	0001 1000	0001 1001
2	0010 0101	0010 0110	0010 0111	0010 1000	0010 1001
3	0011 0101	0011 0110	0011 0111	0011 1000	0011 1001
4	0100 0101	0100 0110	0100 0111	0100 1000	0100 1001
5	0101 0101	0101 0110	0101 0111	0101 1000	0101 1001
6	0110 0101	0110 0110	0110 0111	0110 1000	0110 1001
7	0111 0101	0111 0110	0111 0111	0111 1000	0111 1001
8	1000 0101	1000 0110	1000 0111	1000 1000	1000 1001
9	1001 0101	1001 0110	1001 0111	1001 1000	1001 1001

La caratteristica fondamentale della codifica BCD, immediata ma non banale, diventa assolutamente evidente se esprimiamo *in esadecimale* un qualsiasi valore della tabella appena presentata.

Decimale	Binario	Esadecimale
25	0010 0101	25
39	0011 1001	39
61	0110 0001	61
84	1000 0100	84

Potremmo continuare per tutti i valori possibili, ma è ovvio che otterremmo sempre il medesimo risultato. *I valori BCD espressi in esadecimale usano esattamente le medesime cifre, nel medesimo ordine, della loro rappresentazione decimale*! Ciò risulta estremamente comodo per il programmatore e per il software, anche se ovviamente introduce una ulteriore possibilità nell'interpretare i valori contenuti in un registro o in una qualsiasi cella di memoria. Ora possiamo infatti riprendere e completare la tabella già vista al paragrafo (5.4.2.5 a pagina 90):

Binario	Esadecimale	Packed BCD	Decimale	
			Naturale	Relativo
0000 0000	0	0	0	0
0000 0001	1	1	1	+1
...	...	...	...	...
0000 1001	9	9	9	+9
0000 1010	A	Proibito	10	+10
...	...	...	...	...
0111 1110	7E	Proibito	126	+126
0111 1111	7F	Proibito	127	+127
1000 0000	80	80	128	-128
1000 0001	81	81	129	-127
...	...	...	...	...
1001 1000	98	98	152	-104
1001 1001	99	99	153	-103
1001 1010	9A	Proibito	154	-102
...	...	...	...	...
1111 1110	FE	Proibito	254	-2
1111 1111	FF	Proibito	255	-1

Ovviamente ricade interamente sul programmatore la responsabilità dell'interpretazione dei vari possibili valori associati alla *medesima combinazione di bit*. Le CPU 65xx supportano il BCD con l'apposito flag D nel registro di stato P: quando è attivo, tale flag modifica il comportamento delle istruzioni ADC e SBC. Quando si sommano valori BCD, infatti, occorre impostare il carry al superamento del valore soglia `$99` e non `$FF`[14]; allo stesso modo, in caso di sottrazione il carry segnala se si tenta di sottrarre un numero più grande da uno minore. Le operazioni segnate in BCD sono però un po' più complesse rispetto all'uso del complemento a due, tanto che la via più pratica suggerita universalmente dalla letteratura generalista e suffragata dalla prassi di codifica consiste solitamente nell'eseguire le operazioni segnate in binario e successivamente convertire in BCD.

Somma 8+8 bit BCD. Il seguente frammento di codice esegue una somma BCD: si ricordi di impostare coerentemente i valori per le variabili da sommare, rispettando la tabella precedente. Il risultato dell'esempio dovrà essere $69 + 25 = 94$, non già `$69+$25=$8E` come in modo binario!

```
;****************************************
;** Somma di due valori packed BCD
;****************************************

        *=$C000

;****************************************
Start   CLC             ; Azzera il carry
        SED             ; Imposta il modo decimale BCD
        LDA OP1         ; Carica in A il primo addendo, OP1
        ADC OP2         ; Esegue la somma in A con OP2
        STA RES         ; Memorizza il risultato
;** La gestione del carry è lasciata come facile esercizio per il lettore
        CLD             ; Reimposta il modo binario
End     RTS             ; Fine lavoro
;****************************************
```

[14] In particolare, la *normalizzazione* di un valore packed BCD si esegue sommando la costante 6 al valore eventualmente eccedente il 9, in modo da riportare nel range consentito le due cifre generando un riporto e azzerando la cifra corrente.

```
;** Variabili inizializzate
OP1 byte $69
OP2 byte $25
RES word 0
```

Conversione da BCD a PETSCII. Mostriamo ora con quale facilità sia possibile convertire in PETSCII delle cifre BCD packed, che d'altro canto è il caso relativamente più complesso (rispetto all'unpacked). L'esempio si riferisce ad una fittizia lettura in formato packed BCD da un generico chip datario che fornisce anno, mese, giorno, ora e minuti, che supponiamo caricati a monte in quest'ordine nelle locazioni del byte array `RTC_data`. Si noti tra l'altro come viene utilizzato il registro X per salvare temporaneamente il valore (siamo costretti a modificare tale valore in A per isolare il nibble più significativo) e ripristinarlo al minor costo possibile[15]. Il codice seguente funzionerà come presentato su tutti i CBM/PET, per altre architetture sarà di norma necessario modificare il vettore per la routine Kernal `CHROUT`. Si noti che, per caricare i byte BCD dell'array, viene in questo caso utilizzato l'indirizzamento assoluto indicizzato Y: al solito, nella colonna del timing, un asterisco a seguito del costo in cicli indica che sono possibili penalità in caso di salto pagina, e per i branch indica inoltre la penalità aggiuntiva di un ciclo necessaria quando il salto condizionato viene effettivamente intrapreso.

Si noti inoltre che le CPU 65xx non dispongono di istruzioni di shift per un numero arbitrario di posizioni in una singola operazione, né della potente istruzione `SWAP` che scambia i due nibble di un registro: occorre quindi iterare quattro volte la `LSR A`, con un costo complessivo di ben 8 cicli macchina.

```
;****************************************
;** Conversione packed BCD->PETSCII
;****************************************

        *=$C000

;****************************************
;** Routine KERNAL per stampa carattere
CHROUT = $FFD2

;****************************************
Start   LDY #0      ; Ripete per 8 bit
Loop    LDA RTC_data,Y
        TAX         ; Salva temporaneamente il valore
        LSR A       ; Isola il nibble alto
        LSR A
        LSR A
        LSR A
        ORA #'0'    ; Converte in PETSCII
        JSR CHROUT  ; Visualizza cifra

        TXA         ; Ripristina efficientemente il valore
        AND #$0F    ; Isola il nibble basso
        ORA #'0'    ; Converte in PETSCII
        JSR CHROUT

        INY
        CPY #$5
        BNE Loop
End     RTS
```

[15] La soluzione con uso del registro X (o Y), quando non necessario per altri scopi, occupa 2 byte ed ha un costo di 2 cicli per il salvataggio con `TAX` (risp. `TAY`) e altri 2 per il ripristino con `TXA` (risp. `TYA`). L'altro caso più favorevole, a parità di footprint in memoria, sarebbe quello in cui l'array risiede in pagina zero, dove una lettura indicizzata X avrebbe ugualmente un costo di 4 cicli: tuttavia lo spazio in pagina zero è notoriamente limitato sugli home.

Ricaricare il valore in A tramite una seconda lettura dall'indirizzo assoluto indicizzato Y avrebbe un costo pari a 4 o 5 cicli (la penalità si applica quando l'indirizzo appartiene ad una pagina diversa rispetto al PC corrispondente all'istruzione `LDA addr16,Y`) e soprattutto un footprint di 3 byte. L'uso dello stack, infine, avrebbe un costo complessivo di ben 7 cicli (3 per `PHA` e 4 per `PLA`) con un footprint ancora pari a 2 byte ed è quindi la soluzione più svantaggiosa, da usare solamente quando non vi è possibilità di fare diversamente. Si sottolinea nuovamente come buona parte del lavoro del programmatore Assembly consista proprio nel valutare razionalmente le forme alternative secondo il target specifico da perseguire (velocità, minimizzazione del footprint, disponibilità di memoria privilegiata...).

```
;***************************************
;** Dati inizializzati di esempio:
RTC_data        byte $20, $2, $2, $20, $20
```

Si noti come la conversione ASCII avviene semplicemente aggiungendo una costante, in particolare il codice corrispondente a '0', ossia $30. Tale addizione avviene in realtà con un OR logico, in quanto (in questo caso) le due operazioni sono equivalenti e non siamo interessati a sommare il riporto, come invece in altri casi analoghi di conversione. Riportiamo di seguito il listing completo:

```
Line   Addr ## Code         Source
-------------------------------------------------------------------------------
00001  0000                 ;***************************************
00002  0001                 ;** CONVERSIONE PACKED BCD->PETSCII
00003  0001                 ;***************************************
00004  0001
00005  0001                          *=$C000
00006  C000
00007  C000                 ;***************************************
00008  C000                 ;** ROUTINE KERNAL PER STAMPA CARATTERE
00009  C000                 CHROUT      = $FFD2
00010  C000
00011  C000                 ;***************************************
00012  C000 2  A0 00        START       LDY #0       ; RIPETE PER 8 BIT
00013  C002 4* B9 1D C0     LOOP        LDA RTC_DATA,Y
00014  C005 2  AA                      TAX          ; SALVA TEMPORANEAMENTE IL VALORE
00015  C006 2  4A                      LSR A        ; ISOLA IL NIBBLE ALTO
00016  C007 2  4A                      LSR A
00017  C008 2  4A                      LSR A
00018  C009 2  4A                      LSR A
00019  C00A 2  09 30                   ORA #'0'     ; CONVERTE IN PETSCII
00020  C00C 6  20 D2 FF                JSR CHROUT   ; VISUALIZZA CIFRA
00021  C00F
00022  C00F 2  8A                      TXA          ; RIPRISTINA EFFICIENTEMENTE IL V
00023  C010 2  29 0F                   AND #$0F     ; ISOLA IL NIBBLE BASSO
00024  C012 2  09 30                   ORA #'0'     ; CONVERTE IN PETSCII
00025  C014 6  20 D2 FF                JSR CHROUT
00026  C017
00027  C017 2  C8                      INY
00028  C018 2  C0 05                   CPY #$5
00029  C01A 2* D0 E6                   BNE LOOP
00030  C01C 6  60           END         RTS
00031  C01D                 ;***************************************
00032  C01D                 ;** DATI INIZIALIZZATI
00033  C01D     20 02 02    RTC_DATA    BYTE $20, $2, $2, $20, $20
```

Riportiamo qui il segmento della tabella PETSCII di nostro interesse:

Dec	Hex	Simbolo
48	30	'0'
49	31	'1'
50	32	'2'
51	33	'3'
52	34	'4'
53	35	'5'
54	36	'6'
55	37	'7'
56	38	'8'
57	39	'9'

Conversione da BCD a decimale e viceversa. Data la natura introduttiva del testo, chiudiamo con un singolo esempio delle versioni più semplici dei classici algoritmi di conversione da BCD a decimale e viceversa[16]: si rimanda alla bibliografia per eventuali approfondimenti.

```
;**************************************
;** Esempi di conversione tra decimale e
;** BCD packed
;**************************************

     *=$C000

;**************************************
BinVal = $FB          ; Byte binario da convertire
BCDL   = $FC          ; Cifra BCD meno significativa
BCDH   = $FD          ; Cifra BCD più significativa

;**************************************
Start   LDA #157      ; Inizializza la variabile da convertire
        STA BinVal
        JSR Bin2BCD   ; Converte da binario a BCD

        LDA #$74
        STA BCDL
        JSR BCD2Bin   ; Converte da due cifre packed BCD a binario

        RTS
;**************************************

;**************************************
;** Subroutine di conversione 8 bit
;** binario->BCD
;**************************************
Bin2BCD LDA #0        ; Azzera le variabili di output
        STA BCDL
        STA BCDH
        SED           ; Imposta il flag Decimale
        LDY #8        ; Ripete per gli 8 bit
Loop    ASL BinVal    ; MSB->Carry
        LDA BCDL      ; BCDL = BCDL + BCDL + Carry, in Decimal Mode
        ADC BCDL
        STA BCDL
        LDA BCDH      ; Ripete per il byte alto
        ADC BCDH
        STA BCDH
        DEY
        BNE Loop

        CLD
        RTS

;**************************************
;** Subroutine di conversione 8 bit
;** BCD->binario
;**************************************
BCD2Bin LDA #$80
        STA BinVal    ; Imposta a 1 il MSB per la terminazione

UP      LSR BCDL      ; Divide per 2, LSB->Carry
```

[16] L'esempio è sostanzialmente basato su due routine presentate in [Jon84], pagg. 129-130, a loro volta basate sui tradizionali algoritmi descritti in [Knu73, Pea77].

```
        ROR BinVal   ; Carry->MSB di BinVal
        BCS Exit     ; Terminatore raggiunto dopo 8 cicli

        LDA BCDL     ; Corregge i valori non consentiti
        AND #$8      ; Bit 3 alto?
        BEQ UP       ; No, continua

FIX     LDA BCDL     ; Sì, corregge
        SEC
        SBC #3
        STA BCDL
        BCS UP

Exit    RTS
;*****************************************
```

5.4.3 Esempi aritmetici più avanzati.

Le CPU di cui discutiamo, e praticamente tutte quelle della medesima generazione, non dispongono di uno *stadio moltiplicatore*. Di conseguenza, praticamente tutti i sistemi progettati per l'uso domestico e office forniscono le necessarie routine sotto forma di firmware. Il maggiore svantaggio di tali routine è la loro genericità: essendo pensate per gestire la totalità dei casi, incluso il calcolo in virgola mobile, sono lente e ingombranti. Ma in realtà la quasi totalità dei calcoli può essere effettuata con numeri interi, o al limite in virgola fissa (*fixed point*, che è comunque un formato intero), e vi sono situazioni nelle quali è necessario saper implementare una moltiplicazione senza richiamare le routine del firmware.

5.4.3.1 Moltiplicazione (e divisione) per una costante.

Il caso più semplice è quello in cui si debba moltiplicare in modo *hardcoded* un valore a 8 bit per una (piccola) costante.

Come abbiamo a più riprese ricordato, il valore di un numero binario non è altro che la somma di un piccolo numero di potenze del due: ad esempio, $00011010b = 2^1 + 2^3 + 2^4$. Risulta un esercizio piacevole e divertente per la sua grande semplicità verificare la distribuzione dei 256 possibili valori a 8 bit in funzione del numero di potenze del due necessarie ad esprimerne il valore, ossia del numero complessivo di bit alti che li caratterizzano.

Il numero totale di valori esprimibili con n bit, come sappiamo, è dato dal numero di *disposizioni con ripetizioni* dei due simboli binari 0 e 1, ossia $DR(2,n) = 2^n$. Ma quanti sono tra questi i valori a n bit contenenti esattamente k bit pari a 1, con $0 \leq k \leq n$? La risposta è data dal calcolo delle *permutazioni con ripetizioni*, espresso dal *coefficiente multinomiale*. Ne ricordiamo brevemente la definizione generale: si abbiano m interi non negativi $k_1, \ldots, k_m$ (le *occorrenze* degli elementi considerati nell'insieme di base), non necessariamente distinti, con $m > 1$ e tali che $k_1 + ... + k_m = \sum_{j=1}^{m} k_m = n$. Vale quindi:

$$\binom{k_1 + k_2 + \cdots + k_m}{k_1, k_2, \ldots, k_m} \stackrel{def}{=} \frac{(k_1 + k_2 + \cdots + k_m)!}{k_1! k_2! \ldots k_m!} = \frac{n!}{\prod_{j=1}^{m} k_j!} \tag{5.4.1}$$

Mnemonicamente, il coefficiente multinomiale non è che il rapporto tra il *fattoriale della somma* e il *prodotto dei fattoriali* delle occorrenze $k_1 ... k_m$. Nel nostro caso l'insieme di base consiste solamente dei due simboli binari, quindi $m = 2$ e le relative occorrenze dello zero k_0 e dell'uno k_1 (usando opportunamente dei pedici mnemonici) sono *complementari* rispetto all'ampiezza della parola binaria considerata, essendo in particolare $k_0 + k_1 = n \Rightarrow k_0 = n - k_1$, il che in ultima analisi riduce la formula a:

$$\binom{k_0 + k_1}{k_0, k_1} = \frac{n!}{(n - k_1)! k_1!} = \binom{n}{k_1} \tag{5.4.2}$$

Pertanto, per qualsiasi numero k_1 di bit alti considerato tra 0 ed n, il relativo numero di valori espressi è dato semplicemente dal coefficiente binomiale della 5.4.2. Ad esempio, i valori a 8 bit contenenti esattamente 3 bit alti sono in totale $\binom{8}{3} = 56$ ed è banale costruire la seguente tabella, in funzione di k_1:

Bit alti	Valori	Esempi
0	1	00000000
1	8	$00000001, 00000010, \ldots, 10000000$
2	28	$00000011, 00000101, \ldots, 11000000$
3	56	$00000111, 00001011, \ldots, 11100000$
4	70	$00001111, 00010111, \ldots, 11110000$
5	56	$00011111, 00101111, \ldots, 11111000$
6	28	$00111111, 01011111, \ldots, 11111100$
7	8	$01111111, 10111111, \ldots, 11111110$
8	1	11111111
Totale	256	

Osservando le righe con lo sfondo evidenziato in grigio, che corrispondono alla maggioranza dei casi ($\frac{182}{256} = 0,7109$ ossia circa il 71%) si deduce che occorre sommare *da tre a cinque potenze del due* per esprimere un valore costante ampio un byte: ciò ha importanti conseguenze sul *numero medio di istruzioni* richiesto dall'esecuzione di moltiplicazioni e divisioni tramite *shift e somme*.

La formula Regina della Matematica Discreta. La tabella delle occorrenze dei bit alti evidenzia anche un altro aspetto: la sommatoria delle espressioni 5.4.2, per k_1 che varia tra 0 e 8 compresi, è pari a 2^8. Risulta immediato dimostrare che vale la generalizzazione di tale risultato:

$$2^n = \sum_{k=0}^{n} \binom{n}{k} \tag{5.4.3}$$

È sufficiente applicare la formula del binomio di Newton (o teorema binomiale), che qui ricordiamo senza dimostrarla:

$$(a+b)^n = \sum_{j=0}^{n} \binom{n}{j} a^{n-j} b^j \tag{5.4.4}$$

Si ha quindi banalmente:

$$2^n = (1+1)^n = \sum_{k=0}^{n} \binom{n}{k} 1^{n-k} 1^k = \sum_{k=0}^{n} \binom{n}{k} \qquad \text{QED}$$

La vasta maggioranza dei matematici discreti e computazionali, come riportato anche da Donald E. Knuth [Knu97], concorda nel ritenere la 5.4.3 la formula più bella e significativa della combinatoria.

Il valore massimo assunto dalla moltiplicazione di due byte può essere espresso con 16 bit, in quanto si ha al caso peggiore $255 \cdot 255 = 65.025 < 2^{16}$: in generale, moltiplicando due parole binarie da n bit il risultato sarà sicuramente esprimibile con $2n$ bit, e la moltiplicazione di $n \times m$ bit, con $n \neq m$, richiede un totale a $m+n$ bit.

In quest'area, l'idioma Assembly probabilmente più semplice è quello che implementa il **raddoppio** di un valore a 16 bit, in questo caso residente ad un indirizzo assoluto di memoria. L'unico accorgimento è quello di usare una rotazione per il byte più significativo, in modo da fare «entrare» il carry della precedente operazione di shift come LSB. Ovviamente il codice che implementa la divisione per due è *specularmente identico*. Rimane ovviamente a carico del programmatore gestire ogni eventuale overflow: anche un semplice raddoppio, se il valore di partenza è superiore a 32.768, può essere fonte di problemi.

```
        *=$C000

Start   ASL OP1         ; Shift a sinistra, MSB->Carry
        ROL OP1+1       ; Rotazione a sinistra, Carry->LSB
        LSR OP2         ; Come sopra, ma divide per due
        ROR OP2+1
End     RTS             ; Fine lavoro

OP1 word $255
OP2 word $ACF0
```

Mostriamo ora, per chiudere l'argomento, come moltiplicare un valore a 8 bit per la costante $9 = 2^0 + 2^3$. Il risultato avrà 16 bit, ampiamente sufficienti a contenere agevolmente i 12 bit del massimo risultato possibile, pari a $255 \cdot 9 = 2.295 = 1000\ 1111\ 0111b$. Si noti come i valori caricati in `OP1` siano scelti per poter testare esaustivamente ambedue i rami dell'albero di esecuzione: generando risultati che rispettivamente non richiedono e richiedono il salto relativo gestito da `BCC Exit`. Quando la densità di potenze del due nella costante moltiplicativa (o nel divisore per il caso simmetrico) aumenta, è ovviamente opportuno fare uso di loop[17].

```
Line   Addr ## Code        Source
-------------------------------------------------------
00001  0000               ;**************************************
00002  0001               ;** MOLTIPLICAZIONE PER UNA COSTANTE
00003  0001               ;** TRAMITE SHIFT E SOMME
00004  0001               ;**************************************
00005  0001
00006  0001                        *=$C000
00007  C000
00008  C000               ;** MOLTIPLICANDO IN PAGINA ZERO, 8 BIT
00009  C000               OP1 = $FB
00010  C000
00011  C000               ;**************************************
00012  C000 2  A9 1A      START   LDA #$1A    ; RISULTATO A 8 BIT
00013  C002 3  85 FB              STA OP1
00014  C004 6  20 16 C0          JSR MULTBY9
00015  C007
00016  C007 2  A9 61              LDA #$61    ; OVERFLOW -> 9 BIT
00017  C009 3  85 FB              STA OP1
00018  C00B 6  20 16 C0          JSR MULTBY9
00019  C00E
00020  C00E 2  A9 FF              LDA #$FF    ; VALORE LIMITE
00021  C010 3  85 FB              STA OP1
00022  C012 6  20 16 C0          JSR MULTBY9
00023  C015
00024  C015 6  60          END     RTS
00025  C016               ;**************************************
00026  C016
00027  C016               ;**************************************
00028  C016               ;** SUBROUTINE DI MOLTIPLICAZIONE PER 9
00029  C016               ;**************************************
00030  C016 2  A9 00      MULTBY9 LDA #$0     ; AZZERA IL BYTE ALTO
00031  C018 4  8D 36 C0          STA RES+1   ; DEL RISULTATO
00032  C01B 3  A5 FB              LDA OP1
00033  C01D 2  0A                 ASL A       ; MOLTIPLICA PER 8=2^3
00034  C01E 6  2E 36 C0          ROL RES+1   ;
00035  C021 2  0A                 ASL A       ;
00036  C022 6  2E 36 C0          ROL RES+1   ;
00037  C025 2  0A                 ASL A       ;
00038  C026 6  2E 36 C0          ROL RES+1   ;
00039  C029 2  18                 CLC         ; SOMMA IL VALORE INIZIALE
00040  C02A 3  65 FB              ADC OP1
00041  C02C 4  8D 35 C0          STA RES     ; RES = OP1*8 + OP1
00042  C02F 2* 90 03              BCC EXIT    ; NO CARRY? ESCE
00043  C031 6  EE 36 C0          INC RES+1   ; AGGIORNA RES+1
00044  C034 6  60          EXIT    RTS         ; FINE LAVORO
00045  C035               ;**************************************
```

[17] Vale la pena di sottolineare nuovamente come nei sistemi embedded la prassi del *codesign* risulti fondamentale ai fini prestazionali e di pulizia del codice. In una vasta maggioranza di casi, si può pensare l'hardware in funzione della semplificazione di talune costanti moltiplicative utilizzate per la conversione di segnali digitalizzati da sensori e periferiche, ad esempio agendo opportunamente su un fattore di amplificazione in un front-end analogico o scegliendo riferimenti in tensione adeguati per i convertitori ADC, poiché queste operazioni aritmetiche sono legate al sample rate e in molti casi devono essere iterate molte migliaia di volte al secondo, rendendone critica l'efficienza. In altri casi è comunque possibile riscrivere le equazioni in modo da favorire l'uso di costanti espresse da singole potenze del due.

```
00046   C035                    ;** VARIABILE RISULTATO, 2 BYTE
00047   C035    00  00          RES         WORD 0
```

5.4.3.2 Moltiplicazione (e divisione) 8x8 bit.

Le moltiplicazione di due byte in binario è relativamente semplice. I concetti dell'aritmetica decimale di Peano, appresi alle scuole elementari, rimangono pressoché invariati e sono piuttosto semplici da implementare tramite operazioni atomiche elementari: *somme e scorrimenti*. Ancora una volta, la strutturale semplicità del sistema binario semplifica in modo drastico lo scenario, consentendo una implementazione computazionale ragionevolmente semplice e lineare. La tabella di verità seguente mostra il funzionamento della moltiplicazione tra coppie di bit:

#	A	B	$A \cdot B$
0	0	0	0
1	0	1	0
2	1	0	0
3	1	1	1

Si può notare che trattasi sostanzialmente di una funzione logica AND. Possiamo anche riscrivere la tabella usando una *condizione di indifferenza* per il bit B:

#	A	B	$A \cdot B$
0	0	x	0
1	1	x	B

In sostanza, il valore di B viene mantenuto unicamente quando $A = 1$ e questo si riflette in modo immediato sui totali parziali e sull'intera meccanica dell'operazione. Per evitare lunghe e dispendiose perifrasi, ci affidiamo ad un esempio:

(5)			1	0	1	×	MPD
(6)			1	1	0		MPR
			0	0	0		(0)
		1	0	1			(1)
	1	0	1				(1)
(30)	1	1	1	1	0		RES

L'evoluzione dei totali parziali, in sostanza, corrisponde con il core dell'intero algoritmo: poiché abbiamo solo due casi possibili (il totale parziale può essere unicamente nullo o identico al MPD, a meno degli scorrimenti), partecipano alla somma solo quei valori corrispondenti ad un bit alto del moltiplicatore.

