copyright (c) 2005 stefano frangioni.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just this first page, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU Free Documentation License".

questo materiale è fornito "così com'è", senza alcuna garanzia in merito alla sua correttezza; ogni affermazione contraria, sia implicita che esplicita, è da considerarsi nulla.

questo materiale è disponibile anche su http://www.perpassione.org/downloads/calcelettronici.zip senza necessità di password o altro, così come prescritto dalla FDL.

appunti di

CALCOLATORI ELETTRONICI I

parte II

basati sui miei appunti di calcolatori elettronici.
Contributi:
-il libro del bucci;
-gli appunti di alessio bazzica;
-droscy (teoria dell'informazione);
-chiunque altro vuole contribuire!!

convenzioni:
//?testo?// indica del testo non chiaro, non preciso, probabilmente sbagliato.
/*testo*/ indica un commento

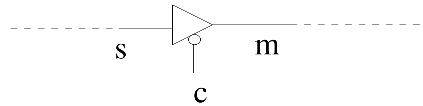
nota: il “sorgente” è scritto con openoffice.org 1.1.3 su sistema linux, usando prevalentemente i caratteri “Nimbus Roman No9 L”, “Bitstream Vera Sans Mono”, “Nimbus Mono L”. Prima di modificare il documento conviene installarsi questi font, altrimenti il testo potrebbe risultare un po' scombinato. Questi caratteri sono scaricabili da http://www.perpassione.org/downloads/fonts.zip.

Indice

-NOTA: buffer tristate.....5
-LA CPU (Central Processing Unit).....5
  La sequenza di FETCH.....5
-Struttura a singolo BUS interno.....6
  Ancora sul FETCH.....8
  Istruzioni e indirizzamento.....9
    salto incondizionato.....9
    salto condizionato.....10
    modi di indirizzamento.....10
    calcolo del numero di cicli di bus necessario per eseguire un'istruzione.....10
-Modelli CISC e RISC.....11
-Prestazioni di un calcolatore.....12
    PIPELINE.....12
-Aree di memoria.....13
  lo stack.....13
    chiamata a sottoprogrammi.....13
-Le memorie.....16
    diagramma a caramella.....18
    esercizio “chip e banco” (1).....19
    esercizio “chip e banco” (2).....20
  Gerarchie di memoria.....21
-Architettura 8086.....22
  Modello di programmazione 8086.....22
    Bus dati.....22
    Bus indirizzi.....22
    Registri.....22
  ancora sulla segmentazione.....23
  Little-endian, big-endian.....24
  Accesso alla memoria ad indirizzi pari e ad indirizzi dispari.....25
-Assembler 8086.....26
  Per definire un segmento.....26
  All'interno del segmento dati.....27
    definizione di variabili.....27
    definizione di costanti.....27
  All'interno del segmento di stack.....27
  All'interno del segmento di codice.....28
    Le etichette.....28
    istruzioni per far tornare il controllo al sistema operativo.....28
  la direttiva END.....29
  macro e procedure.....29

macro.....29
procedure.....30
modi di indirizzamento.....31
PSP (program segment prefix).....32
alcune istruzioni, lunghezza/codifica delle istruzioni.....32
alcune istruzioni.....32
codifica delle istruzioni.....33
parole chiave offset e seg.....35
assemblare e debuggare.....35
-Input / Output.....36
Istruzioni assembler tipiche.....37
Esempio di interfaccia di sola uscita a controllo di programma.....38
gestione sotto controllo di interruzione (interrupt esterni).....39
stabilire l'identità del dispositivo che ha richiesto l'interruzione.....39
Schema generale di una interruzione.....40
Gestione delle priorità (interruzioni vettorizzate).....40
nota: classificazione delle interruzioni.....41
DMA (direct memory access).....42
-GNU Free Documentation License.....43

NOTA: buffer tristate.
è un dispositvo che, sulla base di un segnale di controllo, permette di porre un segnale in terzo stato oppure lasciar passare il segnale che gli giunge. Ad esempio:



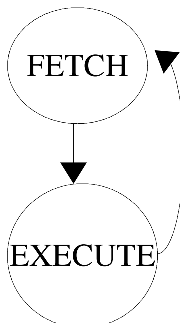
dove c è il segnale di controllo che, se asserito, pone m in terzo stato. Quando m è in terzo stato è come se non fosse collegato a niente. Se invece c è posto a zero, allora m ha lo stesso valore di s.