Dopo avere preliminarmente posto a zero il risultato RES, ad ogni passo di una moltiplicazione di due parole binarie a n bit ciascuna (moltiplicazione $n \times n$):

1. Si considera il bit i-esimo del moltiplicatore MPR, da destra a sinistra, a partire dal LSB (bit 0);
2. Se tale bit è pari a 1, si esegue la somma (a $2n$ bit) del moltiplicando MPD con il risultato RES;
3. Si fa scorrere di una posizione a *sinistra* il moltiplicando MPD, considerato come valore a $2n$ bit;
4. Si itera il procedimento ripartendo dal punto 1 e considerando il bit successivo $i+1$ (a sinistra dell'attuale) di MPR, fino al raggiungimento del bit $n-1$.

Il codice sorgente dell'implementazione si potrebbe in prima istanza configurare come segue. Fa uso di una variabile ausiliaria che viene impiegata come byte alto del moltiplicando, per gestirne gli scorrimenti senza overflow:

```
;**************************************
;** Moltiplicazione 8x8 bit,
;** risultato a 2 byte
;**************************************

    *=$C000

;**************************************
Mult8x8 LDA #0
        STA MPDH        ; Azzera il risultato e la variabile
        STA Res         ; ausiliaria.
        STA Res+1
        LDX #8          ; Ripete per 8 bit

Mult    LSR MPR         ; Scorrimento a destra, LSB->Carry
        BCC NoSum       ; Se il LSB era basso, procede oltre

        CLC             ; Res = Res + MPD
        LDA Res
        ADC MPD
        STA Res

        LDA Res+1
        ADC MPDH
        STA Res+1

NoSum   ASL MPD         ; Sposta il MSB nel Carry
        ROL MPDH        ; Carry->LSB di MPDH

        DEX
        BNE Mult

End     RTS
;**************************************
;** Variabili inizializzate
;** Moltiplicando, 8 bit
MPD byte 15
;** Moltiplicatore, 8 bit
MPR byte 6
;** Risultato, 16 bit
Res word 0
;** Estensione a 16 bit di MPD
MPDH = $FB
```

Moltiplicazione modulare 8x8 bit ottimizzata. Il codice appena presentato è quello che probabilmente sarebbe stato prodotto da un programmatore HLL dopo avere letto la procedura illustrata in un qualsiasi testo di aritmetica digitale. Ha il pregio di una immediata leggibilità e si tratta di una implementazione sostanzialmente corretta, ma è ampiamente migliorabile, come mostra il sorgente che segue.

```
Line    Addr ## Code        Source
-------------------------------------------
00001   0000                ;**************************************
00002   0001                ;** MOLTIPLICAZIONE 8X8 BIT,
00003   0001                ;** RISULTATO A 2 BYTE
00004   0001                ;** VERSIONE MIGLIORATA
00005   0001                ;**************************************
00006   0001
00007   0001                        *=$C000
00008   C000
```

```
00009   C000                    ;***************************************
00010   C000                    ;* MOLTIPLICANDO, 8 BIT
00011   C000                    MPD         = $FB
00012   C000                    ;* MOLTIPLICATORE, 8 BIT
00013   C000                    MPR         = $FC
00014   C000                    ;* RISULTATO, 16 BIT
00015   C000                    RES         = $FD
00016   C000
00017   C000                    ;***************************************
00018   C000 2   A9 0C         START       LDA #12     ; RES = 12 X 7
00019   C002 3   85 FB                     STA MPD
00020   C004 2   A9 07                     LDA #7
00021   C006 3   85 FC                     STA MPR
00022   C008 6   20 22 C0                  JSR MULT8X8
00023   C00B
00024   C00B 2   A9 99                     LDA #$99    ; RES = 153 X 10
00025   C00D 3   85 FB                     STA MPD
00026   C00F 2   A9 0A                     LDA #$0A
00027   C011 3   85 FC                     STA MPR
00028   C013 6   20 22 C0                  JSR MULT8X8
00029   C016
00030   C016 2   A9 FF                     LDA #$FF    ; RES = 255 X 255
00031   C018 3   85 FB                     STA MPD
00032   C01A 2   A9 FF                     LDA #$FF
00033   C01C 3   85 FC                     STA MPR
00034   C01E 6   20 22 C0                  JSR MULT8X8
00035   C021
00036   C021 6   60             END         RTS
00037   C022                    ;***************************************
00038   C022
00039   C022                    ;***************************************
00040   C022                    ;** SUBROUTINE DI MOLTIPLICAZIONE 8X8
00041   C022                    ;***************************************
00042   C022 2   A9 00         MULT8X8     LDA #0      ; AZZERA IL RISULTATO
00043   C024 3   85 FD                     STA RES
00044   C026 3   85 FE                     STA RES+1
00045   C028 2   A2 08                     LDX #8      ; RIPETE 8 VOLTE
00046   C02A
00047   C02A 5   46 FC         MULT        LSR MPR     ; SCORRE MPR
00048   C02C 2*  90 03                     BCC NOSUM
00049   C02E
00050   C02E 2   18                        CLC         ; LASCIA IL PARZIALE IN A
00051   C02F 3   65 FB                     ADC MPD
00052   C031
00053   C031 2   6A             NOSUM      ROR A       ; SCORRE IL RISULTATO
00054   C032 5   66 FD                     ROR RES
00055   C034
00056   C034 2   CA                        DEX
00057   C035 2*  D0 F3                     BNE MULT
00058   C037
00059   C037 3   85 FE                     STA RES+1
00060   C039 6   60             EXIT       RTS
00061   C03A                    ;***************************************
```

L'ottimizzazione rispetto alla versione «ingenua» è palese, sebbene l'algoritmo sotteso sia invariato: iterazione di una somma condizionata avente di volta in volta per addendi il valore del moltiplicando e il precedente risultato parziale. Come già visto in altri esempi precedenti, si utilizza l'Accumulatore per contenere il byte alto del risultato parziale, e l'effetto degli scorrimenti a *sinistra* dei risultati parziali stessi viene ottenuto equivalentemente (ma in maniera più efficiente) facendo invece scorrere a *destra* ad ogni iterazione i due byte del risultato parziale, rispettivamente `A` e `Res`. Ne risulta un codice più veloce e più compatto.

Il lettore tragga le dovute conclusioni dalla comparazione tra i due approcci all'algoritmo e rifletta sulla *forma mentis* da raggiungere per creare codice efficiente in Assembly.

Divisione modulare 8x8 bit. I principi operativi di moltiplicazioni con diverse ampiezze degli operandi (es. 16×8, che come già illustrato richiederà un risultato a $16+8=24$ bit) e della divisione (che procede parimenti per sottrazioni e scorrimenti) sono del tutto identici a quelli fin qui illustrati. Proponiamo quindi un ultimo esempio, questa volta una *divisione* 8x8, sempre con organizzazione modulare.

```
Line   Addr ## Code        Source
-------------------------------------------------------------------------------
00001  0000               ;**************************************
00002  0001               ;** DIVISIONE 8X8 BIT,
00003  0001               ;** RISULTATO A 1 BYTE
00004  0001               ;**************************************
00005  0001
00006  0001                        *=$C000
00007  C000
00008  C000               ;**************************************
00009  C000               ;* DIVIDENDO, 8 BIT
00010  C000               OP1  = $FB
00011  C000               ;* DIVISORE, 8 BIT
00012  C000               OP2  = $FC
00013  C000               ;* QUOZIENTE, 8 BIT
00014  C000               QUOT = $FD
00015  C000
00016  C000               ;**************************************
00017  C000 2  A9 79      START       LDA #121
00018  C002 3  85 FB                  STA OP1
00019  C004 2  A9 07                  LDA #7
00020  C006 3  85 FC                  STA OP2
00021  C008 6  20 17 C0               JSR DIV8X8
00022  C00B
00023  C00B 2  A9 99                  LDA #$99
00024  C00D 3  85 FB                  STA OP1
00025  C00F 2  A9 0A                  LDA #$0A
00026  C011 3  85 FC                  STA OP2
00027  C013 6  20 17 C0               JSR DIV8X8
00028  C016
00029  C016 6  60         END         RTS
00030  C017               ;**************************************
00031  C017
00032  C017               ;**************************************
00033  C017               ;** SUBROUTINE DI DIVISIONE 8X8
00034  C017               ;**************************************
00035  C017 2  A9 00      DIV8X8      LDA #0
00036  C019 2  A0 08                  LDY #8      ; RIPETE 8 VOLTE
00037  C01B
00038  C01B 5  06 FB      UP          ASL OP1     ; SCORRE IL DIVIDENDO
00039  C01D 2  2A                     ROL A       ; NELL'ACCUMULATORE
00040  C01E 3  C5 FC                  CMP OP2     ; CONFRONTA COL DIVISORE
00041  C020 2* 90 02                  BCC DOWN    ; SE TROPPO GRANDE, NON SOTTRAE
00042  C022
00043  C022 3  E5 FC                  SBC OP2     ; EFFETTUA LA SOTTRAZIONE
00044  C024 5  26 FD      DOWN        ROL QUOT    ; SPOSTA IL CARRY NEL QUOZIENTE
00045  C026 2  88                     DEY
00046  C027 2* D0 F2                  BNE UP
00047  C029
00048  C029 6  60         EXIT        RTS
00049  C02A               ;**************************************
```

Per gli scopi e i limiti di un testo introduttivo, si può così considerare esaurito l'argomento delle routine aritmetiche. Si rimanda ai numerosi testi in bibliografia per eventuali approfondimenti.

5.4.4 Istruzioni logiche e conversioni.

Lavorare in Assembly significa anche e soprattutto operare sui bit. Questo genere di operazione può essere familiare per alcuni programmatori in C abituati ad utilizzare gli operatori bitwise, ma generalmente risulta oscuro per chi lavora con la maggioranza dei linguaggi HLL. Che si tratti di configurare i registri di un chip periferico, estrarre il valore di un bit in un registro o emulare un protocollo seriale in *bitbang* (ovvero serializzando opportunamente un byte, un bit alla volta), è uno skill essenziale per un programmatore Assembly saper eseguire senza errori le operazioni fondamentali sui bit di un registro:

Set: impostazione a 1;

Reset: azzeramento;

Toggle: inversione;

Test: isolamento e controllo di un singolo bit in posizione arbitraria.

5.4.4.1 Operazioni logiche fondamentali.

Si rimanda il lettore alle tabelle di verità delle funzioni logiche già presentate nella parte introduttiva. Gli idiomi fondamentali sono concentrati nel seguente esempio, da seguire (come gli altri) con un monitor LM, nel quale si apprezza appieno la comodità di poter usare direttamente valori binari per le *maschere*, ossia le costanti impiegate per le operazioni logiche:

```
      *=$C000

Start   LDA #%00101111
        AND #%00000100 ; Si isola il bit 2
        ORA #%11000000 ; Imposta i bit 7 e 6
        EOR #%00000100 ; Inverte il bit 2
        EOR #%00000100 ; Inverte nuovamente il bit 2
        LDA #%00010011
        AND #%00100000 ; Isola il bit 5
        BNE NotZero    ; Se il bit 5 è alto, effettua il salto
        LDA #$4F       ; Se il bit 5 era nullo, A<-$4F
NotZero AND #$0F       ; Isola il nibble basso, come già visto
        LDA #$A5
        AND #$F0       ; Isola il nibble alto
        LDA #$CD
        AND #%01010101 ; Isola i bit di posto pari
End     RTS
```

Tra le operazioni logiche occorre menzionare anche l'istruzione `BIT`, che esegue un AND implicito senza salvarne il risultato ma impostando i flag di conseguenza (in particolare il flag di zero), e quindi fa uso di maschere AND. Tuttavia, tale istruzione presenta numerose peculiarità e, soprattutto, può utilizzare solo due modi di indirizzamento (pagina zero e assoluto, rispettivamente con costo di 3 e 4 cicli) il che ne limita fortemente l'uso.

5.4.4.2 Scorrimenti e rotazioni.

Come già a più riprese accennato, le istruzioni di rotazione e scorrimento sono borderline tra logica e aritmetica (due campi spesso ampiamente sovrapposti in matematica discreta e computazionale). Abbiamo già accennato al loro utilizzo aritmetico, in particolare per divisioni e moltiplicazioni: vediamo un altro caso tipico in questa sezione specifica. La combinazione tra estrazione sequenziale dei bit tramite rotazione e isolamento dei nibble con maschere `AND` è di fondamentale utilità nelle routine di conversione di base e visualizzazione.

Visualizzazione di un byte in binario. L'esempio più classico ci consente di visualizzare il valore binario di una locazione arbitraria di memoria. In questo caso, si fa uso di un buffer in pagina zero per rendere più efficienti gli scorrimenti in-place e la somma in accumulatore col valore ASCII/PETSCII di '0', che rende stampabile il singolo bit 0 o 1:

```
Line   Addr ## Code        Source
-------------------------------------------------------------------------
00001  0000               ;*************************************
00002  0001               ;** VISUALIZZA UN BYTE IN BINARIO
00003  0001               ;*************************************
00004  0001
00005  0001                        *=$C000
00006  C000
00007  C000               ;** ROUTINE KERNAL PER STAMPA CARATTERE
00008  C000               CHROUT = $FFD2
00009  C000               ;** LOCAZIONE DI MEMORIA DA VISUALIZZARE
00010  C000               MEMORY = $AA24
00011  C000               ;** BUFFER IN PAGINA ZERO
00012  C000               BUFFER = $FB
00013  C000
00014  C000               ;*************************************
00015  C000 4  AD 24 AA   DISPBIN     LDA MEMORY ; COPIA IL CONTENUTO DELLA CELLA
00016  C003 3  85 FB                  STA BUFFER ; IN UN BUFFER DI PAGINA ZERO
00017  C005 2  A0 08                  LDY #8     ; RIPETE PER GLI 8 BIT
00018  C007
00019  C007 2  A9 00      LOOP        LDA #0     ; PREPARA L'ACCUMULATORE
00020  C009 5  26 FB                  ROL BUFFER ; MSB->CARRY
00021  C00B 2  69 30                  ADC #$30   ; CONVERTE IN PETSCII
00022  C00D 6  20 D2 FF               JSR CHROUT ; VISUALIZZA IL BIT
00023  C010 2  88                     DEY
00024  C011 2* D0 F4                  BNE LOOP
00025  C013 6  60         END         RTS
00026  C014               ;*************************************
```

Visualizzazione in esadecimale. Come ulteriore esempio presentiamo conversione da binario (1 byte) a due cifre esadecimali. Scegliamo però una implementazione più sofisticata, che ci consente di illustrare una tecnica universale la quale (al costo di una certa occupazione di memoria) offre i tempi di esecuzione *in assoluto migliori* rispetto a qualsiasi sequenza di istruzioni Assembly: la tabella di lookup (**LUT**: Look-Up Table). Quando i valori da convertire sono in numero ragionevole, tipicamente meno di 256 byte (limite indirizzabile con un singolo registro indice), si può tabulare una qualsiasi funzione arbitraria $f : \mathbb{N} \to \mathbb{N}$ usando un semplice array.

```
;*************************************
;** Visualizza un byte in esadecimale
;*************************************

        *=$C000

;** Routine KERNAL per stampa carattere
CHROUT = $FFD2

;*************************************
Start   LDA #$A9
        JSR Bin2Hex

        LDA #$82
        JSR Bin2Hex

        LDA #$F0
        JSR Bin2Hex
```

```
Exit    RTS
;**************************************

;**************************************
;** Subroutine di conversione in HEX
;** tramite LUT
;**************************************
Bin2Hex TAX             ; Salva temporaneamente il valore
        LSR A           ; Isola il nibble alto
        LSR A           ; e lo sposta nel nibble basso
        LSR A           ;
        LSR A           ;
        TAY             ; Carica il valore in Y
        LDA HEX_LUT,Y   ; A = HEX_LUT[Y]
        JSR CHROUT

        TXA             ; Ripristina efficientemente il valore
        AND #$0F        ; Isola il nibble basso
        TAY
        LDA HEX_LUT,Y
        JSR CHROUT

End     RTS
;**************************************
;** Tabella di lookup
HEX_LUT byte '0', '1', '2', '3', '4', '5', '6', '7'
        byte '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'
```

Come si vede, la tabella è semplicissima così come il suo utilizzo: il dato da convertire, opportunamente condizionato (in questo caso sappiamo che non può superare il valore binario $1111b$ ossia 15 in decimale), viene **usato direttamente come indice**! Così *con un singolo accesso indicizzato*[18] la nostra LUT è in grado di restituire direttamente il carattere ASCII/PETSCII corrispondente a ciascuno dei sedici possibili valori di un nibble binario (4 bit), ossia una singola cifra esadecimale. *Nessuna* possibile sequenza di istruzioni della ISA è in grado di fornire un risultato anche solo comparabile in termini di costo in cicli: senza scendere nei dettagli, le soluzioni classiche ubique in letteratura (es. [Jon84, Lev86]) impiegano almeno un salto condizionato (idealmente concepito per eseguire il salto in 6 casi su 16 ovvero nel $37,5\%$ dei casi, ma spesso la scelta effettuata è invece quella statisticamente più svantaggiosa) e richiedono un minimo best case di *14 cicli per ogni singola cifra esadecimale* (`RTS` incluso, anche se è possibile ristrutturare la subroutine in modo da operare direttamente su due cifre, comprimendo un po' i costi). Riportiamo anche il listing completo, per comodità del lettore.

```
Line    Addr ## Code        Source
-------------------------------------------
00001   0000                ;**************************************
00002   0001                ;** VISUALIZZA UN BYTE IN ESADECIMALE
00003   0001                ;**************************************
00004   0001
00005   0001                        *=$C000
00006   C000
00007   C000                ;** ROUTINE KERNAL PER STAMPA CARATTERE
00008   C000                CHROUT      = $FFD2
00009   C000
00010   C000                ;**************************************
00011   C000 2   A9 A9      START       LDA #$A9
00012   C002 6   20 10 C0              JSR BIN2HEX
00013   C005
00014   C005 2   A9 82                 LDA #$82
00015   C007 6   20 10 C0              JSR BIN2HEX
```

[18] In questo caso dimostrativo sono necessari 4 cicli (5 worst case) usando la modalità di indirizzamento assoluto indicizzato Y, al costo di tre byte in memoria.

```
00016   C00A
00017   C00A  2   A9  F0              LDA #$F0
00018   C00C  6   20  10  C0          JSR BIN2HEX
00019   C00F
00020   C00F  6   60          EXIT    RTS
00021   C010                  ;***************************************
00022   C010
00023   C010                  ;***************************************
00024   C010                  ;** SUBROUTINE DI CONVERSIONE IN HEX
00025   C010                  ;** TRAMITE LUT
00026   C010                  ;***************************************
00027   C010  2   AA          BIN2HEX TAX             ; SALVA TEMPORANEAMENTE IL VAL
00028   C011  2   4A                  LSR A           ; ISOLA IL NIBBLE ALTO
00029   C012  2   4A                  LSR A           ; E LO SPOSTA NEL NIBBLE BASSO
00030   C013  2   4A                  LSR A           ;
00031   C014  2   4A                  LSR A           ;
00032   C015  2   A8                  TAY             ; CARICA IL VALORE IN Y
00033   C016  4*  B9  27  C0          LDA HEX_LUT,Y   ; A = HEX_LUT[Y]
00034   C019  6   20  D2  FF          JSR CHROUT
00035   C01C
00036   C01C  2   8A                  TXA             ; RIPRISTINA EFFICIENTEMENTE I
00037   C01D  2   29  0F              AND #$0F        ; ISOLA IL NIBBLE BASSO
00038   C01F  2   A8                  TAY
00039   C020  4*  B9  27  C0          LDA HEX_LUT,Y
00040   C023  6   20  D2  FF          JSR CHROUT
00041   C026
00042   C026  6   60          END     RTS
00043   C027                  ;***************************************
00044   C027                  ;** TABELLA DI LOOKUP
00045   C027      30  31  32  HEX_LUT BYTE '0', '1', '2', '3', '4', '5', '6', '7'
00046   C02F      38  39  41          BYTE '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'
```

5.4.4.3 Funzione di parità (e dintorni).

Riproponiamo, per comodità del lettore, la tabella completa delle funzioni logiche di due variabili, ossia $f_j : \{0,1\}^2 \to \{0,1\}$:

#	A	B	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1	0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
2	1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
3	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Osserviamo sotto una diversa luce l'aggregato dei bit A e B in ingresso, sempre considerati come una singola variabile intera tale che B è il bit meno significativo LSB (ossia il più a destra), e classifichiamone i valori in base al numero di bit pari a 1 presenti:

A	B	Bit alti	Parità
0	0	0	Pari
0	1	1	Dispari
1	0	1	Dispari
1	1	2	Pari

Nella colonna più a destra abbiamo semplicemente specificato se il numero di bit alti è pari[19] o dispari. Tra le sedici funzioni elencate nella tabella combinatoria completa, abbiamo evidenziato in rosso quella particolare

[19] Ricordiamo che secondo la moderna definizione universalmente accettata in matematica discreta, un numero naturale si dice *pari* se è un **multiplo** di due, il che include anche lo zero.

funzione detta «di parità» che espone la proprietà seguente: $f_j(A, B) = 1$ se e solo se «il numero di bit pari a 1 nel vettore booleano di ingresso è *dispari*», quindi in sostanza quando $A \neq B$. Si tratta in realtà della funzione già nota ai lettori come `XOR`: non a caso, una definizione alternativa della **funzione di parità** è `A XOR B` o $A \oplus B$[20].

Oltre alla funzione di parità propriamente detta, in informatica si fa uso anche della sua simmetrica ovvero negata, codificata alla posizione 9 nella nostra tabella. Si hanno quindi due basilari definizioni di parità: «parità pari» e «parità dispari» (ben note a chiunque abbia mai configurato una porta seriale RS232!) che possono forse confondere il lettore. Se ne suggerisce una definizione facilmente memorizzabile, che dovrebbe sgombrare il campo dalle potenziali ambiguità:

Parità PARI - si contano i bit pari a 1 nel vettore di ingresso dato (tipicamente 7 bit, per ragioni storiche, ma l'ampiezza non ha alcun limite teorico e può aggregare più bytes). Se tale numero è *dispari*, si pone a 1 il bit di parità: in questo modo, il **numero complessivo** di bit alti (incluso quello di parità!) sarà *pari*, in accordo con la denominazione. Coincide con la *funzione di parità* definita appena sopra.