LA CPU (Central Processing Unit)

La CPU è una "macchina" di scopo generale: sa fare un certo numero di operazioni e, tramite queste, è in grado di risolvere pressoché ogni problema. I passi che la CPU deve compiere per risolvere un certo specifico problema si chiamano istruzioni. Le istruzioni che la CPU deve eseguire per risolvere un problema, in quanto macchina a scopo generale, devo essere assunte dall'esterno, in particolare dalla memoria.
La sequenza di FETCH è l'insieme ordinato delle operazioni necessarie a prelevare un'istruzione dalla memoria. Al termine del fetch di un'istruzione tale istruzione si trova immagazzinata in un apposito registro interno alla CPU, chiamato IR (Instruction Register). Una istruzione può essere schematizzata ad esempio in questo modo:

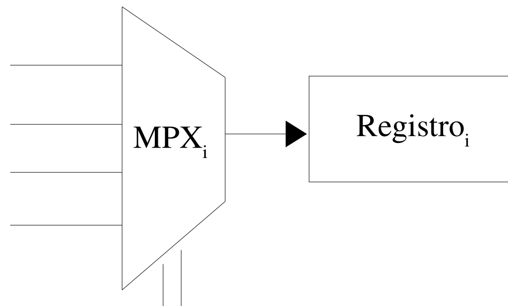
opcode	operando1	operando2	operando3

dove opcode è il codice univoco che identifica una certa operazione. Opcode indica il tipo di operazione da eseguire, il modo di indirizzamento, la lunghezza complessiva dell'istruzione da eseguire. Nello spazio occupato da opcode sono indicati anche gli eventuali registri coinvoli nell'istruzione. Un'istruzione si può indicare anche in questo modo:
ADD 131, 28, R17
che, a logica, fa la somma di 131 e 28 e la salva nel registro R17. In questa istruzione è stata adottata la convenzione che la destinazione è ciò che è indicato più a destra. Nell'8086 invece la convenzione è di usare come destinazione ciò che è scritto più a sinistra. Questo è solo un esempio di istruzione, esempio che per altro non dovrebbe funzionare nell'8086 neppure mettendo R17 a sinistra.
In generale tutte le istruzioni indicate in questa area degli appunti sono solo degli esempi GENERICI che non è detto che poi funzionino realmente nell'8086.
Una volta eseguito il fetch dell'istruzione si passa alla sua esecuzione e poi al fetch della prossima istruzione, e così via con questo schema:

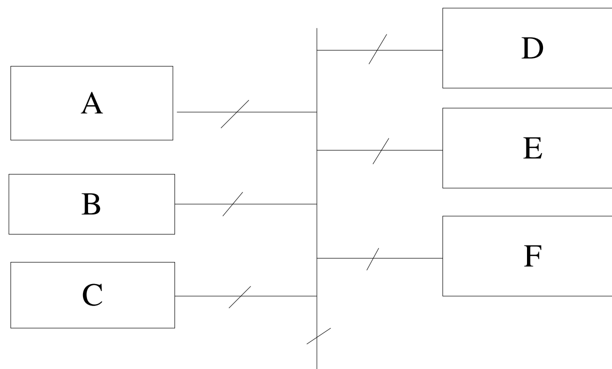


Struttura a singolo BUS interno

I vari componenti della CPU (ALU, registri, CU, ...) devono, per poter operare, comunicare tra loro. Un modo per realizzare tale comunicazione potrebbe essere questo:

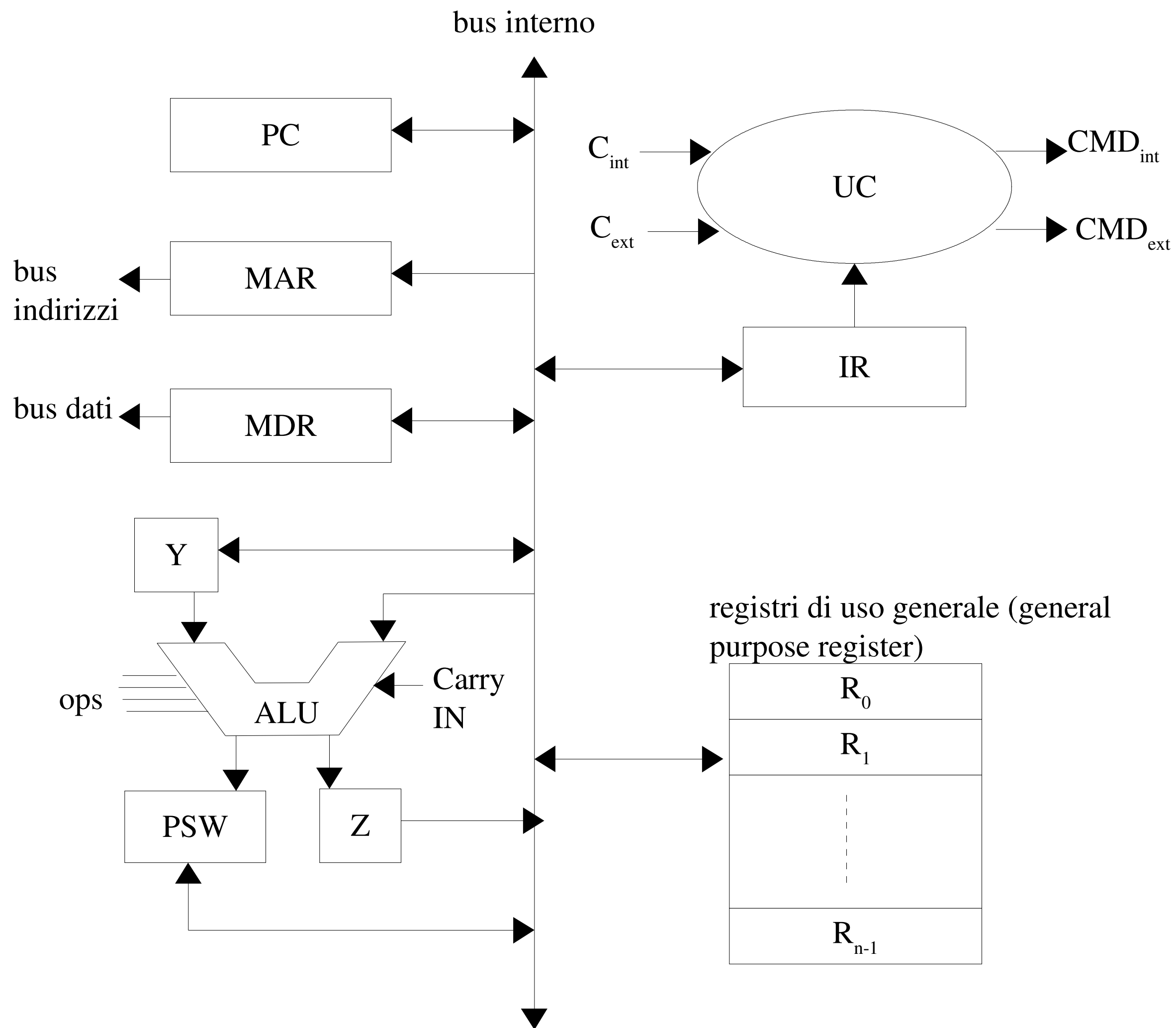


Nel quale le linee di ingresso del multiplexer arrivano dalle uscite dei registri e le linee di selezione arrivano dalla CU. Questo modo però è estremamente complesso, perché se ad esempio abbiamo m registri abbiamo bisogno di $\binom{m}{2}$ linee per farli comunicare tutti. In compenso però si ha la possibilità di fare ad esempio due trasferimenti contemporanei da due registri a due destinazioni arbitrarie.
Un modo alternativo, meno costoso dal punto di vista dei componenti, è questo:



Questa struttura è chiamata struttura a singolo bus interno: c'è una sola linea, il BUS, che collega tutti gli elementi della CPU. Salta subito agli occhi il difetto principale di questa strutturazione: non è possibile trasferire informazioni contemporaneamente da due diversi registri, anche se le destinazioni sono diverse, perché il canale è unico (mentre è però possibile trasferire dati da un solo registro a più registri contemporaneamente). Questa struttura è possibile perché le uscite/entrate di tutti i registri che non devono trasmettere/ricevere una determinata informazione vengono messe in terzo stato ($\bar{R}_{out}/\bar{R}_{in}$). Quindi ad esempio quando si vuole fare ad esempio un trasferimento da B a E, è necessario abilitare l'uscita di B, in modo che il contenuto di B venga posto sul bus, e l'ingresso di E, in modo che il contenuto del bus passi in E. Tutti gli altri ingressi/uscite devono essere disabilitati.
Una struttura di questo tipo obbliga di fatto a fare tutte operazioni in sequenza, anche quelle che teoriacamente potevano essere fatte in modo parallelo. Schematicamente si può indicare così:
B_{out}, E_{in}

Una scrittura di questo genere sta ad indicare che tutte le operazioni presenti sulla stessa riga devono essere fatte contemporaneamente, mentre operazioni in righe diverse devono essere fatte in sequenza. Vediamo ora in dettaglio la struttura a singolo bus interno di una CPU teorica:



In dettaglio i componenti:

-ALU: serve per fare somme, sottrazioni ed operazioni logiche. ops è un insieme di fili di controllo che serve per poter scegliere l'operazione da fare (solo uno di essi risulta asserito);

-Y contiene uno dei due operandi necessari per poter svolgere un'operazione con la ALU. Tale registro è richiesto per il fatto che i collegamenti si basano tutti su un singolo bus. L'altro operando viene prelevato dal bus. Ad esempio, per fare la somma R1+R2 e memorizzarla in R27 la CU fa così:

-pone R1 in Y; ($R1_{out}$, Y_{in})

-pone R2 sul bus, asserisce il controllo necessario a fare la somma; ($R2_{out}$, ADD)

-preleva il contenuto di Z e lo mette in R27; (Z_{out}, $R27_{in}$)

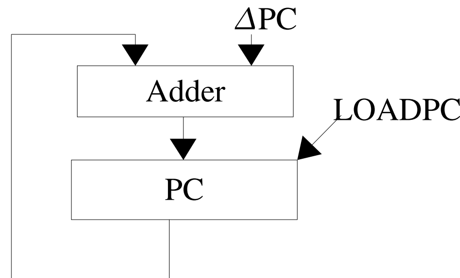
Questa schematizzazione corrisponde a tre istanti temporali successivi, che nel possibile diagramma di stato della CU corrispondono a 3 palle distinte.

-PSW (Processor Status Word, parola di stato del processore): è uno speciale registro in cui vengono memorizzate informazioni relative all'ultima operazione eseguita dall'ALU. Alcuni esempi possono essere il bit di overflow, il bit di carry, il bit di segno, lo zero bit. Tutte queste informazioni vengono lette dalla CU quando occorre (sono delle condizioni per la CU). In alcuni casi ci possono essere dei canali privilegiati di comunicazione fra CU e PSW.
-UC (Unità di Controllo, oppure CU, Control Unit). I due ingressi che presenta sono le condizioni sulle quali si basa la sua attività: condizioni interne (provengono dalla CPU, a volte anche dalla stessa CU) e condizioni esterne (ad esempio dalla memoria o da altre periferiche. un esempio di tali condizioni potrebbe essere un MFC, Memory Function Completed, “ho finito” che proviene dalla memoria, più lenta della CPU, quando ha finito di eseguire l'operazione richiesta). Le linee di uscita invece (CMD) trasportano comandi per la CPU (CMD_{int}) e per l'esterno (CPU_{ext}). Le linee C_{ext} e CMD_{ext} viaggiano sul bus controlli.
-MAR (Memory Address Register) è un registro che contiene indirizzi di memoria. È connesso con l'esterno tramite il bus indirizzi.
-MDR (Memory Data Register) è un registro che contiene dati e che comunica con l'esterno con il bus dati.
-IR (Instruction Register). È un registro che contiene l'istruzione che è correntemente in esecuzione. È qui che viene memorizzata l'istruzione non appena è stata compiuta la fase di FETCH. Ad ogni istruzione corrisponde un numero univoco, la codifica dell'istruzione in macchina, che la rappresenta. Ad esempio se IR è grande 8 bit, posso fare 2^8=256 diverse istruzioni.
-PC (Program Counter): è un registro il cui contenuto è l'indirizzo dell'istruzione successiva all'istruzione corrente (ma non è detto che sia la prossima istruzione che sarà eseguita, vedere i salti).
Esempio di una ipotetica istruzione di un programma:
LOAD.W MEM[8172], R15
che potrebbe significare “carica nel registro R15 l'istruzione contenuta nella locazione di memoria di indirizzo 8172”. A tale istruzione corrispondono più microistruzioni, ad ognuna delle quali corrisponde un diverso stato della CU. Questa istruzione sarebbe realizzata attraverso questa microistruzione:
$IR.Address_{out}$, MAR_{in}, READ, WMFC (Wait for Memory Function Completed).

Ancora sul FETCH

Ecco un esempio di come potrebbe essere realizzata la sequenza di fetch in una CPU molto semplificata che preveda istruzioni tutte lunghe 1 byte ed un bus dati di almeno 8 bit:
-PC_{out}, MAR_{in}, READ, WMFC, CLEAR Y, SetCarryIn, ADD, Z_{in} /*mette l'indirizzo dell'istruzione da prelevare sul bus indirizzi, ci somma 1 (le istruzioni sono lunghe un byte) e salva in Z. È in attesa che la memoria esegua restituisca l'istruzione richiesta*/
-Z_{out}, PC_{in} /*salva il valore appena calcolato nel PC*/
-MDR_{out}, IR_{in} /*l'istruzione letta viene posta inIR*/
in realtà questa ipotesi è molto semplificativa e dispendiosa, per il fatto che ipotizza istruzioni tutte lunghe un byte e utilizza la ALU per incrementare il PC. Una cosa più realistica sarebbe ipotizzare di avere un elemento di somma utile solo all'incremento del PC, per evitare perdite di tempo, e ipotizzare che le istruzioni non siano tutte lunghe uguali. La CU, alla lettura della prima parte di istruzione, sa

quante altre sequenze di fetch sono necessarie per caricare l'intera istruzione nell'IR.
Un esempio schematico potrebbe essere questo:



Istruzioni e indirizzamento

Un'istruzione viene descritta attraverso l'operazione da compiere, il tipo di dato, il numero di operandi e il modo del loro reperimento (modo di indirizzamento). Le istruzioni disponibili possono essere raggruppate così:
-istruzioni di trasferimento dati;
-istruzioni per l'elaborazione logica/aritmetica dei dati;
-controllo del flusso dell'esecuzione del programma (salti).
Per identificare l'indirizzo di un'istruzione (NB: le istruzioni si trovano in memoria!!) si fa uso delle etichette. Le etichette sono delle stringhe seguite dai due punti che si indicano in genere a sinistra o sopra alla riga di codice alla quale si riferiscono. Fra le etichette e gli indirizzi delle istruzioni per le quali le etichette sono definite c'è una corrispondenza biunivoca.

<u>salto incondizionato</u>
Un'istruzione di salto incondizionato quando eseguita va a sostituire il contenuto di PC con l'indirizzo di memoria dell'istruzione a cui saltare. Una tale istruzione potrebbe chiamarsi ad esempio JUMP ed avere la seguente struttura:

JUMP	JUMP.ADDRESS

i cui valori nell'IR potrebbe essere inseriti così:

OPCODE	IR.ADDRESS

quindi l'istruzione farebbe IR.ADDRESS$_{out}$, PC$_{in}$.
Invece di codificare l'intero indirizzo si potrebbe invece decidere di codificare la differenza (offset, scostamento) positiva o negativa di indirizzo fra l'istruzione corrente e quella alla quale si deve saltare. In questo caso allora avrei

OPCODE	IR.OFFSET

e per incrementare il PC dovrei fare:
IR.OFFSET$_{out}$, Y$_{in}$
PC$_{out}$, ADD, Z$_{in}$
Z$_{out}$, PC$_{in}$
L'istruzione potrebbe essere ad esempio JMP etichetta_di_salto.

<u>salto condizionato</u>
Il salto condizionato è un'istruzione che server per saltare ad un'altra istruzione solo se è verificata una certa condizione.
Ad esempio:

```
LOOP: /*questa è un'etichetta*/
        ...
        ...
        ...
        DEC R4 /*decremento R4 e memorizzo il risultato in R4*/
        JNZ LOOP /*JNZ=Jump if Not Zero*/
```

JNZ va a controllare lo zero bit della PSW (che è a uno solo quando la precedente operazione ha dato risultato zero) e se è zero passa il controllo all'istruzione successiva all'etichetta LOOP.

<u>modi di indirizzamento</u>
I modi di indirizzamento indicano come posso esprimere gli operandi di una certa istruzione. Sono uno degli aspetti più importanti di una architettura. Eccone qualche esempio:
-immediato: il valore viene indicato nel codice da eseguire, quindi è memorizzato assieme all'istruzione, viene tirato giù dalla memoria con essa. esempio: 1249
-di registro: viene indicato il nome del registro che contiene il valore che ci interessa. esempio: R5
-indiretto di registro: viene indicato il registro che contiene l'indirizzo della locazione di memoria della quale ci interessa il valore. esempio: [R5]. con questa scrittura sto indicando il contenuto della cella di memoria il cui indirizzo è contenuto in R5
-diretto: viene indicato il nome di una variabile, che rappresenta un indirizzo di memoria. esempio: var. con questa scrittura mi riferisco al contenuto della cella di memoria il cui indirizzo "è" var; nella codifica dell'istruzione viene memorizzato solo l'indirizzo di var! il valore lo deve andare a prendere nella memoria!
-relativo: viene indicato il nome di una variabile e il numero di posizioni di scostamento (offset) rispetto ad essa al quale andare a pescare il valore che serve. esempio: vect[R5], dove vect è la base, dove inizia il vettore ed R5 contiene l'offset dalla base
-relativo (matrice). esempio: matrix[R1][R2]
tutti questi modi di indirizzamento non sono presenti in tutte le macchine, alcuni di essi infatti (quelli più complicati) sono assenti dalle macchine di tipo RISC (vedi dopo).