Parità DISPARI - funziona in modo complementare alla precedente. Si contano ancora i bit pari a 1 nel vettore di ingresso dato. Se tale numero è *pari*, si pone a 1 il bit di parità: in questo modo, il numero complessivo di bit alti (incluso quello di parità!) sarà *dispari* - di nuovo, in accordo con la denominazione.

Quindi, a titolo di esempio, considerando un aggregato di sette bit di dati e uno di parità, avremo le possibilità illustrate in tabella:

Bit di parità PARI	Bit di parità DISPARI	# di bit alti
0	1	0, 2, 4, o 6
1	0	1, 3, 5 o 7

Questa funzione assume grande rilevanza nelle prime fasi della telematica e del controllo seriale: essendo relativamente efficiente da calcolare[21] e richiedendo un solo bit, tale funzione può essere utilizzata per un primitivo controllo di integrità, sebbene non abbia la capacità di *correggere* gli errori come invece i codici Hamming, Reed-Solomon e altri analoghi. Inoltre molte architetture (specialmente mainframe) tra gli anni Sessanta e Ottanta fanno uso di un bit di parità associato a *ciascuna parola di memoria*[22]

Quasi tutte le architetture home sono già dotate di chip seriali con controllo di parità incorporato: tuttavia, vale la pena di conoscere gli algoritmi relativi. Soluzioni hardware a parte, il metodo software *in assoluto* più efficiente per il calcolo della parità consiste nell'uso di una LUT come appena visto al paragrafo (5.4.4.2), in questo caso con 256 locazioni, il massimo indirizzabile tramite uno dei registri indice X o Y della CPU: il codice necessario si discosta di pochissimo da quello già visto per la conversione in esadecimale (e ciò vale in generale per l'uso di qualsiasi LUT di dimensioni ragionevoli) e non viene qui riproposto.

Tuttavia, l'uso di una LUT di tale dimensione su un home computer non è sempre fattibile: ecco quindi che conviene ragionare ancora sulle proprietà dello XOR per elaborare una soluzione intermedia, che non richieda ingenuamente sette o otto operazioni per byte[23] né una LUT con 256 locazioni, ma faccia un uso bilanciato di una LUT a 16 locazioni e di un minimo di elaborazione, in tempo costante $O(1)$ anche se subottimale rispetto all'uso di una LUT completa.

Se scriviamo per esteso la funzione di parità $P(B)$ di un byte[24] come catena di XOR, usando la normale codifica posizionale dei bit decrescente verso destra, avremo:

$$P(B) = b_7 \oplus b_6 \oplus b_5 \oplus b_4 \oplus b_3 \oplus b_2 \oplus b_1 \oplus b_0$$

Sappiamo che l'algebra booleana gode delle normali proprietà commutativa e associativa, il che ci consente di riscrivere la sequenza come più ci fa comodo, e in particolare come segue:

$$P(B) = (b_7 \oplus b_3) \oplus (b_6 \oplus b_2) \oplus (b_5 \oplus b_1) \oplus (b_4 \oplus b_0)$$

In questo modo, di fatto, stiamo prima effettuando uno XOR tra i due nibble alto e basso, e poi sommando i risultati intermedi. Tre brevissime considerazioni di passaggio:

[20] Tale notazione risulta ovviamente generalizzabile alle funzioni di più variabili, con vettore binario in ingresso di generica dimensione k, ovvero $f : \{0,1\}^k \to \{0,1\}$, come $f(n) = n_0 \oplus n_1 \oplus \ldots \oplus n_{k-1}$

[21] La funzione di parità è implementata fin dagli albori dell'elettronica digitale in appositi chip TTL, utilizzabili anche per la gestione del bit di parità nella scrittura e lettura da memorie RAM, in alcune CPU di fascia superiore (specialmente nelle architetture dotate di cache) e praticamente in tutti gli adattatori di interfaccia seriale (UART e affini).

[22] Una delle tante idee che verranno poi spacciate anni dopo nel mainstream come «innovazione» con l'uso di memorie a controllo di parità, ECC e derivati di ogni genere anche sui Personal SOHO, sempre più low cost.

[23] Sono possibili soluzioni alternative, sempre basate su loop, in $O(\log n)$.

[24] A rigore, come già implicitamente richiamato, in ambito seriale la parità si riferisce a 7 bit con il MSB sempre nullo, ossia l'originale codifica ASCII-7.

1. Tale procedimento è facilmente generalizzabile ad una qualsiasi ampiezza di parola $n = 2k$, accoppiando gli XOR $b_i \oplus b_j$ in modo tale che valga $i = j + k$ (con $k \leq i \leq n,\ 0 \leq j < k$) per i pedici e quindi per le posizioni dei bit. Quindi risulta portabile anche su CPU modernissime, a 64 e 128 bit, magari estendendo la LUT a 256 posizioni.

2. Il procedimento è chiaramente applicabile in modo ricorsivo. In effetti, si tratta di un esempio fondamentale del concetto chiave di *raddoppio ricorsivo*, fondamentale in ogni forma (anche elementare) di parallelizzazione. Dal momento che, dando la priorità alle operazioni tra parentesi, il numero di bit si dimezza ad ogni passaggio, si avrà in definitiva una prestazione in $O(\log n)$.

3. In realtà, è possibile applicare un qualsiasi schema di ordinamento dei bit, in particolare su varie classi di CPU è molto usato quello detto «a pettine» che consiste semplicemente nell'accoppiare gli indici dispari e pari immediatamente adiacenti, i.e. 7 e 6, 5 e 4 e così via, il che può essere ottenuto molto semplicemente ed efficientemente effettuando uno XOR tra il valore originale e il medesimo valore dopo uno scorrimento (a destra, per fissare le idee) di una posizione, mascherando poi con un AND i bit non rilevanti. Esempio su 8 bit, dove $\wedge$ indica l'AND, x una condizione di indifferenza e $b_{i,i-1}$ il risultato dello XOR dei bit adiacenti di posto i e $i-1$, con $0 < i \leq 7$:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	$\oplus$
0	b_7	b_6	b_5	b_4	b_3	b_2	b_1	$=$
x	$b_{7,6}$	x	$b_{5,4}$	x	$b_{3,2}$	x	$b_{1,0}$	$\wedge$
0	1	0	1	0	1	0	1	$=$
0	$b_{7,6}$	0	$b_{5,4}$	0	$b_{3,2}$	0	$b_{1,0}$	

L'algoritmo proposto consiste invece, semplicemente, nell'applicare un primo XOR tra i due nibble alto e basso (purtroppo come già detto penalizzato su 65xx dalla mancanza di istruzioni atomiche per l'isolamento del nibble alto con swap o shift atomico di quattro posizioni) per ridurre il byte iniziale ad un singolo nibble equivalente dal punto di vista della parità, del quale poi si ricava direttamente la parità stessa tramite una LUT a 16 entries. Oltre ad essere un buon pretesto didattico, questa soluzione costituisce un esempio di *compromesso ingegneristico accettabile* tra utilizzo di spazio e velocità di esecuzione, emblematico di numerose situazioni analoghe che si presentano realmente sugli home computer, richiedendo decisioni di progetto ponderate e buona capacità algoritmica prima che implementativa.

```
Line    Addr ## Code         Source
-------------------------------------------------------
00001   0000                 ;*************************************
00002   0001                 ;** CALCOLA LA PARITA' PARI DI UN BYTE
00003   0001                 ;** VERSIONE CON LUT RIDOTTA
00004   0001                 ;*************************************
00005   0001
00006   0001                         *=$C000
00007   C000
00008   C000                 ;** ROUTINE KERNAL PER STAMPA CARATTERE
00009   C000                 CHROUT      = $FFD2
00010   C000
00011   C000                 ;*************************************
00012   C000                 ;* BYTE DA CONTROLLARE
00013   C000                 OP1         = $FB
00014   C000
00015   C000                 ;*************************************
00016   C000 2   A9 4B       START       LDA #%01001011  ; PARITY = 0
00017   C002 3   85 FB               STA OP1
00018   C004 6   20 0F C0            JSR EVEN_P
00019   C007
00020   C007 2   A9 D9               LDA #%11011001  ; PARITY = 1
00021   C009 3   85 FB               STA OP1
00022   C00B 6   20 0F C0            JSR EVEN_P
00023   C00E
00024   C00E 6   60           EXIT        RTS
00025   C00F                 ;*************************************
00026   C00F
```

```
00027  C00F                    ;**************************************
00028  C00F                    ;** SUBROUTINE DI CALCOLO PARITA' PARI
00029  C00F                    ;**************************************
00030  C00F 3  A5 FB     EVEN_P      LDA OP1
00031  C011 2  4A                    LSR A           ; ISOLA IL NIBBLE ALTO
00032  C012 2  4A                    LSR A           ; E LO SPOSTA NEL NIBBLE BASSO
00033  C013 2  4A                    LSR A           ;
00034  C014 2  4A                    LSR A           ;
00035  C015 3  45 FB                 EOR OP1         ; CALCOLA LO XOR DEI DUE NIBBLE
00036  C017 2  29 0F                 AND #$0F        ; SCARTA IL NIBBLE ALTO
00037  C019 2  AA                    TAX             ; CARICA IL VALORE IN X
00038  C01A 4* BD 23 C0               LDA EP_LUT,X    ; A = EP_LUT[X]
00039  C01D 2  09 30                 ORA #$30        ; CONVERTE IN ASCII
00040  C01F 6  20 D2 FF               JSR CHROUT      ; STAMPA LA PARITA'
00041  C022 6  60            END     RTS
00042  C023                    ;**************************************
00043  C023                    ;** TABELLA DI LOOKUP
00044  C023                    ;** EP = EVEN PARITY
00045  C023    00 01 01        EP_LUT      BYTE 0, 1, 1, 0, 1, 0, 0, 1
00046  C02B    01 00 00                    BYTE 1, 0, 0, 1, 0, 1, 1, 0
```

5.4.4.4 CRC16.

Il controllo di ridondanza ciclico (Cyclic Redundancy Check, CRC) è un semplice codice di *controllo d'errore* basato sulla divisione modulare di polinomi con coefficienti in un campo finito di Galois $GF(2)$ o ${\mathbb{Z}_2}^{25}$. Esistono letteralmente un'infinità di varianti nella scelta dei polinomi di controllo, la maggior parte delle quali porta i nomi dei principali brand nel mondo dell'automazione industriale e dei protocolli di comunicazione più diffusi. Poiché l'argomento è trattato in modo estremamente approfondito in una mole sterminata di letteratura, non ci interessa qui fornire ulteriori charimenti o descrizioni. Si rimanda il lettore interessato, in primissima istanza, ad un notissimo sito di minuziosa catalogazione amanuense di tali varianti: CRC RevEng.

L'esempio fornito si attesta su una delle varianti di CRC16 universalmente più diffuse, denominata `XMODEM`. Si è optato per una versione del codice priva di LUT, ma dalle prestazioni ragionevoli. Fornendo in input la stringa di prova standard «123456789» si deve ottenere in output il valore di controllo `$31C3`. Senza dilungarci in dettagliate spiegazioni formali, ci limitiamo a sottolineare che il CRC in generale è concepito per essere computazionalmente molto efficiente e infatti i relativi algoritmi, pur con mille varianti di bassa cucina, si riducono ad una breve e semplice **sequenza di scorrimenti e XOR**, strettamente dipendenti dal polinomio scelto. L'unico aspetto da sottolineare nel listato è forse l'idioma tipico utilizzato per lasciare all'Assembler il calcolo della dimensione dell'array, comunque costante, prefissata e nota a tempo di assemblaggio.

```
Line   Addr ## Code        Source
-------------------------------------------------
00001  0000                ;************************************************
00002  0001                ;** CALCOLO CRC16 CON POLINOMIO XMODEM
00003  0001                ;** IN TEMPO COSTANTE E SENZA USO DI LUT.
00004  0001                ;************************************************
00005  0001
00006  0001                        *= $C000
00007  C000
00008  C000                ;************************************************
00009  C000                ;* VARIABILE DI CONTROLLO, 16 BIT
00010  C000                CRC16       = $FB
00011  C000
00012  C000                ;************************************************
00013  C000 2  D8          START       CLD
00014  C001                ;** VALORE INIZIALE RICHIESTO DA XMODEM
00015  C001 2  A0 00               LDY #0
00016  C003 3  84 FB               STY CRC16
```

[25] In questo modo, come già più volte ripetuto anche in modo esplicito a partire dai richiami di algebra booleana, si sfrutta la totale segregazione tra le operazioni tra coppie di bit, che **esclude la presenza di riporti** e consente pertanto la parallelizzazione delle operazioni stesse.

```
00017   C005  3   84 FC              STY CRC16+1
00018   C007
00019   C007  4*  B9 3B C0   BYLOOP   LDA DATA,Y
00020   C00A  6   20 13 C0            JSR CRC_XMODEM
00021   C00D  2   C8                  INY
00022   C00E  2   C0 09               CPY #LENDATA - DATA
00023   C010  2*  30 F5               BMI BYLOOP
00024   C012
00025   C012  6   60          EXIT    RTS
00026   C013                  ;***************************************************
00027   C013
00028   C013                  ;***************************************************
00029   C013                  ;** SUBROUTINE CALCOLO CRC16 XMODEM
00030   C013                  ;***************************************************
00031   C013                  CRC_XMODEM
00032   C013  3   45 FC               EOR CRC16+1
00033   C015  3   85 FC               STA CRC16+1
00034   C017  2   4A                  LSR
00035   C018  2   4A                  LSR
00036   C019  2   4A                  LSR
00037   C01A  2   4A                  LSR
00038   C01B  2   AA                  TAX
00039   C01C  2   0A                  ASL
00040   C01D  3   45 FB               EOR CRC16
00041   C01F  3   85 FB               STA CRC16
00042   C021
00043   C021  2   8A                  TXA
00044   C022  3   45 FC               EOR CRC16+1
00045   C024  3   85 FC               STA CRC16+1
00046   C026
00047   C026  2   0A                  ASL
00048   C027  2   0A                  ASL
00049   C028  2   0A                  ASL
00050   C029  2   AA                  TAX
00051   C02A  2   0A                  ASL
00052   C02B  2   0A                  ASL
00053   C02C  3   45 FC               EOR CRC16+1
00054   C02E  3   85 FC               STA CRC16+1
00055   C030
00056   C030  2   8A                  TXA
00057   C031  2   2A                  ROL
00058   C032  3   45 FB               EOR CRC16
00059   C034  3   A6 FC               LDX CRC16+1
00060   C036  3   85 FC               STA CRC16+1
00061   C038  3   86 FB               STX CRC16
00062   C03A  6   60                  RTS
00063   C03B                  ;***************************************************
00064   C03B                  ;** STRINGA DI CONTROLLO, CRC16 = $31C3
00065   C03B      31 32 33    DATA    TEXT "123456789"
00066   C044                  LENDATA = *
```

5.4.5 Gestione delle stringhe.

Pur essendo naturalmente correlato alla gestione dei loop e degli array, questo argomento risulta normalmente **escluso** dalla maggioranza dei testi sull'Assembly. Le motivazioni sono varie, ma qui possiamo accennare almeno le principali.

1. A basso livello, la gestione di **array** e quella di **stringhe** coincidono ampiamente, essendo quest'ultime un banale sottoinsieme dei primi, in tutte le loro varianti. Pertanto, la maggioranza degli autori pensa: «...visti gli array, viste anche le stringhe.».

2. Allo stesso modo, la «stampa di una stringa» a livello Assembly in genere non è altro che una copia da memoria (generica) a memoria (video buffer). L'unica particolarità, al limite, può essere l'impostazione esplicita degli attributi carattere, che in ogni caso è ancora banalmente una scrittura in memoria ad indirizzi predefiniti.

3. Escludendo famiglie di macchine tra loro molto simili (es. VIC20 e Commodore 64), la specificità degli indirizzi hardware dei controller video, delle routine KERNAL e/o del firmware residente prevale di gran lunga sulla genericità richiesta da un corso sostanzialmente hardware-agnostic (come anche il presente).

4. Per contro, molti algoritmi sulle stringhe risultano eccessivamente sofisticati (Boyer–Moore, Boyer–Moore–Horspool, Knuth–Morris–Pratt, Rabin–Karp, Raita, Sunday, Trigram search, Two-way...), così come strutture dati avanzate tipo *tries*[26]: a maggior ragione se implementati in Assembly su CPU a 8 bit anni Settanta, e sono quindi decisamente inadatti ad un corso introduttivo o intermedio.

5. Ultimo, ma non meno importante: proporre funzioni stringa estremamente generiche, *library-ready* o quasi, può diventare oneroso in termini di complessificazione del codice, che diviene potenzialmente meno comprensibile e quasi sempre meno ottimizzato. L'uso estensivo di puntatori obbliga a ricorrere alle modalità di indirizzamento più lente in assoluto, oppure a codice *self-morphing* (del quale abbiamo già proposto un esempio al paragrafo 5.4.1.3 a pagina 81, in ambito di generica copia multipagina da memoria a memoria).

Le argomentazioni sopra riassunte sono evidentemente valide e fondate, quindi corrette, e anche piuttosto convincenti per la comunità di riferimento dei programmatori. Si può solo aggiungere una ulteriore considerazione: il presente testo introduttivo è dedicato principalmente al lettore con esperienza in linguaggi HLL e poca familiarità con Assembly e C, che ha un'idea di «stringa» come struttura dati complessa (o addirittura *oggetto*) decisamente diversa rispetto alla prospettiva del low level. Ciò diminuisce il peso complessivo delle prime due argomentazioni e, pur lasciando sostanzialmente inalterate la terza e la quarta, induce anche a contestualizzare meglio la quinta. La dimensione media di una stringa, nella generalità delle situazioni applicative, è molto inferiore rispetto ai 255 byte massimi. Quindi, a meno che l'applicazione non sia un editor di testo *screen-oriented* o un text processor (nel qual caso il lettore non avrà neppure necessità del presente manuale, ma piuttosto di un testo avanzato e non necessariamente «retro» sulla progettazione di un pool di idonee strutture dati per ottimizzare la dinamica delle stringhe), la necessità di ottimizzare all'estremo è assai meno impellente e sensata rispetto agli esempi già visti di movimentazione massiva, e si avvicina invece all'esempio 5.4.1.3 a pagina 78.

Dopo attenta valutazione di questi fattori, si è deciso di proporre qui un paio di esempi basilari e sufficientemente generici da poter essere implementati con opportune minimali modifiche sia su schede molto semplici (ovviamente dotate almeno di display LCD) che su un Commodore 64 o superiori.

Si veda la nota 6 a pagina 70 per le differenze tra le varie tipologie di stringhe: qui faremo riferimento a stringhe di tipo BASIC/Pascal con lunghezza prefissa (1 byte) e quindi prive di terminatore[27].

5.4.5.1 Stampa a video di una stringa.

Dopo i numerosi esempi di codice fin qui presentati, si presume che anche il lettore più distratto sia in grado di scrivere in piena autonomia il banalissimo codice per la restituzione della lunghezza della stringa, contenuta nel primo byte della stessa. L'esempio più semplice e utile da cui partire è invece la basilare stampa a video di una stringa passata come parametro, senza preoccuparci di dettagli come la gestione del cursore o degli attributi

[26] Sincresi di *retrieval trees*, detti anche *prefix trees*, si tratta di peculiari alberi-dizionario che consentono una memorizzazione estremamente compatta di stringhe, con tutte le possibilità di attraversamento nei tempi caratteristici delle strutture ad albero.

[27] In realtà occorrerebbe qui lanciarsi in un lungo e approfondito excursus storico sulla interminabile diatriba riguardo le due diverse implementazioni imperanti all'epoca: ASCII-Z (null-terminated) in stile C e Unix o piuttosto *prefixed length* tipiche di Pascal e BASIC e (quindi) anche degli home computer. All'epoca il dibattito ingegneristico verteva ovviamente sui costi di memorizzazione, che tuttavia nel caso (quasi universale nell'ambito degli home computer e primi PC) di stringhe con lunghezza limitata a 255 caratteri perde totalmente di significato: l'overhead è comunque pari ad un solo byte in ambedue i casi.

In estrema sintesi, l'enorme libertà consentita dalle stringhe ASCII-Z (che possono occupare da un minimo teorico di un singolo carattere ad un massimo parimenti teorico che copre tutta la memoria disponibile, senza altro overhead fisso che un singolo byte terminale) aveva l'ovvia contropartita di richiedere un tempo lineare $O(n)$ anziché tempo costante $O(1)$ anche per la più banale delle operazioni, la restituzione della lunghezza della stringa. In soldoni, per conoscere la dimensione di una stringa null-terminated occorre (almeno una volta) una scansione carattere per carattere dell'intera stringa, fino a trovare il terminatore nullo, il che richiede un tempo di elaborazione proporzionale alla lunghezza stessa. Viceversa la lunghezza prefissa richiede un singolo accesso alla memoria, anche se ovviamente occorre aggiornarla quando si eseguono operazioni di modifica sulla stringa stessa, come una concatenazione o anche una semplice sovrascrittura con una nuova stringa.

D'altro canto, la dimensione prefissa contenuta in un byte e poi espansa ad una singola word in varie versioni successive di BASIC e Pascal fu inizialmente scelta proprio perché apparentemente conferiva un notevole vantaggio prestazionale ed elevata semplificazione del codice nell'immediato, ma con la generale crescita della capacità di elaborazione e memorizzazione secondo la legge empirica di Moore ha avuto un devastante effetto boomerang fino alla prima metà degli anni Novanta, costituendo una zavorra di retrocompatibilità assolutamente penalizzante per i linguaggi che avevano adottato tale formato fisso - zavorra che si sommava agli innumerevoli *kludge* dell'architettura x86.

video: quindi usando una funzione di sistema, come peraltro già visto in precedenti esempi. La funzione che proponiamo è comunque molto compatta, efficiente (per quanto lo consenta il tipo di indirizzamento utilizzato) e modulare, quindi adatta con poche o nulle modifiche all'inserimento in un minimale *toolchest* del programmatore principiante.

```
Line   Addr ## Code         Source
-------------------------------------------------------------------------
00001  0000                 ;**************************************
00002  0001                 ;** STAMPA UNA STRINGA BASIC
00003  0001                 ;**************************************
00004  0001
00005  0001                          *=$C000
00006  C000
00007  C000                 ;** ROUTINE KERNAL C64 PER STAMPA CARATTERE
00008  C000                 CHROUT     = $FFD2
00009  C000
00010  C000                 ;** AREA PASSAGGIO PARAMETRI (PAGINA ZERO)
00011  C000                 STR_PTR    = $FE             ; PUNTATORE ALLA STRINGA
00012  C000
00013  C000                 ;**************************************
00014  C000                 START
00015  C000 2  A9 20               LDA #<STR1        ; PREPARA IL PUNTATORE
00016  C002 3  85 FE               STA STR_PTR       ; ALLA STRINGA
00017  C004 2  A9 C0               LDA #>STR1
00018  C006 3  85 FF               STA STR_PTR+1
00019  C008
00020  C008 6  20 0C C0            JSR DISP_STR
00021  C00B
00022  C00B 6  60           END     RTS
00023  C00C                 ;**************************************
00024  C00C                 ; SUBROUTINE DI STAMPA STRINGHE BASIC
00025  C00C                 ; USA: A, X, Y
00026  C00C                 ;**************************************
00027  C00C                 DISP_STR
00028  C00C 2  A0 00               LDY #0            ; INDICE
00029  C00E 5* B1 FE               LDA (STR_PTR),Y   ; DIMENSIONE STRINGA PREF
00030  C010 2  AA                 TAX
00031  C011
00032  C011                 STR_LOOP
00033  C011 2  C8                 INY
00034  C012 5* B1 FE               LDA (STR_PTR),Y
00035  C014 6  20 D2 FF            JSR CHROUT
00036  C017 2  CA                 DEX
00037  C018 2* D0 F7               BNE STR_LOOP
00038  C01A
00039  C01A 2  A9 0D               LDA #$0D
00040  C01C 6  20 D2 FF            JSR CHROUT
00041  C01F
00042  C01F 6  60                 RTS
00043  C020                 ;**************************************
00044  C020                 ;**************************************
00045  C020     16 31 39    STR1    PTEXT "1999: FUGA DA NEW YORK"
00046  C037                 ;**************************************
```

Come è immediato notare, la stampa vera e propria fa uso di una funzione presente nel KERNAL C64 e in molti modelli analoghi. Adattare la routine ad altri sistemi, nella maggioranza nei casi, consisterà semplicemente in una banale modifica dell'indirizzo *hardcoded* della `CHROUT`. Si noti alla riga 00043 del listing che la dimensione della stringa, come già anticipato, è prefissa (16h, ossia 22 bytes) e questo causa un semplice off-by-one (di un byte) del contenuto della stringa stessa. La voluta genericità della routine impone l'uso di un puntatore in pagina zero, peraltro già visto in esempi analoghi, con un costo minimo ben evidenziato pari a 5 cicli per ogni

caricamento in Accumulatore. Per mitigare l'effetto di questo costo, il loop fa uso di ambedue i registri indice: come variabile di induzione a decremento (X) e indice nell'array stringa (Y), eliminando così altri potenziali overhead come l'uso di una `CMP` per l'uscita dal loop.

Al di là di dettagli come la modularizzazione della subroutine o il recupero della lunghezza dalla prima locazione della stringa, si invita nuovamente il lettore a riflettere sulla notevole similarità del codice qui presentato con quello per la serializzazione di un buffer 5.4.1.3 a pagina 78. Questo è il ragionevole quanto intuitivo motivo fondamentale per cui il capitolo «stringhe» risulta sostanzialmente assente dalla maggioranza della manualistica, come già anticipato.

5.4.5.2 Concatenazione di due stringhe.

Mostriamo ora una funzione più avanzata che per molti versi emula il comportamento della `strcat()` dalla libreria standard del C. In questo caso però effettuiamo anche un controllo preliminare sulla lunghezza complessiva delle due stringhe, evitando di eseguire la funzione in caso di superamento del limite massimo consentito dallo standard prescelto. Naturalmente, nel mondo reale, il comportamento della funzione nel caso anomalo è strettamente legato al contesto: si può emettere un semplice codice di errore in una locazione prefissata, oppure stampare a video un vero e proprio messaggio di errore, oppure forzare un troncamento limitando comunque a 255 caratteri la lunghezza della stringa risultante (anche qui, con o senza emissione di un codice d'errore interno a livello di libreria/applicazione), eccetera.

```
Line   Addr ## Code        Source
-----------------------------------------------------------------------
00001  0000               ;**************************************
00002  0001               ;** CONCATENA DUE STRINGHE BASIC
00003  0001               ;** ANALOGA CON MODIFICA ALLA STRCAT()
00004  0001               ;** DEL LINGUAGGIO C
00005  0001               ;**************************************
00006  0001
00007  0001                        *=$C000
00008  C000
00009  C000               ;** ROUTINE KERNAL C64 PER STAMPA CARATTERE
00010  C000               CHROUT   = $FFD2
00011  C000
00012  C000               ;** AREA PASSAGGIO PARAMETRI (PAGINA ZERO)
00013  C000               ORIGIN   = $FA  ; STRINGA DA ACCODARE
00014  C000               DEST     = $FC  ; STRINGA DESTINAZIONE
00015  C000               TEMP     = $FE  ; PUNTATORE TEMPORANEO
00016  C000
00017  C000               ;**************************************
00018  C000               START
00019  C000 2  A9 6E              LDA #<STR2      ; PUNTATORE ALLA STRINGA
00020  C002 3  85 FC              STA DEST        ; DI DESTINAZIONE, CHE
00021  C004 2  A9 C0              LDA #>STR2      ; VIENE MODIFICATA IN MEM
00022  C006 3  85 FD              STA DEST+1
00023  C008
00024  C008 2  A9 53              LDA #<STR1      ; PUNTATORE ALLA STRINGA
00025  C00A 3  85 FA              STA ORIGIN      ; DA ACCODARE
00026  C00C 2  A9 C0              LDA #>STR1
00027  C00E 3  85 FB              STA ORIGIN+1
00028  C010
00029  C010 6  20 17 C0           JSR CAT_STR
00030  C013
00031  C013               ;** STAMPA LA STRINGA MODIFICATA
00032  C013 6  20 3F C0           JSR DISP_STR
00033  C016
00034  C016 6  60         END     RTS
00035  C017
00036  C017               ;**************************************
00037  C017               ;** SUBROUTINE DI CONCATENAZIONE DI
00038  C017               ;** STRINGHE BASIC
00039  C017               ;** USA: A, X, Y
```

```
00040  C017                ;*************************************
00041  C017                CAT_STR
00042  C017                ;** CONTROLLA CHE LA SOMMA DELLE LUNGHEZZE
00043  C017                ;** SIA MINORE DI 256, ALTRIMENTI ESCE.
00044  C017 2  A0 00               LDY #0
00045  C019 5* B1 FC               LDA (DEST),Y
00046  C01B 2  AA                  TAX
00047  C01C 5* 71 FA               ADC (ORIGIN),Y
00048  C01E 2* B0 1E               BCS CAT_EXIT
00049  C020
00050  C020                ;** HOUSEKEEPING PRELIMINARE.
00051  C020 6  91 FC               STA (DEST),Y     ; AGGIORNA LA LUNGHEZZA
00052  C022 5* B1 FA               LDA (ORIGIN),Y
00053  C024 2  A8                  TAY
00054  C025
00055  C025                ;** AGGIORNA IL PUNTATORE AUSILIARIO PER LA
00056  C025                ;** SCRITTURA IN CODA ALLA STRINGA ORIGINE.
00057  C025 2  8A                  TXA
00058  C026 2  18                  CLC
00059  C027 2  D8                  CLD
00060  C028 3  65 FC               ADC DEST
00061  C02A 3  85 FE               STA TEMP
00062  C02C 2  A9 00               LDA #0
00063  C02E 3  65 FD               ADC DEST+1
00064  C030 3  85 FF               STA TEMP+1
00065  C032
00066  C032                ;** LE LUNGHEZZE SONO GIÀ SALVATE IN X E Y
00067  C032                ;** PER VELOCIZZARE E PREPARARE IL LOOP.
00068  C032 2  98                  TYA
00069  C033 2  AA                  TAX
00070  C034 2  A0 00               LDY #0
00071  C036
00072  C036                CAT_LOOP
00073  C036 2  C8                  INY
00074  C037 5* B1 FA               LDA (ORIGIN),Y
00075  C039 6  91 FE               STA (TEMP),Y
00076  C03B 2  CA                  DEX
00077  C03C 2* D0 F8               BNE CAT_LOOP
00078  C03E
00079  C03E                CAT_EXIT
00080  C03E 6  60                  RTS
00081  C03F
00082  C03F                ;*************************************
00083  C03F                ;** SUBROUTINE DI STAMPA STRINGHE BASIC
00084  C03F                ;** USA: A, X, Y
00085  C03F                ;*************************************
00086  C03F                DISP_STR
00087  C03F 2  A0 00               LDY #0           ; INDICE
00088  C041 5* B1 FC               LDA (DEST),Y     ; DIMENSIONE STRINGA PREF
00089  C043 2  AA                  TAX
00090  C044
00091  C044                STR_LOOP
00092  C044 2  C8                  INY
00093  C045 5* B1 FC               LDA (DEST),Y
00094  C047 6  20 D2 FF            JSR CHROUT
00095  C04A 2  CA                  DEX
00096  C04B 2* D0 F7               BNE STR_LOOP
00097  C04D
00098  C04D 2  A9 0D               LDA #$0D
00099  C04F 6  20 D2 FF            JSR CHROUT
00100  C052
```

```
00101   C052 6   60                    RTS
00102   C053                ;*************************************
00103   C053                ;*************************************
00104   C053     1A 20 2D   STR1        PTEXT " - I GUERRIERI DELLA NOTTE"
00105   C06E     08 57 41   STR2        PTEXT "WARRIORS"
00106   C077     00 00 00   BUFFER      BYTES 255, $FF
00107   C275                ;*************************************
```

Come si può facilmente notare, anche questa routine non fa uso del prezioso spazio di stack (risparmiando anche le costose PUSH e POP) e provvede invece a salvare tutti i valori necessari nei registri indice, al costo di qualche scambio in più che forse rende meno scorrevole il codice, ma preserva la prestazione complessiva. Si veda anche la nota 15 a pagina 95.

Il flusso logico è di una semplicità disarmante: posto che stiamo per «allungare» la stringa denominata `DEST` con una scrittura in coda (la verifica della disponibilità effettiva dello spazio necessario è sempre totalmente a carico del programmatore, naturalmente), si ricava la sua ultima locazione usando la lunghezza prefissa e sommandola al puntatore iniziale, ottenendo così la destinazione in cui iniziare a scrivere, carattere per carattere, la stringa `ORIGIN`. Il tutto dopo un banale controllo sulla somma delle due lunghezze date e aggiornamento della lunghezza complessiva. Si noti che non viene in alcun modo gestito il caso di overflow rispetto alla memoria RAM fisicamente presente: ciò esula ampiamente dai fini e dalle possibilità dell'esempio, che si limita ad assumere impostazioni ragionevoli dei puntatori. Stante la enorme varietà di configurazioni hardware che possono ospitare le CPU oggetto della presente trattazione, fermo restando il limite massimo delle 65.536 locazioni stabilito dall'ampiezza del bus indirizzi a 16 bit, tale controllo non può che essere lasciato come proficuo esercizio per il lettore interessato.

Il buffer `BUFFER` di 255 bytes inizializzati a `$FF` a fine programma serve a garantire che l'accodamento non andrà a sovrascrivere alcunché di importante eventualmente presente in memoria, funge da predisposizione per ulteriori modifiche al codice come quelle che indicheremo al prossimo paragrafo, ed è in generale un'ottima abitudine per il test drive di programmi e librerie su macchine con mappe di memoria complesse e interprete residente, specialmente quando si manipolano buffer e stringhe di dimensioni consistenti, con (potenziali) sovrascritture. Inoltre consente facilmente di eseguire dei test con sconfinamento di pagina a metà buffer/stringa, verificando così in modo diretto eventuali *corner case*.

Proposte di esercizio sulle stringhe. Si lascia come banale e utile esercizio per il lettore interessato la realizzazione delle minimali modifiche al codice appena presentato per implementare le seguenti specifiche, peraltro assai comuni in innumerevoli situazioni del real world:

- Copiare una stringa data in una nuova locazione (a tale scopo sarà perfetto il `BUFFER` del codice sopra illustrato), in questo caso avendo la certezza per costruzione che non vi saranno sovrapposizioni in memoria. Si può opzionalmente aggiungere un semplice confronto iniziale tra i puntatori a 16 bit: se coincidono, inutile perdere tempo.
- Come sopra, ma limitando la copia ad un numero **massimo** arbitrario di caratteri passato come parametro: si veda ad esempio la `strncpy()` del C.
- Estrarre una sottostringa da una stringa data, partendo da una posizione arbitraria e copiandola in un buffer preallocato. Tale funzione può essere facilmente estesa e generalizzata con l'uso di due parametri numerici per emulare il comportamento delle varie `LEFT$(), MID$(), RIGHT$()` del BASIC, fino a livelli di sofisticazione molto vicini allo slicing di COMAL o del più noto Python.
- Creare una nuova stringa dalla concatenazione di due stringhe date, in un terzo buffer preallocato, simulando così la funzionalità di uno statement BASIC del tipo `N$=A$+B$`.

Scopo primario di tali esercizi è mostrare con quale facilità sia possibile riciclare oltre il 90% del medesimo codice di base per realizzare le funzionalità indicate (e molte altre, in realtà).

5.4.5.3 Copia «robusta» di una stringa.

Come ultimo e più avanzato esempio vogliamo proporre una versione verticale e leggermente semplificata (non usa il terzo parametro per limitare il numero di bytes copiati) della potente `memmove()` dalla libreria standard del C. La nostra funzione è in grado di copiare una stringa BASIC ovunque in memoria[28], anche nel caso limite

[28] Non viene controllato l'eventuale overflow rispetto ai limiti della RAM fisicamente installata, ma come già ricordato è decisamente banale implementare anche tale controllo secondo il tipo di hardware.

Inoltre, non si controlla se i puntatori di origine e destinazione coincidono, nel qual caso si potrebbe semplicemente evitare di eseguire la routine. Tale elementare modifica è lasciata come ulteriore utile esercizio per il lettore interessato.

di offset minimo di ± 1 byte rispetto al puntatore di origine della stringa data, senza sovrascrtture e perdita di informazione e *senza usare memoria aggiuntiva.*

Nella letteratura tecnica e nelle implementazioni più diffuse, la `memmove()` è una funzione complessa e fa quasi sempre uso di un terzo buffer temporaneo per gestire il caso di aree di origine e destinazione parzialmente sovrapposte. In realtà, tale spreco di risorse non è assolutamente necessario: con uno slack di una manciata di bytes è possibile differenziare la direzione in cui avviene la copia (partendo quindi dal primo o dall'ultimo byte per gestire l'eventuale sovrapposizione) secondo **il segno della differenza** tra i due puntatori origine e destinazione, con prestazioni praticamente identiche nei due casi[29]. Il tutto senza ricorso a ulteriori buffer e senza tutto l'overhead della doppia movimentazione.

La logica è semplicissima, proprio per evitare passaggi superflui: se la destinazione è **anteposta** in memoria all'origine, e quindi `DEST < ORIGIN`, si copierà normalmente a partire dal primo byte con un loop ad incremento, copiando per il primo il byte con l'indirizzo più basso verso un indirizzo ancora più basso. Se, viceversa, la destinazione occupa un indirizzo **maggiore** di quello di origine (anche di un solo byte!), si copia a ritroso partendo dall'ultimo byte della stringa verso una locazione con indirizzo sicuramente maggiore. Questo senza alcun riguardo per il fatto che le due aree siano effettivamente sovrapposte o meno.

Naturalmente è sempre possibile modificare il codice aggiungendo il banale controllo di sovrapposizione delle aree, e in caso di esito negativo richiamare il loop leggermente più efficiente, quello di copia a decremento che (come già ampiamente visto) utilizza il medesimo registro Y come variabile di induzione e come indice per l'accesso alla memoria.

```
;**************************************
;** Copia una stringa BASIC.
;** Funziona anche in caso di
;** sovrapposizione, come la memmove() del C
;**************************************

     *=$C000

;** Routine KERNAL C64 per stampa carattere
CHROUT  = $FFD2

;** Area passaggio parametri (pagina zero)
Origin  = $FA ; Stringa da duplicare
Dest    = $FC ; Destinazione

;**************************************
Start
;** Primo test: forziamo una destinazione
;** alla locazione di memoria immediatamente
;** precedente all'origine: offset -1.
        LDA #<Str1      ; Puntatore alla stringa
        STA Origin      ; di origine
        LDA #>Str1
        STA Origin+1

        LDA #<Dummy     ; Puntatore al buffer
        STA Dest        ; destinazione
        LDA #>Dummy
        STA Dest+1

        JSR Copy_str

;** Stampa la stringa copiata.
        JSR Disp_str

;** Secondo test: forziamo una destinazione
;** alla locazione di memoria immediatamente
```

[29] La differenza in lunghezza e in cicli tra i due inner loop, rispettivamente 1 byte e 2 cicli di clock, risulta decisamente **trascurabile** data la lunghezza media di una stringa. Normalmente non si tratta di movimentare centinaia di stringhe da 255 caratteri cadauna, quindi di fatto intere pagine di memoria, casistica che peraltro abbiamo ampiamente illustrato col più efficiente codice disponibile nell'esempio già richiamato.

```
;** successiva all'origine: offset +1.
        CLC
        LDA #<Str2          ; Puntatore alla stringa
        STA Origin          ; di origine
        ADC #1
        STA Dest            ; destinazione
        LDA #>Str2
        STA Origin+1
        ADC #0
        STA Dest+1

        JSR Copy_str

;** Stampa la stringa copiata.
        JSR Disp_str

;** Ultimo test: sorgente e destinazione
;** sicuramente non sovrapposti.
        LDA #<Dummy
        STA Origin
        LDA #>Dummy
        STA Origin+1

        LDA #<Buffer
        STA Dest
        LDA #>Buffer
        STA Dest+1

        JSR Copy_str

;** Stampa la stringa copiata.
        JSR Disp_str

End     RTS

;****************************************
;** Subroutine di concatenazione di
;** stringhe BASIC
;** Usa: A, X, Y
;****************************************
Copy_str
;** Calcola l'offset tra i puntatori
;** e sceglie di conseguenza la strategia
;** migliore per la copia.
        SEC                 ; Azzera il Prestito (C=1)
        LDA Dest
        SBC Origin
        STA Offset
        LDA Dest+1
        SBC Origin+1
        STA Offset+1        ; Aggiorna il byte alto della somma

;** Offset = Dest - Origin
        BMI Copy_FW         ; Se negativo, copia dal primo byte

;** Copia a decremento dall'ultimo byte
        LDY #0
        LDA (Origin),Y
        TAY
        INY                 ; Off-by one
```

```
BW_loop LDA (Origin),Y
        STA (Dest),Y
        DEY
        BPL BW_loop

        JMP Copy_exit

Copy_FW
;** Copia ad incremento dal primo byte
        LDY #0
        LDA (Origin),Y
        TAX
        STA (Dest),Y    ; Aggiorna la lunghezza

FW_loop INY
        LDA (Origin),Y
        STA (Dest),Y
        DEX
        BNE FW_loop

Copy_exit
        RTS

;***************************************
;** Subroutine di stampa stringhe BASIC
;** Usa: A, X, Y
;***************************************
Disp_str
        LDY #0          ; Indice
        LDA (Dest),Y    ; Dimensione stringa prefissa
        TAX

Str_loop
        INY
        LDA (Dest),Y
        JSR CHROUT
        DEX
        BNE Str_loop

        LDA #$0D
        JSR CHROUT

        RTS

;***************************************
;** Variabili:
Offset  WORD 0          ; Differenza di puntatori
;***************************************
Dummy   BYTE 0
Str1    PTEXT "DIRTY HARRY (1971), MAGNUM FORCE (1973), THE ENFORCER (1976)"
Str2    PTEXT "SUDDEN IMPACT (1983), THE DEAD POOL (1988)"
Buffer  BYTES 255, $FF
;***************************************
```

Per mera comodità del lettore, riportiamo anche il listing assemblato di questo esempio.

```
Line    Addr ## Code        Source
-------------------------------------------
00001   0000                ;***************************************
00002   0001                ;** COPIA UNA STRINGA BASIC.
```

```
00003  0001                ;** FUNZIONA ANCHE IN CASO DI
00004  0001                ;** SOVRAPPOSIZIONE, COME LA MEMMOVE() DEL C
00005  0001                ;**************************************
00006  0001
00007  0001                        *=$C000
00008  C000
00009  C000                ;** ROUTINE KERNAL C64 PER STAMPA CARATTERE
00010  C000                CHROUT      = $FFD2
00011  C000
00012  C000                ;** AREA PASSAGGIO PARAMETRI (PAGINA ZERO)
00013  C000                ORIGIN      = $FA  ; STRINGA DA DUPLICARE
00014  C000                DEST        = $FC  ; DESTINAZIONE
00015  C000
00016  C000                ;**************************************
00017  C000                START
00018  C000                ;** PRIMO TEST: FORZIAMO UNA DESTINAZIONE
00019  C000                ;** ALLA LOCAZIONE DI MEMORIA IMMEDIATAMENTE
00020  C000                ;** PRECEDENTE ALL'ORIGINE: OFFSET -1.
00021  C000 2  A9 8C              LDA #<STR1        ; PUNTATORE ALLA STRINGA
00022  C002 3  85 FA              STA ORIGIN        ; DI ORIGINE
00023  C004 2  A9 C0              LDA #>STR1
00024  C006 3  85 FB              STA ORIGIN+1
00025  C008
00026  C008 2  A9 8B              LDA #<DUMMY       ; PUNTATORE AL BUFFER
00027  C00A 3  85 FC              STA DEST          ; DESTINAZIONE
00028  C00C 2  A9 C0              LDA #>DUMMY
00029  C00E 3  85 FD              STA DEST+1
00030  C010
00031  C010 6  20 44 C0           JSR COPY_STR
00032  C013
00033  C013                ;** STAMPA LA STRINGA COPIATA.
00034  C013 6  20 75 C0           JSR DISP_STR
00035  C016
00036  C016                ;** SECONDO TEST: FORZIAMO UNA DESTINAZIONE
00037  C016                ;** ALLA LOCAZIONE DI MEMORIA IMMEDIATAMENTE
00038  C016                ;** SUCCESSIVA ALL'ORIGINE: OFFSET +1.
00039  C016 2  18                 CLC
00040  C017 2  A9 C9              LDA #<STR2        ; PUNTATORE ALLA STRINGA
00041  C019 3  85 FA              STA ORIGIN        ; DI ORIGINE
00042  C01B 2  69 01              ADC #1
00043  C01D 3  85 FC              STA DEST          ; DESTINAZIONE
00044  C01F 2  A9 C0              LDA #>STR2
00045  C021 3  85 FB              STA ORIGIN+1
00046  C023 2  69 00              ADC #0
00047  C025 3  85 FD              STA DEST+1
00048  C027
00049  C027 6  20 44 C0           JSR COPY_STR
00050  C02A
00051  C02A                ;** STAMPA LA STRINGA COPIATA.
00052  C02A 6  20 75 C0           JSR DISP_STR
00053  C02D
00054  C02D                ;** ULTIMO TEST: SORGENTE E DESTINAZIONE
00055  C02D                ;** SICURAMENTE NON SOVRAPPOSTI.
00056  C02D 2  A9 8B              LDA #<DUMMY
00057  C02F 3  85 FA              STA ORIGIN
00058  C031 2  A9 C0              LDA #>DUMMY
00059  C033 3  85 FB              STA ORIGIN+1
00060  C035
00061  C035 2  A9 F4              LDA #<BUFFER
00062  C037 3  85 FC              STA DEST
00063  C039 2  A9 C0              LDA #>BUFFER
```

```
00064  C03B 3  85 FD              STA DEST+1
00065  C03D
00066  C03D 6  20 44 C0           JSR COPY_STR
00067  C040
00068  C040               ;** STAMPA LA STRINGA COPIATA.
00069  C040 6  20 75 C0           JSR DISP_STR
00070  C043
00071  C043 6  60         END     RTS
00072  C044
00073  C044               ;***************************************
00074  C044               ;** SUBROUTINE DI CONCATENAZIONE DI
00075  C044               ;** STRINGHE BASIC
00076  C044               ;** USA: A, X, Y
00077  C044               ;***************************************
00078  C044               COPY_STR
00079  C044               ;** CALCOLA L'OFFSET TRA I PUNTATORI
00080  C044               ;** E SCEGLIE DI CONSEGUENZA LA STRATEGIA
00081  C044               ;** MIGLIORE PER LA COPIA.
00082  C044 2  38                 SEC             ; AZZERA IL PRESTITO (C=1
00083  C045 3  A5 FC              LDA DEST
00084  C047 3  E5 FA              SBC ORIGIN
00085  C049 4  8D 89 C0           STA OFFSET
00086  C04C 3  A5 FD              LDA DEST+1
00087  C04E 3  E5 FB              SBC ORIGIN+1
00088  C050 4  8D 8A C0           STA OFFSET+1    ; AGGIORNA IL BYTE ALTO D
00089  C053
00090  C053               ;** OFFSET = DEST - ORIGIN
00091  C053 2* 30 10              BMI COPY_FW     ; SE NEGATIVO, COPIA DAL
00092  C055
00093  C055               ;** COPIA A DECREMENTO DALL'ULTIMO BYTE
00094  C055 2  A0 00              LDY #0
00095  C057 5* B1 FA              LDA (ORIGIN),Y
00096  C059 2  A8                 TAY
00097  C05A 2  C8                 INY             ; OFF-BY ONE
00098  C05B
00099  C05B 5* B1 FA      BW_LOOP LDA (ORIGIN),Y
00100  C05D 6  91 FC              STA (DEST),Y
00101  C05F 2  88                 DEY
00102  C060 2* 10 F9              BPL BW_LOOP
00103  C062
00104  C062 3  4C 74 C0           JMP COPY_EXIT
00105  C065
00106  C065               COPY_FW
00107  C065               ;** COPIA AD INCREMENTO DAL PRIMO BYTE
00108  C065 2  A0 00              LDY #0
00109  C067 5* B1 FA              LDA (ORIGIN),Y
00110  C069 2  AA                 TAX
00111  C06A 6  91 FC              STA (DEST),Y    ; AGGIORNA LA LUNGHEZZA
00112  C06C
00113  C06C 2  C8         FW_LOOP INY
00114  C06D 5* B1 FA              LDA (ORIGIN),Y
00115  C06F 6  91 FC              STA (DEST),Y
00116  C071 2  CA                 DEX
00117  C072 2* D0 F8              BNE FW_LOOP
00118  C074
00119  C074               COPY_EXIT
00120  C074 6  60                 RTS
00121  C075
00122  C075               ;***************************************
00123  C075               ;** SUBROUTINE DI STAMPA STRINGHE BASIC
00124  C075               ;** USA: A, X, Y
```

```
00125  C075                  ;**************************************
00126  C075                  DISP_STR
00127  C075 2   A0 00                LDY #0          ; INDICE
00128  C077 5*  B1 FC                LDA (DEST),Y    ; DIMENSIONE STRINGA PREF
00129  C079 2   AA                   TAX
00130  C07A
00131  C07A                  STR_LOOP
00132  C07A 2   C8                    INY
00133  C07B 5*  B1 FC                LDA (DEST),Y
00134  C07D 6   20 D2 FF              JSR CHROUT
00135  C080 2   CA                    DEX
00136  C081 2*  D0 F7                BNE STR_LOOP
00137  C083
00138  C083 2   A9 0D                 LDA #$0D
00139  C085 6   20 D2 FF              JSR CHROUT
00140  C088
00141  C088 6   60                    RTS
00142  C089
00143  C089                  ;**************************************
00144  C089                  ;** VARIABILI:
00145  C089     00 00        OFFSET    WORD 0          ; DIFFERENZA DI PUNTATORI
00146  C08B                  ;**************************************
00147  C08B     00           DUMMY     BYTE 0
00148  C08C     3C 44 49     STR1      PTEXT "DIRTY HARRY (1971), MAGNUM FORCE (
00149  C0C9     2A 53 55     STR2      PTEXT "SUDDEN IMPACT (1983), THE DEAD POO
00150  C0F4     00 00 00     BUFFER    BYTES 255, $FF
00151  C2F2                  ;**************************************
```