<u>calcolo del numero di cicli di bus necessario per eseguire un'istruzione</u>
Per poter calcolare il numero di cicli di bus necessari per poter eseguire un'istruzione (cioè il numero di volte che devo accedere alla memoria) è necessario conoscere:
-la lunghezza dell'istruzione (cioè della codifica dell'istruzione e dei suoi operandi);
-la dimensione del bus dati;
-la dimensione del bus indirizzi;
Con tutte queste informazioni è possibile calcolare il numero di cicli di bus necessari per eseguire una certa istruzione. Ad esempio, supponiamo di avere un bus indirizzi a 2 byte ed un bus dati a 1 byte vogliamo eseguire l'istruzione

```
ADD.DW R5, VAR   /*cioè VAR=R5+VAR    (DW=double word)*/
```

Supponiamo inoltre che il codice dell'istruzione occupi 2 byte. Allora saranno necessarie:
-2 letture per leggere dalla memoria il codice dell'operazione e il nome del registro (perché il bus dati è da 1 byte);
-2 letture per leggere dalla memoria l'indirizzo VAR (perché VAR è a 2 byte, come il bus indirizzi, e il bus dati è a 1 byte); /*qui è conclusa l'operazione di fetch (vedi il paragrafo delle prestazioni)*/
-2 letture per leggere dalla memoria il valore contenuto nell'indirizzo VAR. Servono due letture perché la word è supposta di un byte e VAR è su due word (DW) /*questo rigo è la fase di decode, vedi il paragrafo delle prestazioni*/;
-altre 2 operazioni con il bus, questa volta di scrittura, per porre il risultato della somma all'indirizzo VAR.
In tutto quindi sono 6 cicli di bus.

Un altro esempio: si ha un bus indirizzi a 3 byte ed un bus dati a 2 byte. 1 word è 2 byte. Si vuole eseguire
ADD.DW R2, VAR /*quindi VAR occupa 4 byte, perché DW (double word)*/
Supponendo che per la codifica dell'operazione siano impiegati 2 byte, la dimensione totale dell'istruzione è di 2+3=5byte, per cui saranno necessarie:
-3 operazioni di lettura per leggere tutta l'istruzione (il bus dati infatti è a 2 byte e dobbiamo leggerne 5);
-2 operazioni di lettura per fare il decode di VAR, cioè per leggere il dato di indirizzo VAR (infatti il bus dati è a 2 byte e ne dobbiamo leggere 4);
-altre 2 operazioni, questa volta di scrittura, per memorizzare all'indirizzo VAR il risultato della somma effettuata.
In tutto sono quindi 7 cicli di bus.

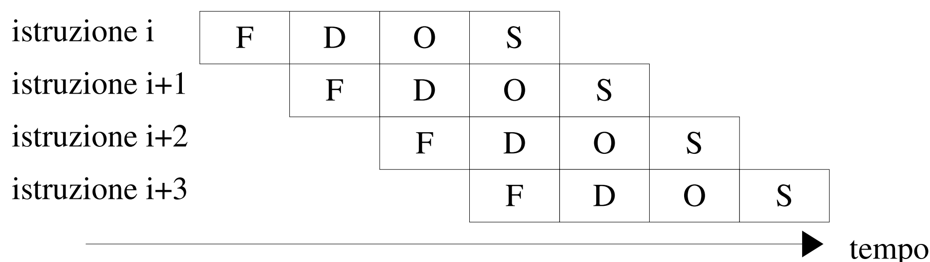
Modelli CISC e RISC

Ci sono principalmente due filosofie diverse e contrapposte nella realizzazione di un'architettura: la filosofia CISC (Complex Instruction Set Computer) e la RISC (Reduced Instruction Set Computer).
Le macchine CISC tendono ad avere un elevato set di istruzioni di macchina, con le quali si possono eseguire operazioni che spesso sono proprie di un linguaggio di alto livello (come il calcolo della radice quadrata). Nelle macchine CISC gli operandi delle istruzioni tendono ad avere una dimensione più elevata, e sono disponibili molti modi di indirizzamento dei dati (anche vettori, matrici, ...). Con un'architettura CISC i programmi sono molto più corti e semplici e il linguaggio macchina si avvicina ad un linguaggio di alto livello. Uno degli svantaggi di un'architettura CISC è l'inefficienza nell'esecuzione delle istruzioni. In genere le istruzioni non hanno una lunghezza costante.
Le macchine RISC tendono a minimizzare il numero di istruzioni di macchina, a vantaggio di una elevata efficienza nell'esecuzione delle (poche) istruzioni presenti, grazie ad una maggiore possibilità di parallelizzazione (vedi pipeline). Saranno poi i programmatori a realizzare ciò che vogliono tramite le essenziali istruzioni presenti. Questa scelta rende quindi i programmi più lunghi e complessi, facendo però calare drasticamente il tempo medio di esecuzione di ciascuna istruzione di macchina.
Princìpi di base: le istruzioni di macchina devono avere tutte la stessa lunghezza (1 word), facile decodifica; le istruzioni devono essere semplici; ridotte possibilità di indirizzamento (solo operazioni tra registri e poi operazioni di lettura/scrittura in memoria); elevato numero di registri di uso generale.