Sperando che il lettore abbia apprezzato anche le citazioni dei titoli di alcuni *cult movies* utilizzate come esempi di stringhe, riteniamo con questo conclusa la sezione degli esempi relativi alle stringhe e, in generale, la presentazione di esempi di codice.

Capitolo 6

Conclusioni e ringraziamenti.

Dopo una prima parte meramente introduttiva e di presentazione dei concetti fondamentali e delle necessarie generalità, si sono contestualmente richiamati, sebbene in modo molto succinto, i concetti fondamentali dell'aritmetica binaria e della logica booleana. Si è quindi illustrata, nella seconda parte, l'architettura di massima della famiglia di CPU 65xx con i relativi principi di progettazione e programmazione. Si è dato conto di tutte le istruzioni documentate della ISA, con relativi tempi di esecuzione, modalità di indirizzamento supportate, codifica binaria, flag modificati. Si è poi passati ad illustrare oltre venti esempi concreti di codifica Assembly, idealmente slegati da una specifica architettura e come tali utilizzabili su una vastissima gamma di piattaforme hardware, dalle più semplici alle più evolute. Tutti gli esempi sono stati testati su emulatore e macchine reali dopo l'assemblaggio con CBM Prg Studio. Si sono trattati gli idiomi più importanti, senza la benché minima pretesa di esaustività, anzi consapevolmente scartando numerosi possibili esempi di programmazione avanzata: I/O su periferiche di interfaccia MMI (Man-Machine Interface) e di memorizzazione di massa, protocolli seriali, gestione di interrupt mascherabili e non, grafica, suono. Tali esempi potranno essere oggetto di una seconda trattazione separata, che dovrà essere necessariamente specifica per una architettura o famiglia di architetture (tipicamente Commodore 64, in quanto in assoluto il più venduto home computer della storia).

L'Autore si augura che lo *Sforzo* compiuto per rendere questo lavoro il più possibile corretto e coerente nella forma e nella sostanza corrisponda ad una piacevole ed istruttiva lettura per quanti, per i più vari motivi, desideravano una guida introduttiva in lingua italiana che fosse comprensibile senza rinunciare a spunti più avanzati, e sempre rimandando (senza appesantire la trattazione e le notazioni) a quei concetti elementari e teoremi di logica e matematica discreta che rendono più comprensibili le scelte di fondo dei progettisti, i limiti dei vari dimensionamenti di bus e registri e la logica di funzionamento delle istruzioni e degli idiomi di base.

In chiusura, l'Autore desidera ringraziare sentitamente tutti gli autori accademici qui citati, direttamente e indirettamente, per i loro contributi.
L'Autore ringrazia anche Dave Brubeck, Larry Carlton, John Coltrane, Paul Desmond, Danny Gatton, Allan Holdsworth, Greg Howe, Illinois Jacquet, Waylon Jennings, Eric Johnson, Paul Kossoff e i Free, Tony MacAlpine, Frank Marino e i Mahogany Rush, Mike Stern, Joey Tafolla, Pat Travers, Carlos Santana, Stevie Ray Vaughan, Johnny Winter, i Deep Purple, gli Uriah Heep nonché Johann Sebastian Bach, Ludwig Van Beethoven, Frédéric Chopin, Wolfgang Amadeus Mozart, Gioacchino Rossini, Antonio Vivaldi e Richard Wagner per l'immortale e variegata colonna sonora che, accompagnandolo nei suoi viaggi, ha sovente fatto da sottofondo alla stesura del presente testo.

Stante la tipica volatilità di Internet, l'autore declina ogni e qualsiasi responsabilità in ordine all'accuratezza dei link a siti di terze parti qui riportati e ai loro contenuti, da considerarsi valevoli e pregnanti unicamente al momento della consultazione effettuata durante la stesura del presente lavoro.

Bibliografia

[And85] Mark Andrews, *Commodore 64/128 assembly language programming*, H.W. Sams, 1985.

[BJ66] Corrado Boehm and Giuseppe Jacopini, *Flow diagrams, turing machines and languages with only two formation rules*, Commun. ACM **9** (1966), no. 5, 366–371.

[But86] Jim Butterfield, *Machine language for the commodore 64, 128, and other commodore computers*, Prentice Hall Press, New York, N.Y, 1986.

[Dav84] Danny Davies, *Commodore 64 machine language for the absolute beginner*, Imprint unknown, 1984.

[Dub78] J. M. Dubbey, *The mathematical work of charles babbage*, Cambridge University Press, Cambridge, 1978.

[Eng85] Lothar Englisch, *The advanced machine language book for the commodore 64*, Abacus Software Inc, 1985.

[IEE87] *Ieee standard for radix-independent floating-point arithmetic*, ANSI/IEEE Std 854-1987 (1987), 1–19.

[Jon84] Marvin L. De Jong, *Assembly language programming with the commodore 64*, Brady, 1984.

[Knu73] Donald Knuth, *The art of computer programming, volume 2*, Addison-Wesley Pub. Co, Reading, Mass, 1973.

[Knu97] Donald E. Knuth, *The art of computer programming, volume 3 (3rd ed.): sorting and searching*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1997.

[Lev86] Lance A. Leventhal, *6502 assembly language programming*, McGraw-Hill Osborne Media, 1986.

[Mak65] David C. Makinson, *The paradox of the preface*, Analysis **25** (1965), no. 6, 205–207.

[Osb80] Adam Osborne, *An introduction to microcomputers - volume 1*, Osborne/McGraw-Hill, Berkeley, Calif, 1980.

[Pea77] John Peatman, *Microcomputer-based design*, McGraw-Hill, New York, 1977.

[SH11] Maureen Sprankle and Jim Hubbard, *Problem solving and programming concepts (9th edition)*, Pearson, 2011.

[Sin84] Ian Robertson Sinclair, *Introducing commodore 64 machine code*, Prentice-Hall, 1984.

[Smi85] Bruce Smith, *Commodore 64 assembly language*, Chapman and Hall, 1985.

[Sut85] James Sutton, *Power programming the commodore 64: Assembly language, graphics, and sound*, Prentice-Hall, 1985.

[War13] Henry Warren, *Hacker's delight*, second ed., Addison-Wesley, Upper Saddle River, NJ, 2013.

[Zak82] Rodnay Zaks, *Advanced 6502 programming*, Sybex, Berkeley, 1982.

[Zak83] Rodnay Zaks, *Programming the 6502*, SYBEX, 1983.

Indice

ALLEGATI

- Datasheet CPU MOS 6510 (10 pagine)
- Datasheet CPU MOS 6510 – novembre 1982 (10 pagine)
- Timing table ISA 6502, in formato A3

COMMODORE SEMICONDUCTOR GROUP
a division of Commodore Business Machines, Inc.
950 Rittenhouse Road, Norristown, PA 19403 • 215/666-7950 • TWX 510-660-4168

HMOS

6510 MICROPROCESSOR WITH I/O

DESCRIPTION

The 6510 is a low-cost microprocessor capable of solving a broad range of small-systems and peripheral-control problems at minimum cost to the user.

An 8-bit Bi-Directional I/O Port is located on-chip with the Output Register at Address 0001 and the Data-Direction Register at Address 0000. The I/O Port is bit-by-bit programmable.

The Three-State sixteen-bit Address Bus allows Direct Memory Accessing (DMA) and multi-processor systems sharing a common memory.

The internal processor architecture is identical to the Commodore Semiconductor Group 6502 to provide software compatibility.

FEATURES OF THE 6510 . . .

- 8-Bit Bi-Directional I/O Port
- Single +5 volt supply
- HMOS, silicon gate, depletion load technology
- Eight bit parallel processing
- 56 Instructions
- Decimal and binary arithmetic
- Thirteen addressing modes
- True indexing capability
- Programmable stack pointer
- Variable length stack
- Interrupt capability
- 8 Bit Bi-Directional Data Bus
- Addressable memory range of up to 65K bytes
- Direct memory access capability
- Bus compatible with M6800
- Pipeline architecture
- 1 MHz, 2MHz and 3 MHz operation
- Use with any type or speed memory
- 4 MHz operation availability expected in 1986.

PIN CONFIGURATIONS

6510

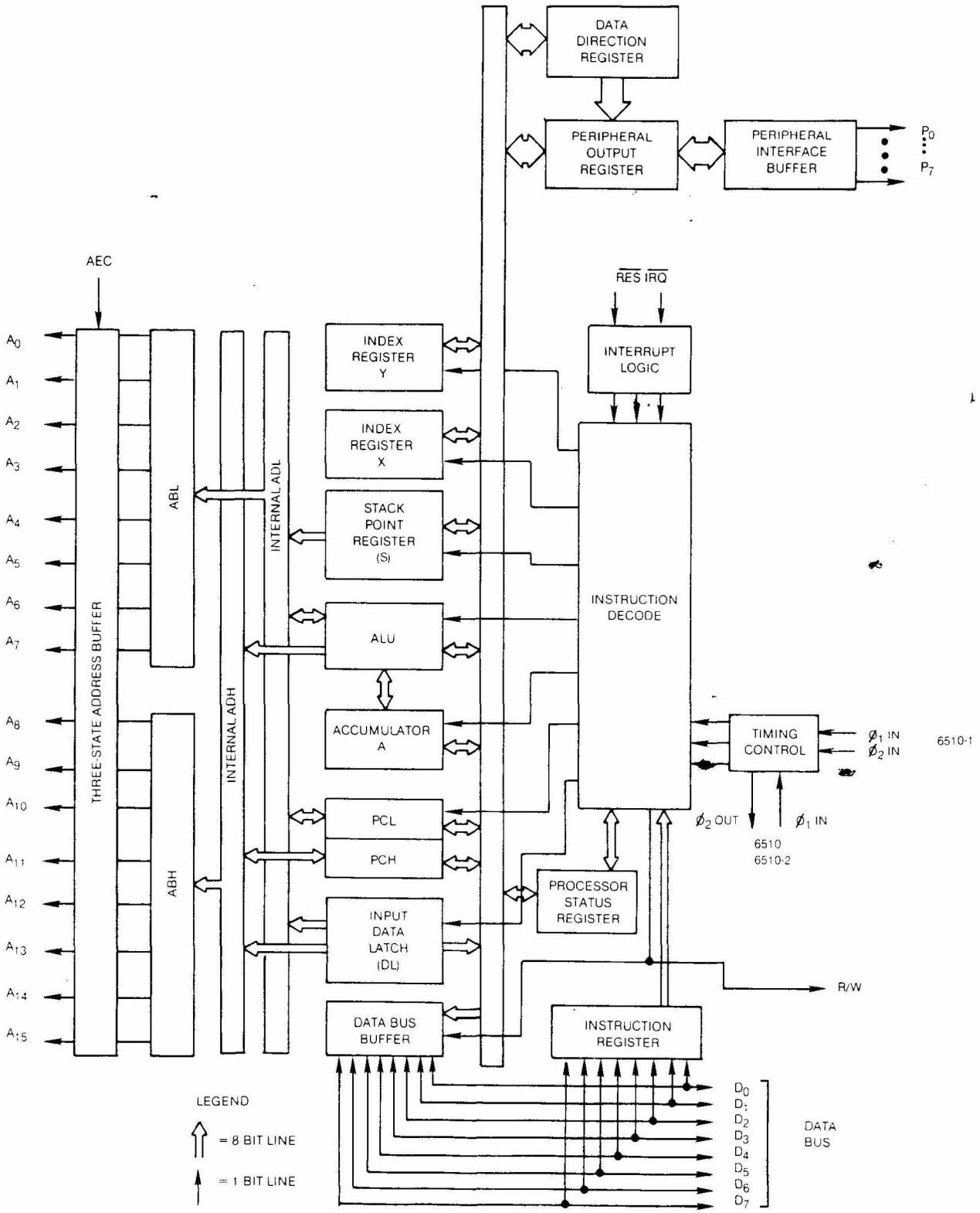
ϕ_1 IN	1	40	$\overline{RES}$
RDY	2	39	ϕ_2 OUT
$\overline{IRQ}$	3	38	R/W
$\overline{NMI}$	4	37	DB_0
AEC	5	36	DB_1
V_{CC}	6	35	DB_2
A_0	7	34	DB_3
A_1	8	33	DB_4
A_2	9	32	DB_5
A_3	10	31	DB_6
A_4	11	30	DB_7
A_5	12	29	P_0
A_6	13	28	P_1
A_7	14	27	P_2
A_8	15	26	P_3
A_9	16	25	P_4
A_{10}	17	24	P_5
A_{11}	18	23	A_{15}
A_{12}	19	22	A_{14}
A_{13}	20	21	V_{SS}

6510-1

$\overline{RES}$	1	40	ϕ_2IN
ϕ_1 IN	2	39	R/W
$\overline{IRQ}$	3	38	DB_0
AEC	4	37	DB_1
V_{CC}	5	36	DB_2
A_0	6	35	DB_3
A_1	7	34	DB_4
A_2	8	33	DB_5
A_3	9	32	DB_6
A_4	10	31	DB_7
A_5	11	30	P_0
A_6	12	29	P_1
A_7	13	28	P_2
A_8	14	27	P_3
A_9	15	26	P_4
A_{10}	16	25	P_5
A_{11}	17	24	P_6
A_{12}	18	23	P_7
A_{13}	19	22	A_{15}
V_{SS}	20	21	A_{14}

6510-2

$\overline{RES}$	1	40	ϕ_2 OUT
ϕ_2 IN	2	39	R/W
$\overline{IRQ}$	3	38	DB_0
AEC	4	37	DB_1
V_{CC}	5	36	DB_2
A_0	6	35	DB_3
A_1	7	34	DB_4
A_2	8	33	DB_5
A_3	9	32	DB_6
A_4	10	31	DB_7
A_5	11	30	P_0
A_6	12	29	P_1
A_7	13	28	P_2
A_8	14	27	P_3
A_9	15	26	P_4
A_{10}	16	25	P_5
A_{11}	17	24	P_6
A_{12}	18	23	P_7
A_{13}	19	22	A_{15}
V_{SS}	20	21	A_{14}



6510 BLOCK DIAGRAM

6510 CHARACTERISTICS

MAXIMUM RATINGS

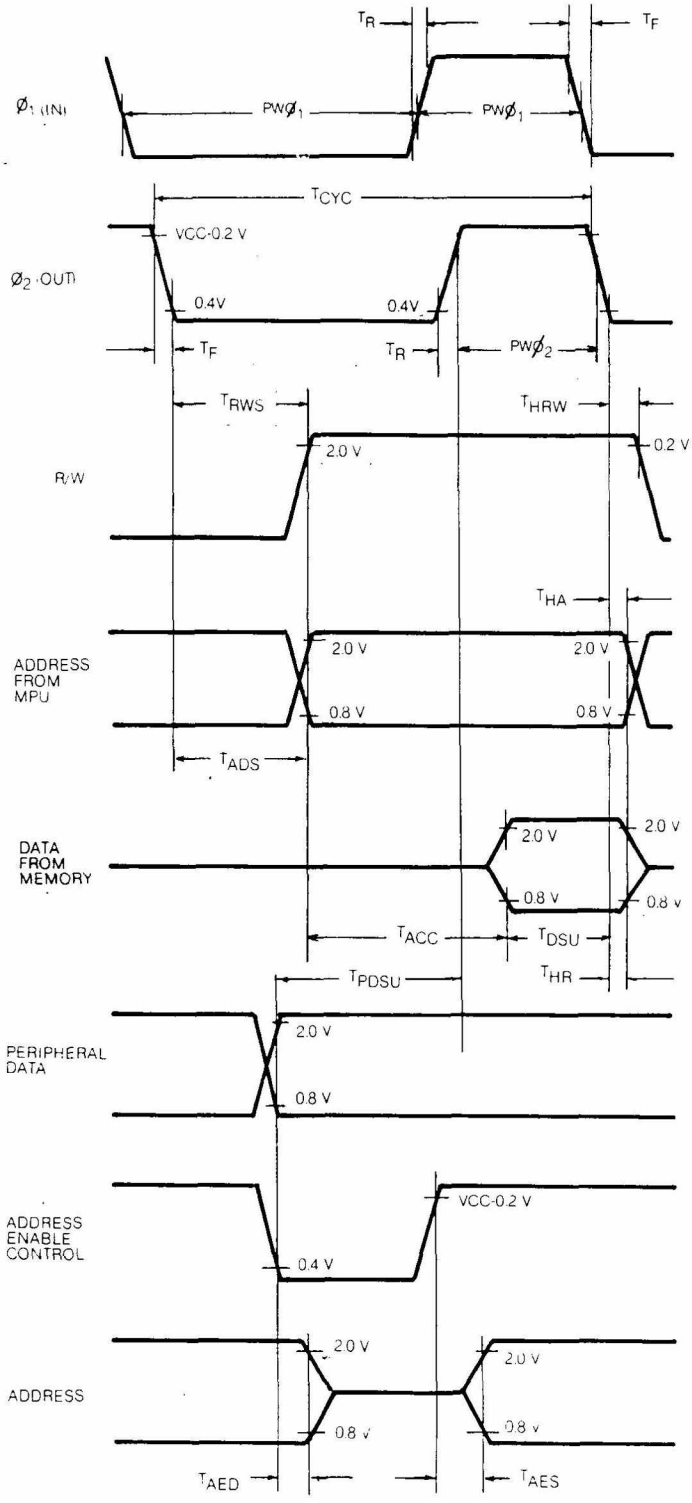
RATING	SYMBOL	VALUE	UNIT
SUPPLY VOLTAGE	V_{CC}	−0.3 to + 7.0	Vdc
INPUT VOLTAGE	Vin	−0.3 to + 7.0	Vdc
OPERATING TEMPERATURE	T_A	0 to + 70	C
STORAGE TEMPERATURE	T_{STG}	−55 to + 150	C

This device contains input protection against damage due to high static voltages or electric fields; however, precautions should be taken to avoid application of voltages higher than the maximum rating.