Prestazioni di un calcolatore

In genere sono misurate in tempo di esecuzione dei programmi. Il tempo di esecuzione dei programmi dipende anche dal set di istruzioni della macchina, oltre che dalla frequenza del clock. Sia T_{prog} il tempo di esecuzione di un programma, si ha $T_{prog}=CPP\cdot T_{clk}=\frac{CPP}{f_{clk}}$, dove CPP è il numero di colpi di clock necessari ad eseguire tutto il programma, T_{clk} è il periodo del clock e f_{clk} la frequenza. Inoltre si può scrivere $CPP=\sum_{i=1}^{N} CPI_i\cdot m_i=N\cdot\overline{CPI}$, dove CPI_i è il numero di colpi di clock necessari per eseguire l'i-esima istruzione, m_i è il numero di volte che tale istruzione viene ripetuta, N è il numero di istruzioni del programma e $\overline{CPI}$ è il numero medio di colpi di clock per ogni istruzione. Sostituendo si ha $T_{prog}=\frac{N\cdot\overline{CPI}}{f_{clock}}$. Nel CISC l'idea è quella di diminuire il più possibile N, il numero di istruzioni necessarie per eseguire un certo programma (implementando molte e dettagliate istruzioni svolte dall'hardware), a fronte però spesso di un $\overline{CPI}$ più elevato; nei processori RISC invece l'idea è quella di ridurre il più possibile $\overline{CPI}$.

Un modo per incrementare le prestazioni generali potrebbe essere quello di suddividere la sequenza di fetch/execute di una istruzione in più passi, così da poter parallelizzare passi diversi del fetch di istruzioni diverse. La sequenza elementare di istruzioni potrebbe ad esempio essere questa:
-fetch, F (prelevamento dell'istruzione dalla memoria);
-decode, D (eventuale prelevamento degli operandi dalla memoria);
-operate, O (esecuzione dell'istruzione);
-store, S (eventuale scrittura in memoria del risultato)
Se supponiamo di avere una macchina con un parallelismo interno tale da avere 4 unità nella CPU (F, D, O, S), ognuna delle quali in grado di compiere una delle quattro operazioni elencate sopra, in contemporanea con le altre, ogni istruzione che vogliamo eseguire deve passare da ognuna di queste quattro macchine. Si ha una cosa del genere, chiamata PIPELINE:



In questo modo a regime, se il tempo impiegato per ogni fase è uguale per tutte le fasi, si termina l'esecuzione di una istruzione in ogni intervallo di tempo.

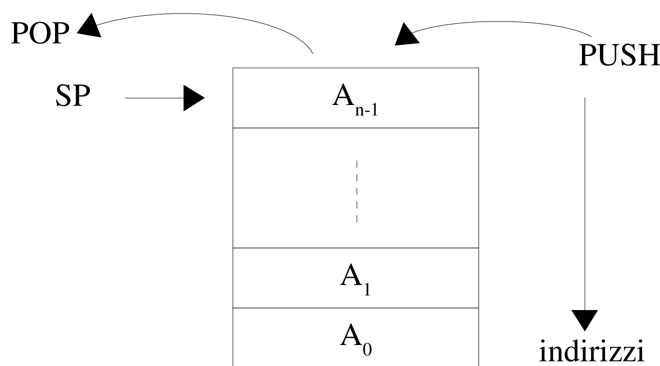
Ovviamente le prestazioni nell'esecuzione dipendono fortemente dall'architettura che si ha a disposizione. Ad esempio se invece di avere un singolo bus interno avessimo 3 bus, saremmo in grado di fare un'operazione di somma con una sola microistruzione, perché si avrebbero tre bus disponibili contemporaneamente in cui far transitare gli operandi ed il risultato.

Aree di memoria

La memoria centrale si può suddividere in tre aree (memoria segmentata):
-l'area stack (pila);
-l'area dati;
-l'area codice;
L'indirizzo di inizio di ciascun segmento di memoria è memorizzato in un apposito registro.

lo stack

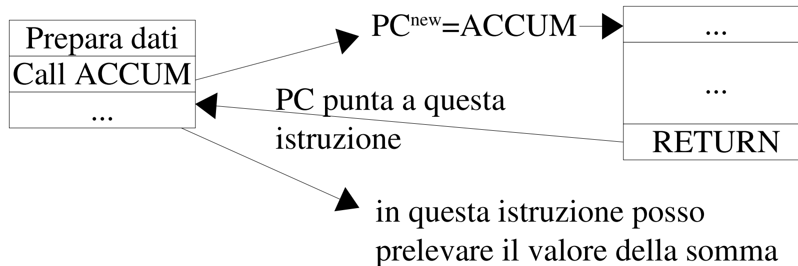
Lo stack è un'area di memoria che viene utilizzata in modalità LIFO (Last In First Out, esce per primo l'ultimo entrato), attraverso due operazioni, il PUSH ed il POP.
Con il PUSH si effettua il caricamento nello stack di un dato, con il POP invece si preleva l'ultimo dato inserito nello stack. Si può vedere rappresentato così, con gli indirizzi delle celle che aumentano dall'alto verso il basso ed il caricamento che viene effettuato dall'alto (l'n-esimo elemento inserito è A_{n-1}):