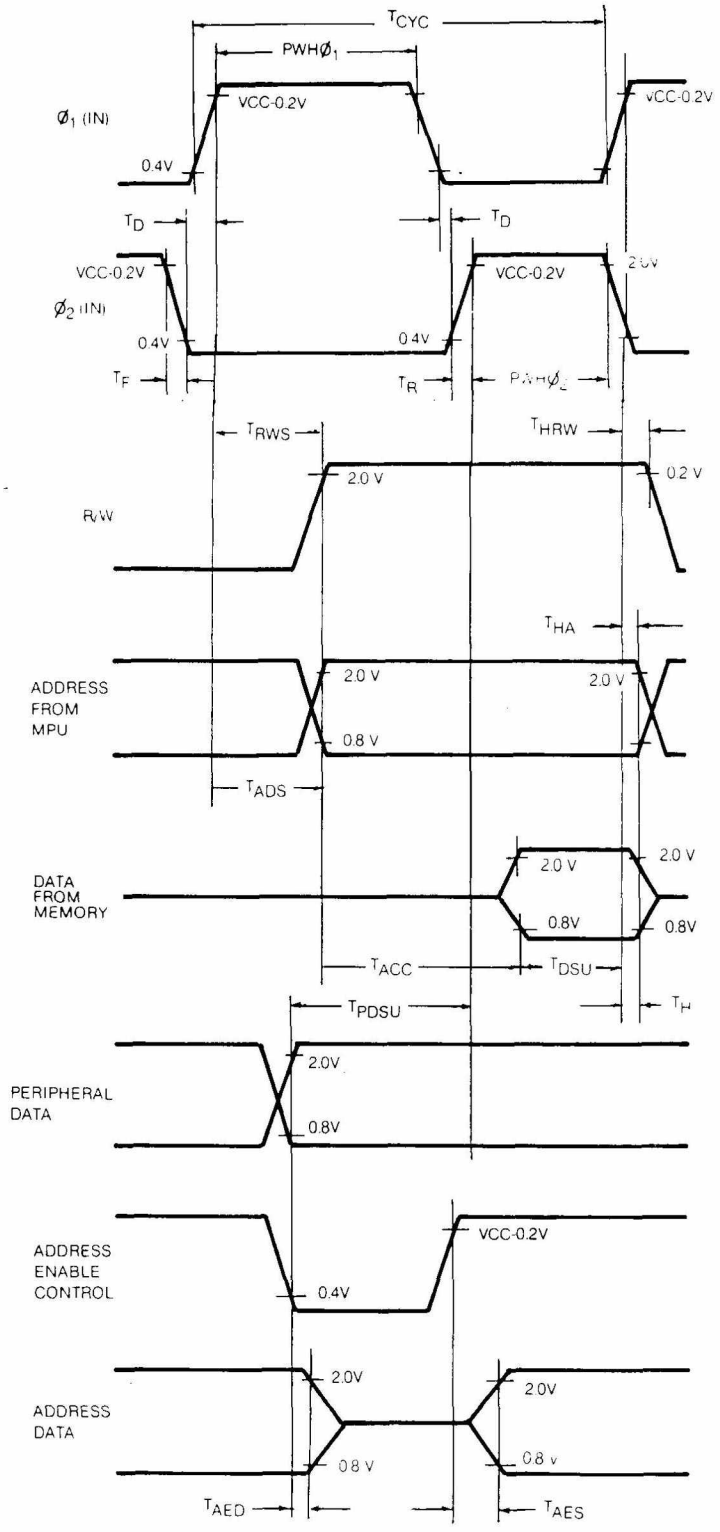
ELECTRICAL CHARACTERISTICS (Vcc = 5.0V ± 5%, Vss = 0, T_A = 0° to + 70°C)

CHARACTERISTIC	SYMBOL	MIN.	TYP.	MAX.	UNIT
Input High Voltage $Ø_1$, $Ø_2$(in) — 6510-1 $Ø_1$ (in) — 6510, 6510-2 $\overline{RES}$, P_0-P_7 $\overline{IRQ}$, Data	VIH	V_{CC}- 0.2 2.4 2.0	— — —	Vcc + 1.0V — —	Vdc Vdc Vdc
Input Low Voltage $Ø_1$ (in) — 6510, 6510-2 $Ø_1$, $Ø_2$(in) — 6510-1 $\overline{RES}$, P_0-P_7 $\overline{IRQ}$, Data	VIL	— — —	— — —	0.4 0.2 0.8	Vdc Vdc Vdc
Input Leakage Current (V_{in} = 0 to 5.25V, Vcc = 5.25V) Logic $Ø_1$, $Ø_2$(in)	Iin	— —	— —	2.5 100	μA μA
Three State (Off State) Input Current (V_{in}=0.4 to 2.4V, Vcc = 5.25V) DB_0-DB_7, A_0-A_{15}, R/W	ITSI	—	—	10	μA
Output High Voltage (I_{OH} = −100μAdc, Vcc = 4.75V) Data, A0-A15, R/W, P_0-P_7	VOH	2.4	—	—	Vdc
Out Low Voltage (I_{OL} = 1.6mAdc, Vcc = 4.75V) Data, A0-A15, R/W, P_0-P_7	VOL	—	—	0.5	Vdc
Power Supply Current	ICC	—	—	130	mA
Capacitance (V_{in} = 0, T_A = 25°C, f = 1 MHz) Logic, P_0-P_7 Data A0-A15, R/W $Ø_1$ $Ø_2$	C C_{in} C_{out} $C_{Ø1}$ $C_{Ø2}$	— — — — —	— — — 30 50	10 15 12 50 80	pF

6510, 6510-2
Internal Clock Format

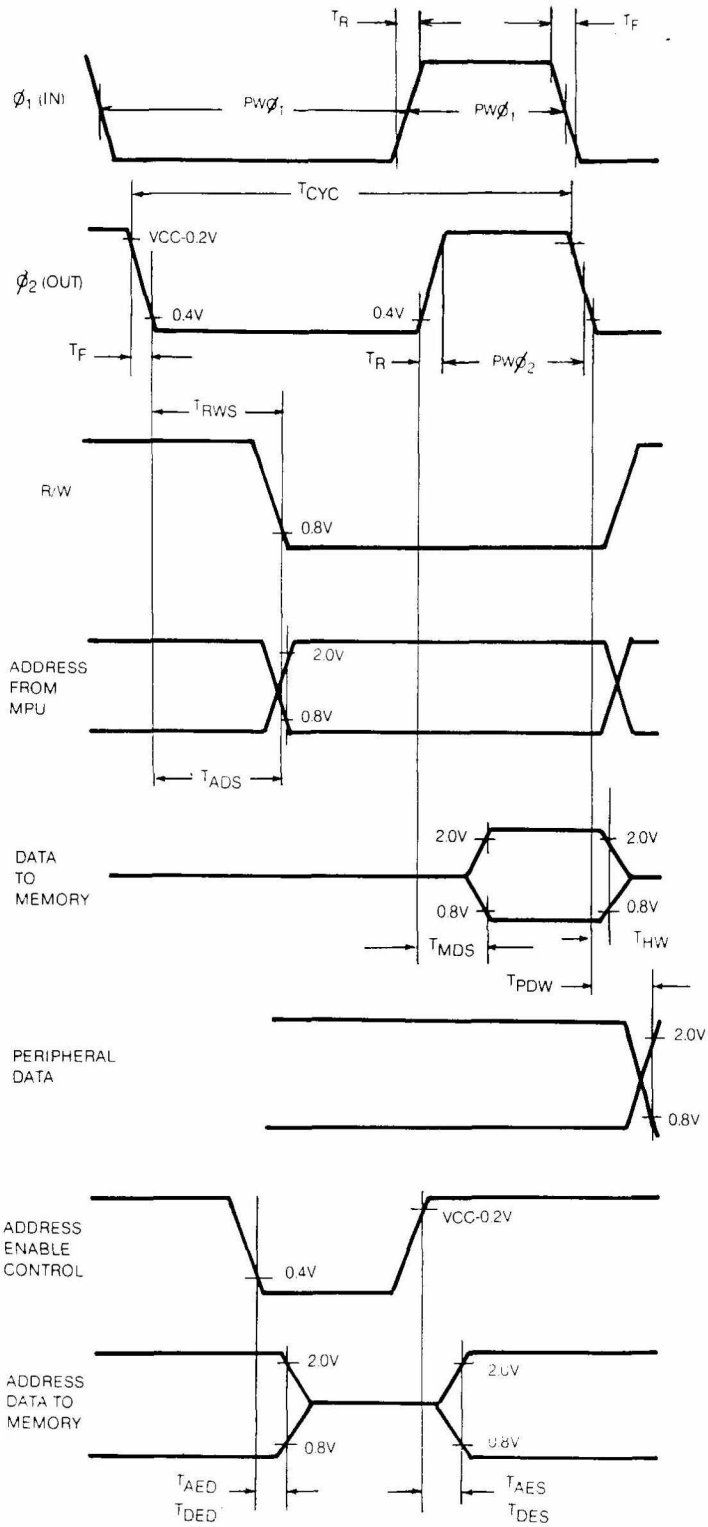


6510-1
Two Phase Clock Input Format

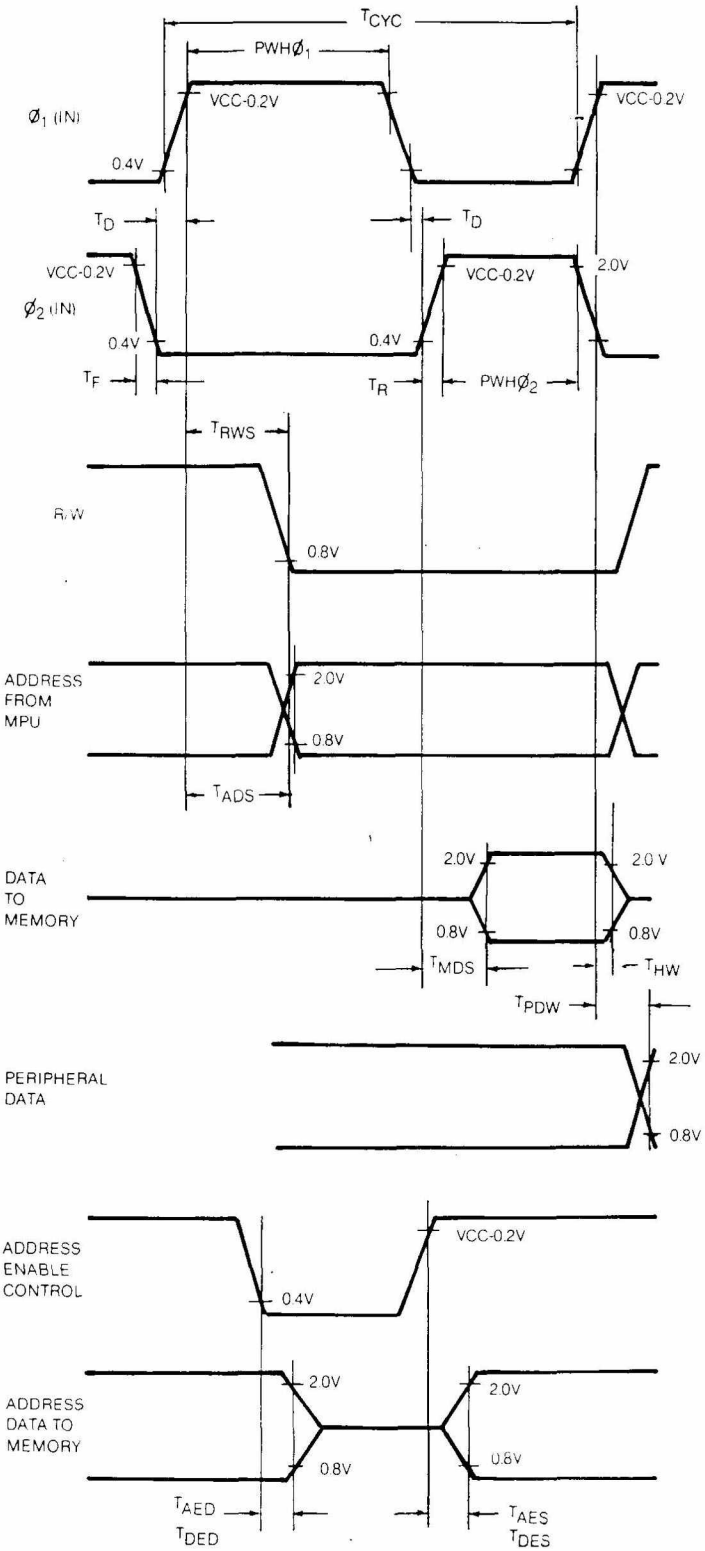


TIMING FOR READING DATA FROM MEMORY OR PERIPHERALS

6510, 6510-2
Internal Clock Format



6510-1
Two Phase Clock Input Format



TIMING FOR WRITING DATA TO MEMORY OR PERIPHERALS

AC CHARACTERISTICS

1 MHz TIMING | **2 MHz TIMING** | **3 MHz TIMING**

ELECTRICAL CHARACTERISTICS (V_{CC} = 5V ± 5%, V_{SS} = 0V, T_A = 0 –70 C)
Minimum Clock Frequency = 50 KHz

CLOCK TIMING

CHARACTERISTIC	SYMBOL	MIN.	TYP.	MAX.	MIN.	TYP.	MAX.	MIN.	TYP.	MAX.	UNITS
Cycle Time	T_{CYC}	1000	—	—	500	—	—	333	—	—	ns
Clock Pulse Width Ø1 IN (Measured at VCC-0.2V) Ø2 IN	PWHØ1 PWHØ2	430 470	— —	— —	215 235	— —	— —	150 160	— —	— —	ns ns
Fall Time, Rise Time Ø1 IN, Ø2 IN (Measured from 0.2V to VCC-0.2V)	T_F, T_R	—	—	10	—	—	10	—	—	10	ns
Delay Time between Clocks (Measured at 0.2V) 6510-1	T_D	0	—	—	0	—	—	0	—	—	ns
Ø1 in Pulse Width (Measured at 1.5V)	PWØ1	460	—	520	240	—	260	170	—	180	ns
Ø2 OUT Pulse Width* (Measured at 1.5V)	PWØ2	420	—	510	200	—	250	130	—	170	ns
Ø2 OUT Rise, Fall Time (Measured 0.4V to 2.0V)*	T_R, T_F	—	—	25	—	—	25	—	—	25	ns

READING/WRITE TIMING (LOAD=1 TTL)

CHARACTERISTIC	SYMBOL	MIN.	TYP.	MAX.	MIN.	TYP.	MAX.	MIN.	TYP.	MAX.	UNITS
Read/Write Setup Time from 6510	T_{RWS}	—	100	300	—	100	150	—	100	110	ns
Address Setup Time from 6510	T_{ADS}	—	100	300	—	100	150	—	100	125	ns
Memory Read Access Time	T_{ACC}	—	—	575	—	—	300	—	—	170	ns
Data Stability Time Period	T_{DSU}	100	—	—	60	—	—	40	—	—	ns
Data Hold Time-Read	T_{HR}	10	—	—	10	—	—	10	—	—	ns
Data Hold Time-Write	T_{HW}	10	30	—	10	30	—	10	30	—	ns
Data Setup Time from 6510	T_{MDS}	—	150	200	—	75	100	—	75	90	ns
Address Hold Time	T_{HA}	10	30	—	10	30	—	10	30	—	ns
R/W Hold Time	T_{HRW}	10	30	—	10	30	—	10	30	—	ns
Delay Time, Ø2 negative transition to Peripheral Data valid	T_{PDW}	—	—	300	—	—	150	—	—	125	ns
Peripheral Data Setup Time	T_{PDSU}	300	—	—	150	—	—	100	—	—	ns
Address Enable Setup Time	T_{AES}	—	—	75	—	—	75	—	—	75	ns
Data Enable Setup Time	T_{DES}	—	—	120	—	—	120	—	—	120	ns
Address Disable Hold Time*	T_{AED}	—	—	120	—	—	120	—	—	120	ns
Data Disable Hold Time*	T_{DED}	—	—	130	—	—	130	—	—	130	ns
Peripheral Data Hold Time	T_{PDH}	30	—	—	20	—	—	10	—	—	ns

*Note — 1 TTL Load, CL=30 pF

SIGNAL DESCRIPTION

Clocks ($\emptyset_1$, $\emptyset_2$)

The 6510 requires either a two phase non-overlapping clock that runs at the Vcc voltage level, or an external control for the internal clock generator.

Address Bus (A_0-A_{15})

The three state outputs are TTL compatible, capable of driving one standard TTL load and 130 pf.

Data Bus (D_0-D_7)

Eight pins are used for the data bus. This is a Bi-Directional bus, transferring data to and from the device and peripherals. The outputs are tri-state buffers capable of driving one standard TTL load and 130 pf.

Reset

This input is used to reset or start the microprocessor from a power down condition. During the time that this line is held low, writing to or from the microprocessor is inhibited. When a positive edge is detected on the input, the microprocessor will immediately begin the reset sequence.

After a system initialization time of six clock cycles, the mask interrupt flag will be set and the microprocessor will load the program counter from the memory vector locations FFFC and FFFD. This is the start location for program control.

After Vcc reaches 4.75 volts in a power up routine, reset must be held low for at least two clock cycles. At this time the R/W signal will become valid.

When the reset signal goes high following these two clock cycles, the microprocessor will proceed with the normal reset procedure detailed above.

Interrupt Request ($\overline{IRQ}$)

This TTL level input requests that an interrupt sequence begin within the microprocessor. The micorprocessor will complete the current instruction being executed before recognizing the request. At that time, the interrupt mask bit in the Status Code Register will be examined. If the interrupt mask flag is not set, the microprocessor will begin an interrupt sequence. The Program Counter and Processor Status Register are stored in the stack. The microprocessor will then set the interrupt mask flag high so that no further interrupts may occur. At the end of this cycle, the program counter low will be loaded from address FFFE, and program counter high from location FFFF, therefore transferring program control to the memory vector located at these addresses.

Address Enable Control (AEC)

The Address Bus, R/W, and Data Bus are valid only when the Address Enable Control line is high. When low, the Address Bus, R/W and Data Bus are in a high-impedance state. This feature allows easy DMA and multiprocessor systems.

I/O Port (P_0-P_7)

Eight pins are used for the peripheral port, which can transfer data to or from peripheral devices. The Output Register is located in RAM at Address 0001, and the Data Direction Register is at Address 0000. The outputs are capable at driving one standard TTL load and 130 pf.

Read/Write (R/W)

This signal is generated by the microprocessor to control the direction of data transfers on the Data Bus. This line is high except when the microprocessor is writing to memory or a peripheral device.

ADDRESSING MODES

ACCUMULATOR ADDRESSING — This form of addressing is represented with a one byte instruction, implying an operation on the accumulator.

IMMEDIATE ADDRESSING — In immediate addressing, the operand is contained in the second byte of the instruction, with no further memory addressing required.

ABSOLUTE ADDRESSING — In absolute addressing, the second byte of the instruction specifies the eight low order bits of the effective address while the third byte specifies the eight high order bits. Thus, the absolute addressing mode allows access to *the entire 65K bytes of addressable memory.*

ZERO PAGE ADDRESSING — The zero page instructions allow for shorter code and execution times by only fetching the second byte of the instruction and assuming a zero high address byte. Careful use of the zero page can result in significant increase in code efficiency.

INDEXED ZERO PAGE ADDRESSING — (X, Y indexing) — This form of addressing is used in conjunction with the index register and is referred to as "Zero Page, X" or "Zero Page, Y." The effective address is calculated by adding the second byte to the contents of the index register. Since this is a form of "Zero Page" addressing, the content of the second byte references a location in page zero. Additionally, due to the "Zero Page" addressing nature of this mode, no carry is added to the high order 8 bits of memory and crossing of page boundaries does not occur.

INDEX ABSOLUTE ADDRESSING — (X, Y indexing) — This form of addressing is used in conjunction with X and Y index register and is referred to as "Absolute, X," and "Absolute, Y." The effective address is formed by adding the contents of X and Y to the address contained in the second and third bytes of the instruction. This mode allows the index register to contain the index or count value and the instruction to contain the base address. This type of indexing allows any location referencing and the index to modify multiple fields resulting in reduced coding and execution time.

IMPLIED ADDRESSING — In the implied addressing mode, the address containing the operand is implicitly stated in the operation code of the instruction.

RELATIVE ADDRESSING — Relative addressing is used only with branch instructions and establishes a destination for the conditional branch.

The second byte of the instruction becomes the operand which is an "Offset" added to the contents of the lower eight bits of the program counter when the counter is set at the next instruction. The range of the offset is — 128 to + 127 bytes from the next instruction.

INDEXED INDIRECT ADDRESSING — In indexed indirect addressing (referred to as [Indirect, X]), the second byte of the instruction is added to the contents of the X index register, discarding the carry. The result of this addition points to a memory location on page zero whose contents is the low order eight bits of the effective address. The next memory location in page zero contains the high order eight bits of the effective address. Both memory locations specifying the high and low order bytes of the effective address must be in page zero.

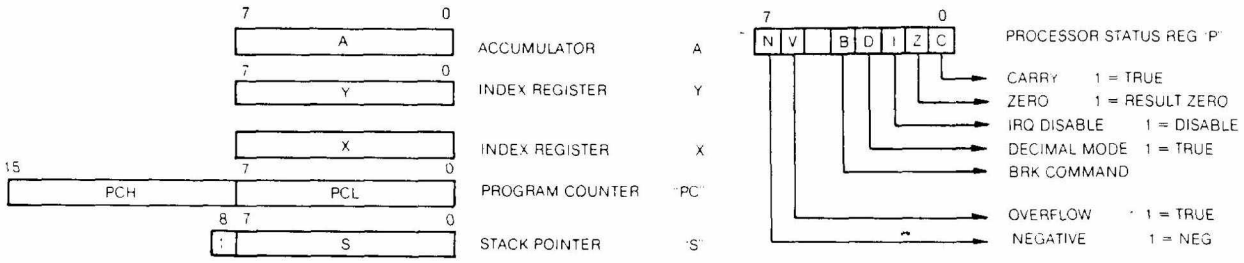
INDIRECT INDEXED ADDRESSING — In indirect indexed addressing (referred to as [Indirect, Y]), the second byte of the instruction points to a memory location in page zero. The contents of this memory location is added to the contents of the Y index register, the result being the low order eight bits of the effective address. The carry from this addition is added to the contents of the next page zero memory location, the result being the high order eight bits of the effective address.

ABSOLUTE INDIRECT — The second byte of the instruction contains the low order eight bits of a memory location. The high order eight bits of that memory location is contained in the third byte of the instruction. The contents of the fully specified memory location is the low order byte of the effective address. The next memory location contains the high order byte of the effective address which is loaded into the sixteen bits of the program counter.

INSTRUCTION SET — ALPHABETIC SEQUENCE

ADS	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift One Bit Right (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	*Pull Processor Status from Stack*
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Register
TYA	Transfer Index Y to Accumulator

PROGRAMMING MODEL



INSTRUCTION SET—OP CODES, Execution Time, Memory Requirements

MNEMONIC	OPERATION	IMMEDIATE OP	N	#	ABSOLUTE OP	N	#	ZERO PAGE OP	N	#	ACCUM. OP	N	#	IMPLIED OP	N	#	(IND. X) OP	N	#	(IND. Y OP	N	#	Z. PAGE, X OP	N	#	ABS. X OP	N	#	ABS. Y OP	N	#	RELATIVE OP	N	#	INDIRECT OP	N	#	Z. PAGE, Y OP	N	#	CONDITION CODES N	Z	C	I	D	V
ADC	A + M + C → A	69	2	2	6D	4	3	65	3	2							61	6	2	71	5	2	75	4	2	7D	4	3	79	4	3														
AND	A ∧ M → A	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2	35	4	2	3D	4	3	39	4	3														
ASL	C ← 7 0 ← 0				0E	6	3	06	5	2	0A	2	1										16	6	2	1E	7	3																
BCC	BRANCH ON C = 0																															90	2	2										
BCS	BRANCH ON C = 1																															B0	2	2										
BEQ	BRANCH ON Z = 1																															F0	2	2										
BIT	A ∧ M				2C	4	3	24	3	2																																		M6	M7
BMI	BRANCH ON N = 1																															30	2	2										
BNE	BRANCH ON Z = 0																															D0	2	2										
BPL	BRANCH ON N = 0																															10	2	2										
BRK														00	7	1																												
BVC	BRANCH ON V = 0																															50	2	2										
BVS	BRANCH ON V = 1																															70	2	2										
CLC	0 → C													18	2	1																												0	
CLD	0 → D													D8	2	1																													0
CLI	0 → I													58	2	1																												0	
CLV	0 → V													B8	2	1																													0
CMP	A − M	C9	2	2	CD	4	3	C5	3	2							C1	6	2	D1	5	2	D5	4	2	DD	4	3	D9	4	3														
CPX	X − M	E0	2	2	EC	4	3	E4	3	2																																			
CPY	Y − M	C0	2	2	CC	4	3	C4	3	2																																			
DEC	M − 1 → M				CE	6	3	C6	5	2													D6	6	2	DE	7	3																
DEX	X − 1 → X													CA	2	1																													
DEY	Y − 1 → Y													88	2	1																													
EOR	A ⊻ M → A	49	2	2	4D	4	3	45	3	2							41	6	2	51	5	2	55	4	2	5D	4	3	59	4	3														
INC	M + 1 → M				EE	6	3	E6	5	2													F6	6	2	FE	7	3																
INX	X + 1 → X													E8	2	1																													
INY	Y + 1 → Y													C8	2	1																													
JMP	JUMP TO NEW LOC				4C	3	3																												6C	5	3							
JSR	JUMP SUB				20	6	3																																					
LDA	M → A	A9	2	2	AD	4	3	A5	3	2							A1	6	2	B1	5	2	B5	4	2	BD	4	3	B9	4	3														

MNEMONIC	OPERATION	IMMEDIATE OP	N	#	ABSOLUTE OP	N	#	ZERO PAGE OP	N	#	ACCUM. OP	N	#	IMPLIED OP	N	#	(IND. X) OP	N	#	(IND. Y OP	N	#	Z. PAGE, X OP	N	#	ABS. X OP	N	#	ABS. Y OP	N	#	RELATIVE OP	N	#	INDIRECT OP	N	#	Z. PAGE, Y OP	N	#	CONDITION CODES N	Z	C	I	D	V
LDX	M → X	A2	2	2	AE	4	3	A6	3	2																			BE	4	3							B6	4	2						
LDY	M → Y	A0	2	2	AC	4	3	A4	3	2													B4	4	2	BC	4	3																	
LSR	0 → 7 0 → C				4E	6	3	46	5	2	4A	2	1										56	6	2	5E	7	3																
NOP	NO OPERATION													EA	2	1																													
ORA	A ∨ M → A	09	2	2	0D	4	3	05	3	2							01	6	2	11	5	2	15	4	2	1D	4	3	19	4	3														
PHA	A → Ms	S − 1 → S												48	3	1																													
PHP	P → Ms	S − 1 → S												08	3	1																													
PLA	S + 1 → S	Ms → A												68	4	1																													
PLP	S + 1 → S	Ms → P												28	4	1																													(RESTORED)
ROL	7 0 C				2E	6	3	26	5	2	2A	2	1										36	6	2	3E	7	3																
ROR	C 7 0				6E	6	3	66	5	2	6A	2	1										76	6	2	7E	7	3																
RTI	RTRN INT													40	6	1																													(RESTORED)
RTS	RTRN SUB													60	6	1																													
SBC	A − M − C → A	E9	2	2	ED	4	3	E5	3	2							E1	6	2	F1	5	2	F5	4	2	FD	4	3	F9	4	3														
SEC	1 → C													38	2	1																												1	
SED	1 → D													F8	2	1																													1
SEI	1 → I													78	2	1																													
STA	A → M				8D	4	3	85	3	2							81	6	2	91	6	2	95	4	2	9D	5	3	99	5	3														
STX	X → M				8E	4	3	86	3	2																													96	4	2						
STY	Y → M				8C	4	3	84	3	2													94	4	2																				
TAX	A → X													AA	2	1																													
TAY	A → Y													A8	2	1																													
TSX	S → X													BA	2	1																													
TXA	X → A													8A	2	1																													
TXS	X → S													9A	2	1																													
TYA	Y → A													98	2	1																													

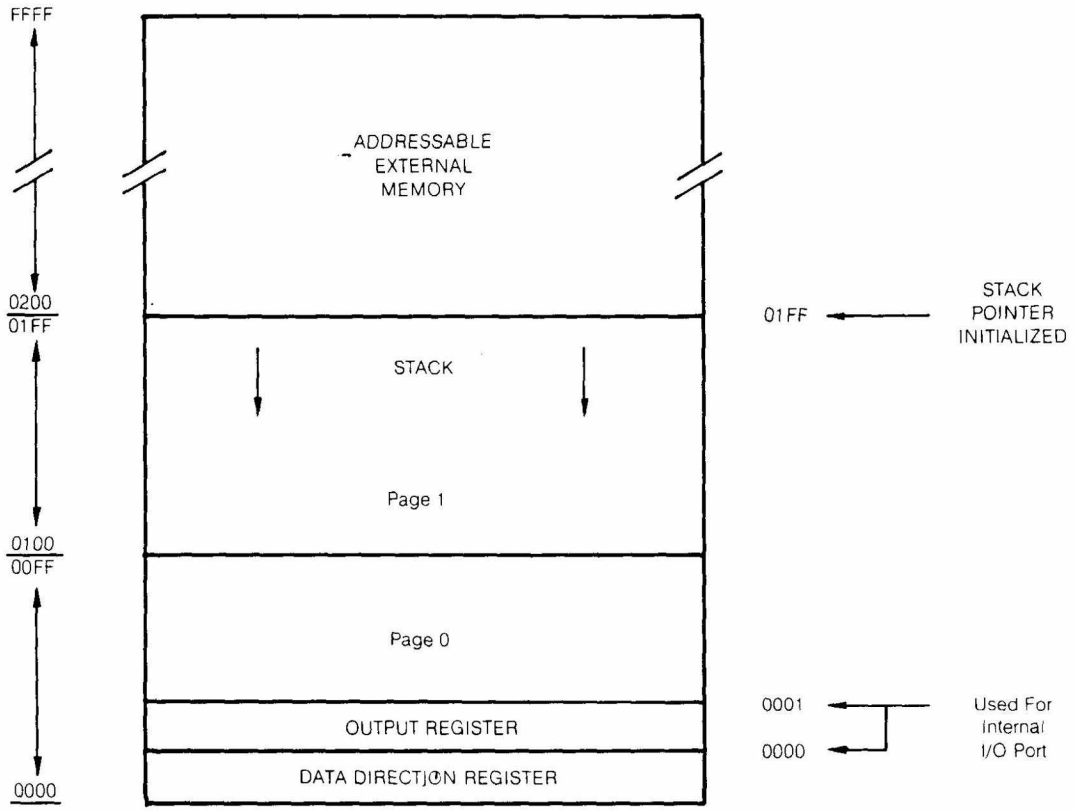
1. ADD 1 TO "N" IF PAGE BOUNDRY IS CROSSED
2. ADD 1 TO "N" IF BRANCH OCCURS TO SAME PAGE
   ADD 2 TO "N" IF BRANCH OCCURS TO DIFFERENT PAGE
3. CARRY NOT-BORROW
4. IF IN DECIMAL MODE Z FLAG IS INVALID.
   ACCUMULATOR MUST BE CHECKED FOR ZERO RESULT

X INDEX X
Y INDEX Y
A ACCUMULATOR
M MEMORY PER EFFECTIVE ADDRESS
Ms MEMORY PER STACK POINTER

+ ADD
− SUBTRACT
∧ AND
∨ OR
⊻ EXCLUSIVE OR
↑ MODIFIED
– NOT MODIFIED
M7 MEMORY BIT 7
M6 MEMORY BIT 6

N CYCLES
NO BYTES

Note: Commodore Semiconductor Group cannot assume liability for the use of undefined OP Codes



6510 MEMORY MAP

APPLICATIONS NOTES

Locating the Output Register at the internal I/O Port in Page Zero enhances the powerful Zero Page Addressing instructions of the 6510.

By assigning the I/O Pins as inputs (using the Data Direction Register) the user has the ability to change the contents of address 0001 (the Output Register) using peripheral devices. The ability to change these contents using peripheral inputs, together with Zero Page Indirect Addressing instructions, allows novel and versatile programming techniques not possible earlier.

commodore
mos technology
NMOS

950 Rittenhouse Rd., Norristown, PA 19403 • Tel.: 215/666-7950 • TWX: 510/660-4168

6510 MICROPROCESSOR WITH I/O

DESCRIPTION

The 6510 is a low-cost microcomputer system capable of solving a broad range of small-systems and peripheral-control problems at minimum cost to the user.

An 8-bit Bi-Directional I/O Port is located on-chip with the Output Register at Address 0001 and the Data-Direction Register at Address 0000. The I/O Port is bit-by-bit programmable.

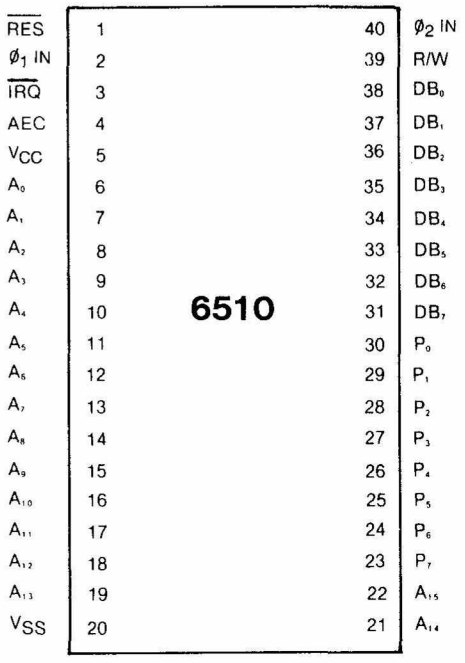
The Three-State sixteen-bit Address Bus allows Direct Memory Accessing (DMA) and multi-processor systems sharing a common memory.

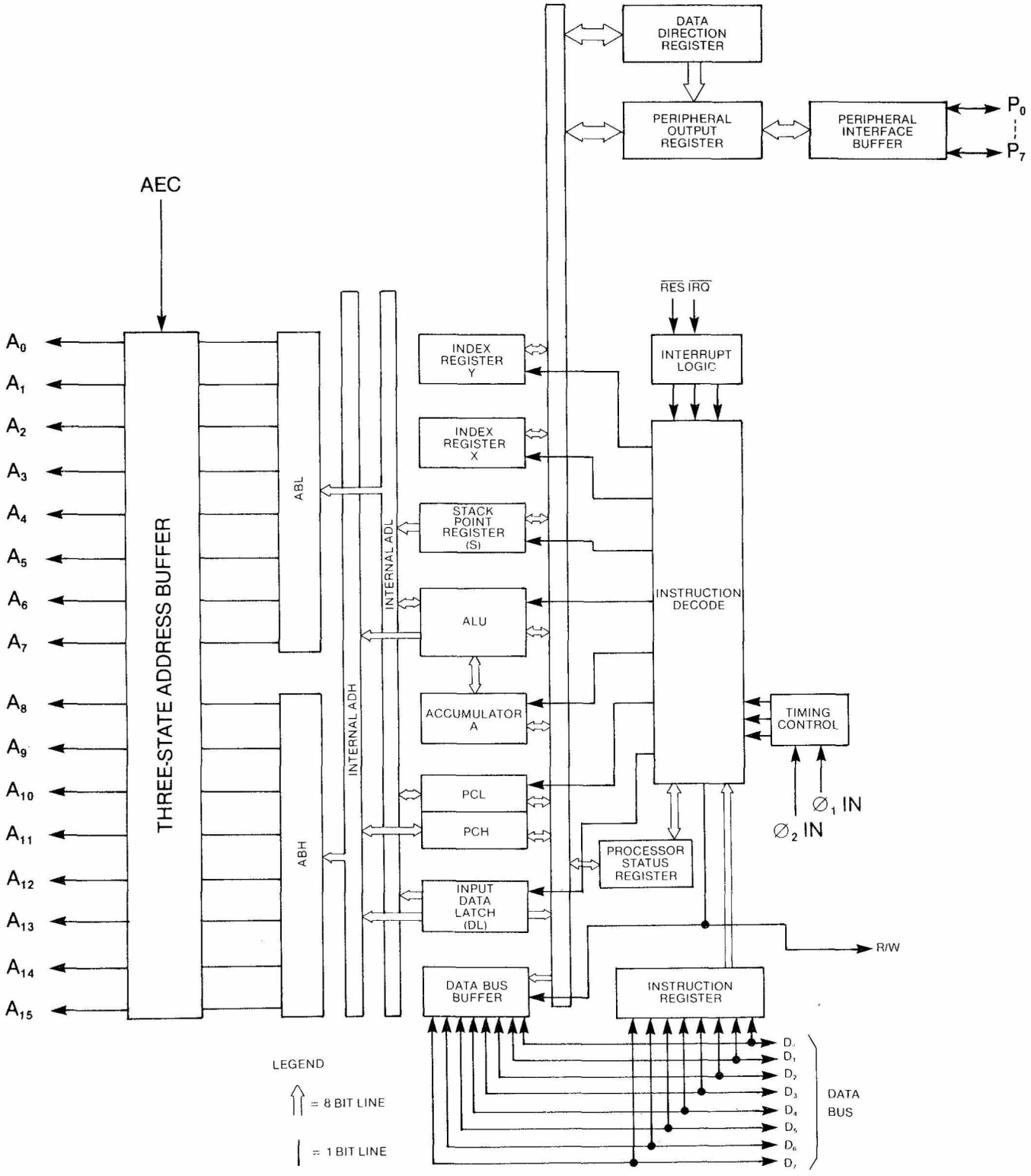
The internal processor architecture is identical to the MOS Technology 6502 to provide software compatibility.

FEATURES OF THE 6510 . . .

- 8-Bit Bi-Directional I/O Port
- 256 Bytes fully Static RAM (internal)
- Single +5 volt supply
- N channel, silicon gate, depletion load technology
- Eight bit parallel processing
- 56 Instructions
- Decimal and binary arithmetic
- Thirteen addressing modes
- True indexing capability
- Programmable stack pointer
- Variable length stack
- Interrupt capability
- 8 Bit Bi-Directional Data Bus
- Addressable memory range of up to 65K bytes
- Direct memory access capability
- Bus compatible with M6800
- Pipeline architecture
- 1 MHz and 2MHz operation
- Use with any type or speed memory

PIN CONFIGURATION





6510 BLOCK DIAGRAM

6510 CHARACTERISTICS

MAXIMUM RATINGS

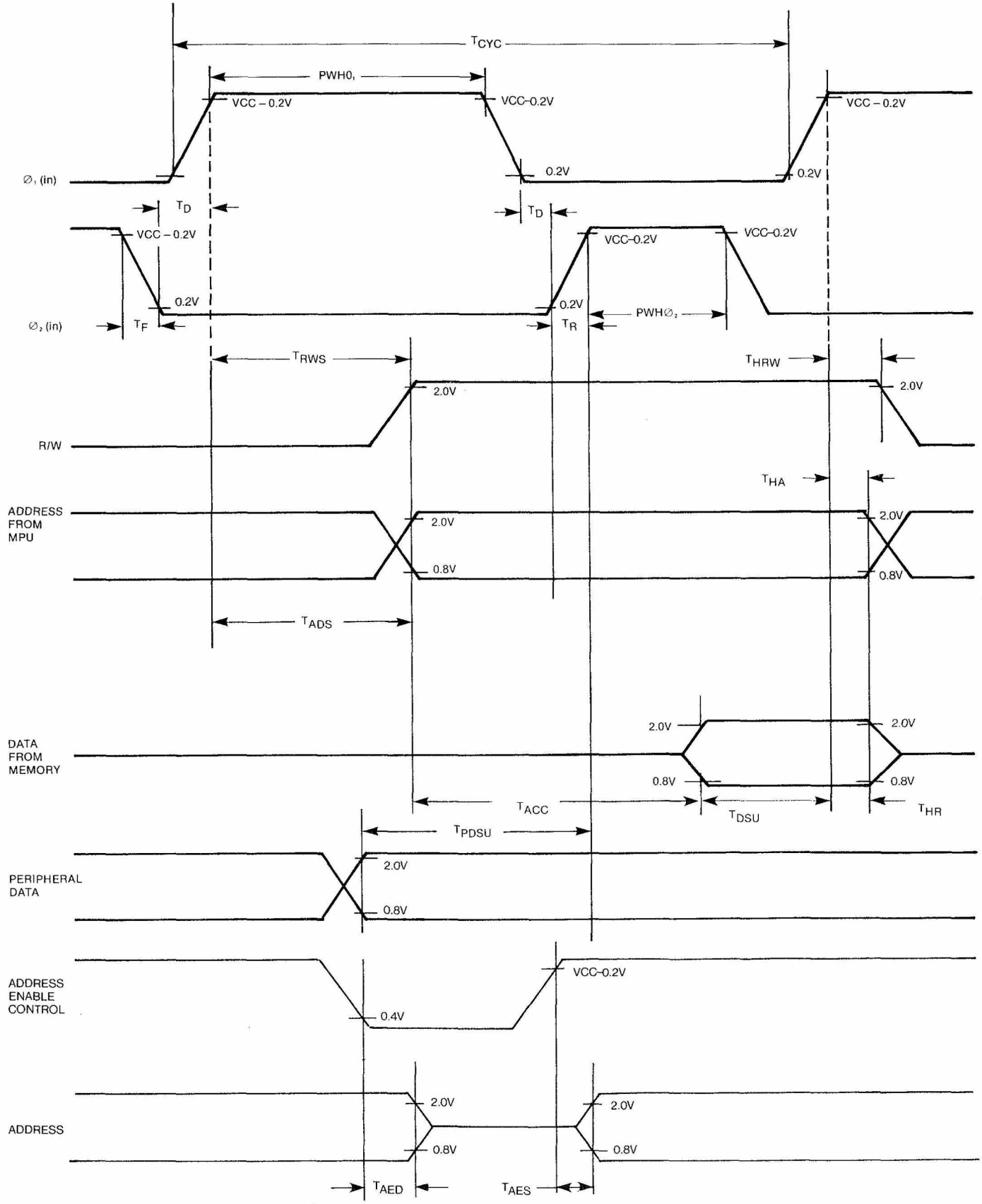
RATING	SYMBOL	VALUE	UNIT
SUPPLY VOLTAGE	Vcc	−0.3 to +7.0	Vdc
INPUT VOLTAGE	Vin	−0.3 to +7.0	Vdc
OPERATING TEMPERATURE	T_A	0 to +70	°C
STORAGE TEMPERATURE	T_{STG}	−55 to +150	°C

This device contains input protection against damage due to high static voltages or electric fields; however, precautions should be taken to avoid application of voltages higher than the maximum rating.

ELECTRICAL CHARACTERISTICS (Vcc = 5.0V ± 5%, Vss = 0, T_A = 0° to + 70°C)

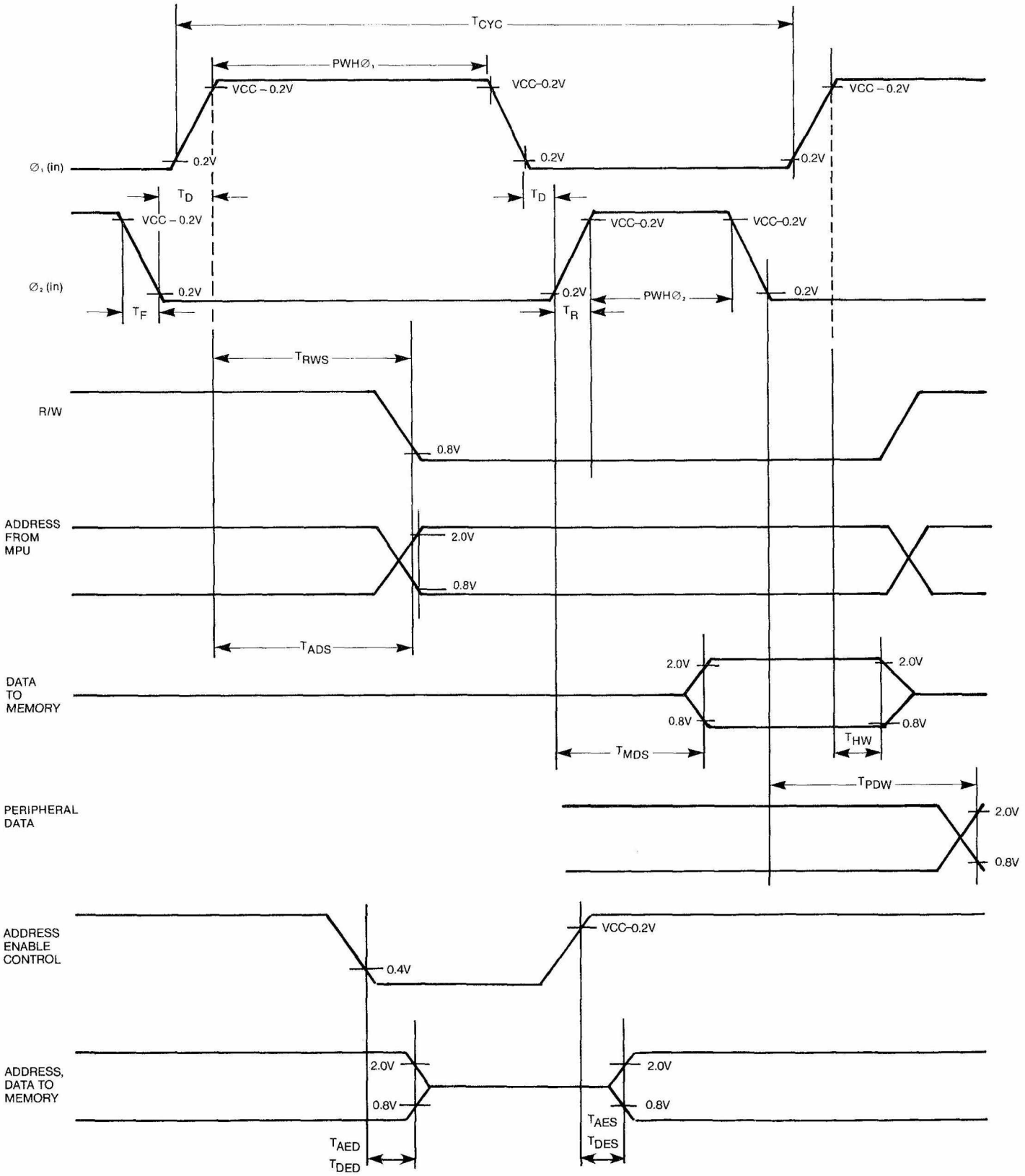
CHARACTERISTIC	SYMBOL	MIN.	TYP.	MAX.	UNIT
Input High Voltage					
$\varnothing_1$, $\varnothing_2$(in)	VIH	Vcc - 0.2	—	Vcc + 1.0V	Vdc
Input High Voltage					
$\overline{RES}$, P_0-P_7, $\overline{IRQ}$, Data		Vss + 2.0	—	—	Vdc
Input Low Voltage					
$\varnothing_1$, $\varnothing_2$(in)	VIL	Vss - 0.3	—	Vss + 0.2	Vdc
$\overline{RES}$, P_0-P_7, $\overline{IRQ}$, Data		—	—	Vss + 0.8	Vdc
Input Leakage Current (V_{in} = 0 to 5.25V, Vcc = 5.25V)					
Logic	Iin	—	—	2.5	μA
$\varnothing_1$, $\varnothing_2$(in)		—	—	100	μA
Three State (Off State) Input Current (V_{in} = 0.4 to 2.4V, Vcc = 5.25V)					
Data Lines	ITSI	—	—	10	μA
Output High Voltage (I_{OH} = −100μAdc, Vcc = 4.75V)					
Data, A0-A15, R/W, P_0-P_7	VOH	Vss + 2.4	—	—	Vdc
Out Low Voltage (I_{OL} = 1.6mAdc, Vcc = 4.75V)					
Data, A0-A15, R/W, P_0-P_7	VOL	—	—	Vss + 0.4	Vdc
Power Supply Current	ICC	—	125		mA
Capacitance	C				pF
V_{in} = O, T_A = 25°C, f = 1MHz)					
Logic, P_0-P_7	C_{in}	—	—	10	
Data		—	—	15	
A0-A15, R/W	C_{out}	—	—	12	
$\varnothing_1$	$C_{\varnothing_1}$	—	30	50	
$\varnothing_2$	$C_{\varnothing_2}$	—	50	80	

CLOCK TIMING



TIMING FOR READING DATA FROM MEMORY OR PERIPHERALS

CLOCK TIMING



TIMING FOR WRITING DATA TO MEMORY OR PERIPHERALS

AC CHARACTERISTICS

1 MHz TIMING	2 MHz TIMING

ELECTRICAL CHARACTERISTICS (VCC = 5V ± 5%, VSS = 0V, T_A = 0° –70°C)

CLOCK TIMING

CHARACTERISTIC	SYMBOL	MIN.	TYP.	MAX.	MIN.	TYP.	MAX.	UNITS
Cycle Time	T_{CYC}	1000	—	—	500	—	—	ns
Clock Pulse Width Ø1 (Measured at VCC – 0.2V) Ø2	PWHØ1 PWHØ2	430 470	— —	— —	215 235	— —	— —	ns ns
Fall Time, Rise Time (Measured from 0.2V to VCC–0.2V)	T_F, T_R	—	—	25	—	—	15	ns
Delay Time between Clocks (Measured at 0.2V)	T_D	0	—	—	0	—	—	ns

READ/WRITE TIMING (LOAD=1TTL)

CHARACTERISTIC	SYMBOL	MIN.	TYP.	MAX.	MIN.	TYP.	MAX.	UNITS
Read/Write Setup Time from 6508	T_{RWS}	—	100	300	—	100	150	ns
Address Setup Time from 6508	T_{ADS}	—	100	300	—	100	150	ns
Memory Read Access Time	T_{ACC}	—	—	575	—	—	300	ns
Data Stability Time Period	T_{DSU}	100	—	—	50			ns
Data Hold Time-Read	T_{HR}	10	—	—	10			ns
Data Hold Time-Write	T_{HW}	10	30	—	10	30		ns
Data Setup Time from 6510	T_{MDS}	—	150	200	—	75	100	ns
Address Hold Time	T_{HA}	10	30	—	10	30		ns
R/W Hold Time	T_{HRW}	10	30	—	10	30		ns
Delay Time, Ø2 negative transition to Peripheral Data valid	T_{PDW}	—	—	300			150	μs
Peripheral Data Setup Time	T_{PDSU}	300	—	—	150			ns
Address Enable Setup Time	T_{AES}			75			75	ns
Address Disable *See Note 1	T_{AED}			120			120	ns
Data Enable Setup Time	T_{DES}			120			120	ns
Data Disable *See Note 1	T_{DED}			130			130	ns

*Note 1 — 1TTL Load
CL = 30 pf

SIGNAL DESCRIPTION

Clocks ($\varnothing_1$, $\varnothing_2$)
The 6510 requires a two phase non-overlapping clock that runs at the Vcc voltage level.

Address Bus (A_0-A_{15})
The tri state outputs are TTL compatible, capable of driving one standard TTL load and 130 pf.

Data Bus (D_0-D_7)
Eight pins are used for the data bus. This is a Bi-Directional bus, transferring data to and from the device and peripherals. The outputs are tri-state buffers capable of driving one standard TTL load and 130 pf.

Reset
This input is used to reset or start the microprocessor from a power down condition. During the time that this line is held low, writing to or from the microprocessor is inhibited. When a positive edge is detected on the input, the microprocessor will immediately begin the reset sequence.

After a system initialization time of six clock cycles, the mask interrupt flag will be set and the microprocessor will load the program counter from the memory vector locations FFFC and FFFD. This is the start location for program control.

After Vcc reaches 4.75 volts in a power up routine, reset must be held low for at least two clock cycles. At this time the R/W signal will become valid.

When the reset signal goes high following these two clock cycles, the microprocessor will proceed with the normal reset procedure detailed above.

Interrupt Request ($\overline{IRQ}$)
This TTL level input requests that an interrupt sequence begin within the microprocessor. The microprocessor will complete the current instruction being executed before recognizing the request. At that time, the interrupt mask bit in the Status Code Register will be examined. If the interrupt mask flag is not set, the microprocessor will begin an interrupt sequence. The Program Counter and Processor Status Register are stored in the stack. The microprocessor will then set the interrupt mask flag high so that no further interrupts may occur. At the end of this cycle, the program counter low will be loaded from address FFFE, and program counter high from location FFFF, therefore transferring program control to the memory vector located at these addresses.

Address Enable Control (AEC)
The Address Bus is valid only when the Address Enable Control line is high. When low, the Address Bus is in a high-impedance state. This feature allows easy DMA and multiprocessor systems.

I/O Port (P_0-P_7)
Eight pins are used for the peripheral port, which can transfer data to or from peripheral devices. The Output Register is located in RAM at Address 0001, and the Data Direction Register is at Address 0000. The outputs are capable at driving one standard TTL load and 130 pf.

Read/Write (R/W)
This signal is generated by the microprocessor to control the direction of data transfers on the Data Bus. This line is high except when the microprocessor is writing to memory or a peripheral device.

ADDRESSING MODES

ACCUMULATOR ADDRESSING—This form of addressing is represented with a one byte instruction, implying an operation on the accumulator.

IMMEDIATE ADDRESSING—In immediate addressing, the operand is contained in the second byte of the instruction, with no further memory addressing required.

ABSOLUTE ADDRESSING—In absolute addressing, the second byte of the instruction specifies the eight low order bits of the effective address while the third byte specifies the eight high order bits. Thus, the absolute addressing mode allows access to the entire 65K bytes of addressable memory.

ZERO PAGE ADDRESSING—The zero page instructions allow for shorter code and execution times by only fetching the second byte of the instruction and assuming a zero high address byte. Careful use of the zero page can result in significant increase in code efficiency.

INDEXED ZERO PAGE ADDRESSING—(X, Y indexing)—This form of addressing is used in conjunction with the index register and is referred to as "Zero Page, X" or "Zero Page, Y." The effective address is calculated by adding the second byte to the contents of the index register. Since this is a form of "Zero Page" addressing, the content of the second byte references a location in page zero. Additionally, due to the "Zero Page" addressing nature of this mode, no carry is added to the high order 8 bits of memory and crossing of page boundaries does not occur.

INDEXED ABSOLUTE ADDRESSING—(X, Y indexing)—This form of addressing is used in conjunction with X and Y index register and is referred to as "Absolute, X," and "Absolute, Y." The effective address is formed by adding the contents of X and Y to the address contained in the second and third bytes of the instruction. This mode allows the index register to contain the index or count value and the instruction to contain the base address. This type of indexing allows any location referencing and the index to modify multiple fields resulting in reduced coding and execution time.

IMPLIED ADDRESSING—In the implied addressing mode, the address containing the operand is implicitly stated in the operation code of the instruction.

RELATIVE ADDRESSING—Relative addressing is used only with branch instructions and establishes a destination for the conditional branch.

The second byte of the instruction becomes the operand which is an "Offset" added to the contents of the lower eight bits of the program counter when the counter is set at the next instruction. The range of the offset is −128 to +127 bytes from the next instruction.

INDEXED INDIRECT ADDRESSING—In indexed indirect addressing (referred to as [Indirect, X]), the second byte of the instruction is added to the contents of the X index register, discarding the carry. The result of this addition points to a memory location on page zero whose contents is the low order eight bits of the effective address. The next memory location in page zero contains the high order eight bits of the effective address. Both memory locations specifying the high and low order bytes of the effective address must be in page zero.

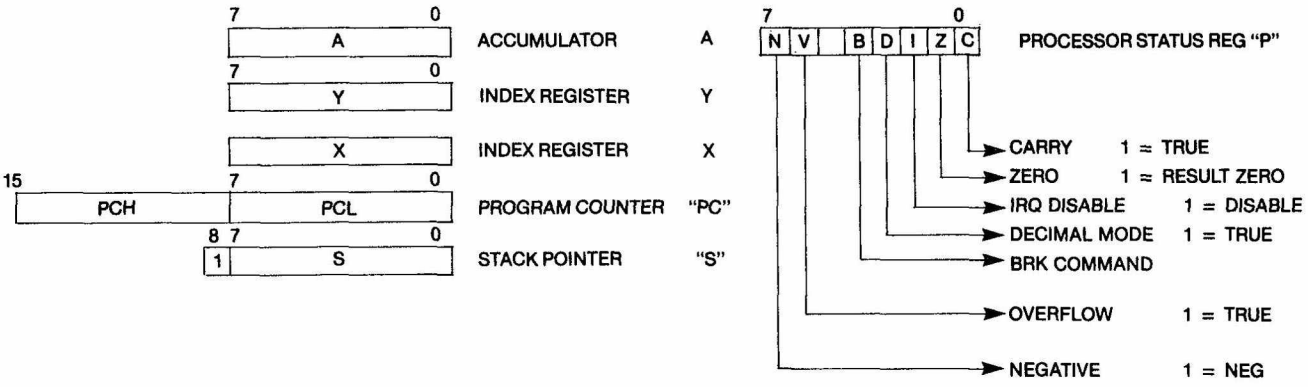
INDIRECT INDEXED ADDRESSING—In indirect indexed addressing (referred to as [Indirect, Y]). the second byte of the instruction points to a memory location in page zero. The contents of this memory location is added to the contents of the Y index register, the result being the low order eight bits of the effective address. The carry from this addition is added to the contents of the next page zero memory location, the result being the high order eight bits of the effective address.

ABSOLUTE INDIRECT—The second byte of the instruction contains the low order eight bits of a memory location. The high order eight bits of that memory location is contained in the third byte of the instruction. The contents of the fully specified memory location is the low order byte of the effective address. The next memory location contains the high order byte of the effective address which is loaded into the sixteen bits of the program counter.

INSTRUCTION SET—ALPHABETIC SEQUENCE

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive-or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift One Bit Right (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Register
TYA	Transfer Index Y to Accumulator

PROGRAMMING MODEL



INSTRUCTION SET — OP CODES, Execution Time, Memory Requirements

INSTRUCTIONS		IMMEDIATE			ABSOLUTE			ZERO PAGE			ACCUM.			IMPLIED			(IND.,X)			(IND.),Y			ZERO PAGE,X			ABS.,X			ABS.,Y			RELATIVE			INDIRECT			ZERO PAGE,Y			CONDITION CODES					
MNEMONIC	OPERATION	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	OP	N	#	N	Z	C	I	D	V
A D C	A+M+C→A (1)	69	2	2	6D	4	3	65	3	2							61	6	2	71	5	2	75	4	2	7D	4	3	79	4	3										✓	✓	✓	-	-	✓
A N D	A∧M→A (1)	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2	35	4	2	3D	4	3	39	4	3										✓	✓	-	-	-	-
A S L	C ← 7 0 ← 0				0E	6	3	06	5	2	0A	2	1										16	6	2	1E	7	3													✓	✓	✓	-	-	-
B C C	BRANCH ON C=0 (2)																																	90	2	2							-	-	-	-	-	-
B C S	BRANCH ON C=1 (2)																																	B0	2	2							-	-	-	-	-	-
B E Q	BRANCH ON Z=1 (2)																																	F0	2	2							-	-	-	-	-	-
B I T	A∧M				2C	4	3	24	3	2																																M_7	✓	-	-	-	M_6
B M I	BRANCH ON N=1 (2)																																	30	2	2							-	-	-	-	-	-
B N E	BRANCH ON Z=0 (2)																																	D0	2	2							-	-	-	-	-	-
B P L	BRANCH ON N=0 (2)																																	10	2	2							-	-	-	-	-	-
B R K	(See Fig. 1)													00	7	1																										-	-	-	1	-	-
B V C	BRANCH ON V=0 (2)																																	50	2	2							-	-	-	-	-	-
B V S	BRANCH ON V=1 (2)																																	70	2	2							-	-	-	-	-	-
C L C	0→C													18	2	1																										-	-	0	-	-	-
C L D	0→D													D8	2	1																										-	-	-	-	0	-
C L I	0→I													58	2	1																										-	-	-	0	-	-
C L V	0→V													B8	2	1																										-	-	-	-	-	0
C M P	A−M (1)	C9	2	2	CD	4	3	C5	3	2							C1	6	2	D1	5	2	D5	4	2	DD	4	3	D9	4	3										✓	✓	✓	-	-	-
C P X	X−M	E0	2	2	EC	4	3	E4	3	2																																✓	✓	✓	-	-	-
C P Y	Y−M	C0	2	2	CC	4	3	C4	3	2																																✓	✓	✓	-	-	-
D E C	M−1→M				CE	6	3	C6	5	2													D6	6	2	DE	7	3													✓	✓	-	-	-	-
D E X	X−1→X													CA	2	1																										✓	✓	-	-	-	-
D E Y	Y−1→Y													88	2	1																										✓	✓	-	-	-	-
E O R	A⊻M→A (1)	49	2	2	4D	4	3	45	3	2							41	6	2	51	5	2	55	4	2	5D	4	3	59	4	3										✓	✓	-	-	-	-
I N C	M+1→M				EE	6	3	E6	5	2													F6	6	2	FE	7	3													✓	✓	-	-	-	-
I N X	X+1→X													E8	2	1																										✓	✓	-	-	-	-
I N Y	Y+1→Y													C8	2	1																										✓	✓	-	-	-	-
J M P	JUMP TO NEW LOC.				4C	3	3																														6C	5	3				-	-	-	-	-	-
J S R	(See Fig 2) JUMP SUB				20	6	3																																				-	-	-	-	-	-
L D A	M→A (1)	A9	2	2	AD	4	3	A5	3	2							A1	6	2	B1	5	2	B5	4	2	BD	4	3	B9	4	3										✓	✓	-	-	-	-
L D X	M→X (1)	A2	2	2	AE	4	3	A6	3	2																			BE	4	3							B6	4	2	✓	✓	-	-	-	-
L D Y	M→Y (1)	A0	2	2	AC	4	3	A4	3	2													B4	4	2	BC	4	3													✓	✓	-	-	-	-
L S R	0→ 7 0 →C				4E	6	3	46	5	2	4A	2	1										56	6	2	5E	7	3													0	✓	✓	-	-	-
N O P	NO OPERATION													EA	2	1																										-	-	-	-	-	-
O R A	A∨M→A	09	2	2	0D	4	3	05	3	2							01	6	2	11	5	2	15	4	2	1D	4	3	19	4	3										✓	✓	-	-	-	-
P H A	A→M_S S−1→S													48	3	1																										-	-	-	-	-	-
P H P	P→M_S S−1→S													08	3	1																										-	-	-	-	-	-
P L A	S+1→S M_S→A													68	4	1																										✓	✓	-	-	-	-
P L P	S+1→S M_S→P													28	4	1																										(RESTORED)					
R O L	7 0 C				2E	6	3	26	5	2	2A	2	1										36	6	2	3E	7	3													✓	✓	✓	-	-	-
R O R	C → 7 0				6E	6	3	66	5	2	6A	2	1										76	6	2	7E	7	3													✓	✓	✓	-	-	-
R T I	(See Fig 1) RTRN INT													40	6	1																										(RESTORED)					
R T S	(See Fig 2) RTRN SUB													60	6	1																										-	-	-	-	-	-
S B C	A−M−C→A (1)	E9	2	2	ED	4	3	E5	3	2							E1	6	2	F1	5	2	F5	4	2	FD	4	3	F9	4	3										✓	✓	(3)	-	-	✓
S E C	1→C													38	2	1																										-	-	1	-	-	-
S E D	1→D													F8	2	1																										-	-	-	-	1	-
S E I	1→I													78	2	1																										-	-	-	1	-	-
S T A	A→M				8D	4	3	85	3	2							81	6	2	91	6	2	95	4	2	9D	5	3	99	5	3										-	-	-	-	-	-
S T X	X→M				8E	4	3	86	3	2																																96	4	2	-	-	-	-	-	-
S T Y	Y→M				8C	4	3	84	3	2													94	4	2																		-	-	-	-	-	-
T A X	A→X													AA	2	1																										✓	✓	-	-	-	-
T A Y	A→Y													A8	2	1																										✓	✓	-	-	-	-
T S X	S→X													BA	2	1																										✓	✓	-	-	-	-
T X A	X→A													8A	2	1																										✓	✓	-	-	-	-
T X S	X→S													9A	2	1																										-	-	-	-	-	-
T Y A	Y→A													98	2	1																										✓	✓	-	-	-	-

(1) ADD 1 TO "N" IF PAGE BOUNDARY IS CROSSED

(2) ADD 1 TO "N" IF BRANCH OCCURS TO SAME PAGE.
ADD 2 TO "N" IF BRANCH OCCURS TO DIFFERENT PAGE.

(3) CARRY NOT=BORROW

(4) IF IN DECIMAL MODE Z FLAG IS INVALID.
ACCUMULATOR MUST BE CHECKED FOR ZERO RESULT.

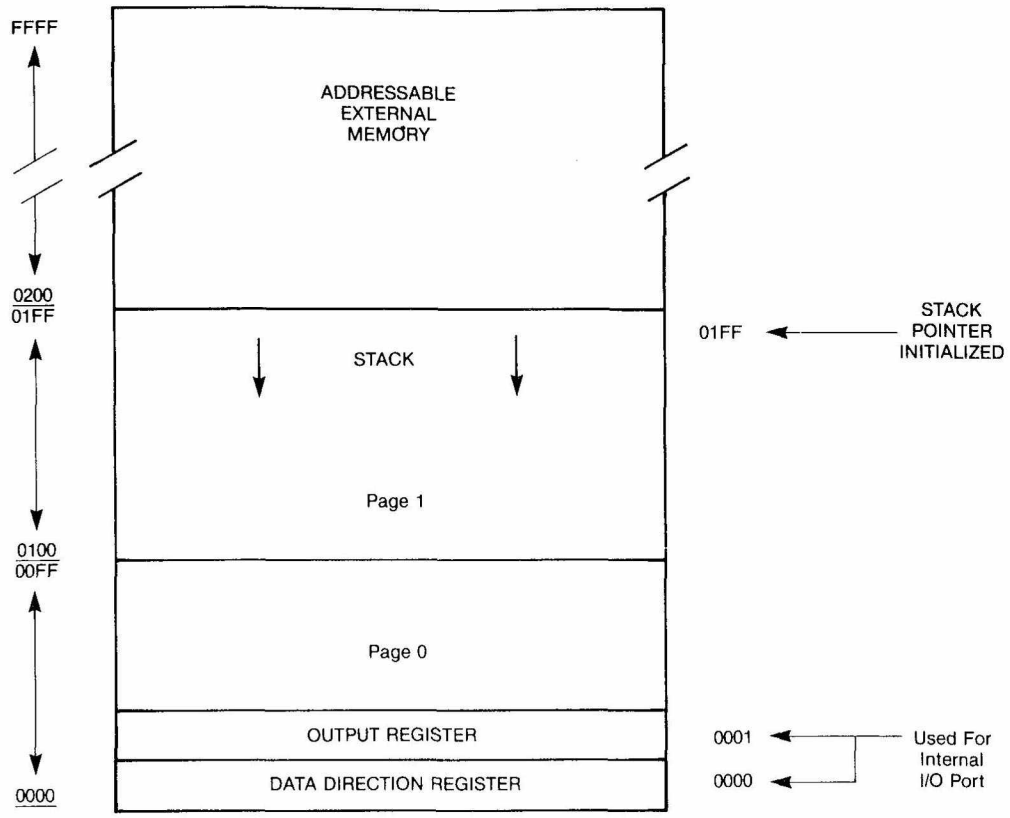
X INDEX, X
Y INDEX, Y
A ACCUMULATOR
M MEMORY PER EFFECTIVE ADDRESS
M_S MEMORY PER STACK POINTER

+ ADD
− SUBTRACT
∧ AND
∨ OR

⊻ EXCLUSIVE OR
✓ MODIFIED
- NOT MODIFIED
M_7 MEMORY BIT 7
M_6 MEMORY BIT 6

N NO. CYCLES
NO. BYTES

Note: Commodore Semiconductor Group cannot assume liability for the use of undefined OP Codes



6510 MEMORY MAP

APPLICATIONS NOTES

Locating the Output Register at the internal I/O Port in Page Zero enhances the powerful Zero Page Addressing instructions of the 6510.

By assigning the I/O Pins as inputs (using the Data Direction Register) the user has the ability to change the contents of address 0001 (the Output Register) using peripheral devices. The ability to change these contents using peripheral inputs, together with Zero Page Indirect Addressing instructions, allows novel and versatile programming techniques not possible earlier.

	Implicito			Accum.			Immediato			Pag. zero			Assoluto			Zero,X			Zero,Y			Assoluto, X			Assoluto,Y			(Zero),Y			(Zero,X)			Relativo			Indiretto			Flags							
Mnemonico	OP	#	b	OP	#	b	OP	#	b	OP	#	b	OP	#	b	OP	#	b	OP	#	b	OP	#	b	OP	#	b	OP	#	b	OP	#	b	OP	#	b	OP	#	b	N	V	-	B	D	I	Z	C
ADC							**69**	2	2	**65**	3	2	**6D**	4	3	**75**	4	2				**7D**	4*	3	**79**	4*	3	**71**	5*	2	**61**	6	2							N	V					Z	C
AND							**29**	2	2	**25**	3	2	**2D**	4	3	**35**	4	2				**3D**	4*	3	**39**	4*	3	**31**	5*	2	**21**	6	2							N						Z	
ASL				**0A**	2	1				**6**	5	2	**0E**	6	3	**16**	6	2				**1E**	7	3															N						Z	C
BCC																																	**90**	2**	2										
BCS																																	**B0**	2**	2										
BEQ																																	**F0**	2**	2										
BIT										**24**	3	2	**2C**	4	3																								7	6					Z	
BMI																																	**30**	2**	2										
BNE																																	**D0**	2**	2										
BPL																																	**10**	2**	2										
BRK	**0**	7	1																																						1		1			
BVC																																	**50**	2**	2										
BVS																																	**70**	2**	2										0
CLC	**18**	2	1																																							0			
CLD	**D8**	2	1																																									
CLI	**58**	2	1																																							0			
CLV	**B8**	2	1																																					0					
CMP							**C9**	2	2	**C5**	3	2	**CD**	4	3	**D5**	4	2				**DD**	4*	3	**D9**	4*	3	**D1**	5*	2	**C1**	6	2							N						Z	C
CPX							**E0**	2	2	**E4**	3	2	**EC**	4	3																									N						Z	C
CPY							**C0**	2	2	**C4**	3	2	**CC**	4	3																									N						Z	C
DEC										**C6**	5	2	**CE**	6	3	**D6**	6	2				**DE**	7	3															N						Z	
DEX	**CA**	2	1																																									
DEY	**88**	2	1																																										
EOR							**49**	2	2	**45**	3	2	**4D**	4	3	**55**	4	2				**5D**	4*	3	**59**	4*	3	**51**	5*	2	**41**	6	2							N						Z	
INC										**E6**	5	2	**EE**	6	3	**F6**	6	2				**FE**	7	3															N						Z	
INX	**E8**	2	1																																										
INY	**C8**	2	1																																										
JMP													**4C**	3	3																									**6C**	5	3					
JSR													**20**	6	3																															
LDA							**A9**	2	2	**A5**	3	2	**AD**	4	3	**B5**	4	2				**BD**	4*	3	**B9**	4*	3	**B1**	5*	2	**A1**	6	2							N						Z	
LDX							**A2**	2	2	**A6**	3	2	**AE**	4	3				**B6**	4	2				**BE**	4*	3													N						Z	
LDY							**A0**	2	2	**A4**	3	2	**AC**	4	3	**B4**	4	2				**BC**	4*	3																	N						Z	
LSR				**4A**	2	1				**46**	5	2	**4E**	6	3	**56**	6	2				**5E**	7	3																	0						Z	C
NOP	**EA**	2	1																																										
ORA							**9**	2	2	**5**	3	2	**D**	4	3	**15**	4	2				**1D**	4*	3	**19**	4*	3	**11**	5*	2	**1**	6	2							N						Z	
PHA	**48**	3	1																																										
PHP	**8**	3	1																																										
PLA	**68**	4	1																																									Z	
PLP	**28**	4	1																																							*	*		*	*	*	*	*
ROL				**2A**	2	1				**26**	5	2	**2E**	6	3	**36**	6	2				**3E**	7	3															N						Z	C
ROR				**6A**	2	1				**66**	5	2	**6E**	6	3	**76**	6	2				**7E**	7	3															N						Z	C
RTI	**40**	6	1																																							*	*		*	*	*	*	*
RTS	**60**	6	1																																										
SBC							**E9**	2	2	**E5**	3	2	**ED**	4	3	**F5**	4	2				**FD**	4*	3	**F9**	4*	3	**F1**	5*	2	**E1**	6	2							N	V					Z	C
SEC	**38**	2	1																																										1
SED	**F8**	2	1																																								1			
SEI	**78**	2	1																																								1		
STA										**85**	3	2	**8D**	4	3	**95**	4	2				**9D**	5	3	**99**	5	3	**91**	6	2	**81**	6	2														
STX										**86**	3	2	**8E**	4	3				**96**	4	2																										
STY										**84**	3	2	**8C**	4	3	**94**	4	2																													
TAX	**AA**	2	1																																										
TAY	**A8**	2	1																																										
TSX	**BA**	2	1																																										
TXA	**8A**	2	1																																										
TXS	**9A**	2	1																																										
TYA	**98**	2	1																																											

* Aggiungere 1 ciclo se l'indirizzo oltrepassa il limite di pagina del Program Counter attuale.
** Aggiungere 1 ciclo se il salto viene eseguito. Aggiungere 2 cicli se il salto viene eseguito verso una locazione in una pagina diversa.