Nella CPU c'è un apposito registro, chiamato SP (Stack Pointer), che punta all'ultima cella di memoria scritta nello stack. Se si ipotizza che nello stack si possano trasferire solo dati di dimensione di 1 word, allora SP varia sempre di 1 word ad ogni operazione di PUSH o di POP. In dettaglio,
-con l'istruzione PUSH source si ha che SP viene decrementato della dimensione di 1 word ed in [SP] (cioè nella locazione di memoria puntata da SP) viene posto source;
-con l'istruzione POP dest si ha che in dest viene posto [SP] e poi SP viene incrementato della dimensione di una word. NB con il POP si fa una copia del valore puntato da SP in dest e poi si incrementa SP. La cella precedente diviene scrivibile, ma il suo valore non è stato eliminato.
PUSH e POP sono istruzioni di macchina, anche nell'8086.
Con Top of Stack si identifica la cima massima raggiungibile dallo stack, mentre con Bottom of Stack si identifica la base dello stack.
NB: lo stack è semplicemente SIMULATO nella memoria perché se vogliamo possiamo di fatto accedere ad una qualsiasi posizione, in maniera arbitraria come avviene per una qualsiasi locazione di memoria.
Un esempio di uso dello stack molto comune è nella chiamata a sottoprogrammi. Ad esempio: abbiamo un certo programma che per fare la somma di un vettore si serve di un sottoprogramma al quale passa come parametri l'indirizzo del vettore sul quale operare, il numero di elementi del vettore e l'indirizzo di memoria in cui scrivere il risultato. I parametri passati vengono memorizzati nello stack (lo stack è condiviso fra programma chiamante e chiamato), insieme all'indirizzo in cui si trova l'istruzione alla quale ripassare il controllo al termine del sottoprogramma. L'istruzione da utilizzare

per chiamare un sottoprogramma è la CALL, che prende come parametro l'etichetta del sottoprogramma (che verrà poi tradotta in un indirizzo di memoria).

La schematizzazione di programma chiamante e chiamato potrebbe essere questa:



Mentre l'istruzione JUMP era senza ritorno, l'istruzione CALL (chiamata a procedura), una volta eseguito il chiamato, ritorna al programma chiamante, perché salva il valore di ritorno di PC nello stack con un PUSH PC (e lo recupera con POP PC). Quindi in pratica “CALL sub” esegue “PUSH PC” e “JMP sub”, mentre l'istruzione “RETURN” esegue “POP PC” per ripassare il controllo al programma chiamante.
Notare inoltre che lo stack si presta molto bene anche per chiamate a procedure che al loro interno prevedano chiamate ad altre procedure, oppure per chiamate a procedure ricorsive (che chiamano se stesse).
Quando viene effettuata una chiamata a procedura lo stack viene popolato così:

temp
$PC^{chiamante}$
parametri

dove temp viene scritto dal programma chiamato e serve per salvare il contenuto dei registri (con delle operazioni di PUSH) dei quali la procedura fa uso per poi, al termine, poterli ripristinare con il loro valore originario con delle operazioni di POP.

Esempio: fare la somma dei numeri contenuti nel vettore num usando un sottoprogramma, considerando di avere in N il numero di elementi di num e di volere il risultato all'indirizzo #sum e supponendo che una word sia lunga W byte.

Programma principale (area codice):
/*SP sta puntando ad SP^{inizio} (vedi l'area stack)*/
PUSH num
PUSH n
PUSH #SUM
CALL ACCUM
SUB 3*W, SP /*riporta SP al valore SP^{inizio}*/

Accum (area codice): /*si suppone che la procedure Accum faccia uso dei registri R0, R1, R2, R3*/
PUSH R3 /*salvo nello stack i registri che andrò a toccare*/

materiale sotto licenza FDL – Free Documentation License - si veda la prima pagina e l'ultima sezione

PUSH R2
PUSH R1
PUSH R0 /*dopo questa istruzione SP vale SP^1*/
MOVE SP, R3
ADD 7*W, R3 /*dopo questa istruzione R3 vale A^2*/
MOVE [R3], R2 /* dopo questa istruzione R2 contiene num*/
SUB 1*W, R3 /* dopo questa istruzione R3 vale A^3*/
MOVE [R3], R1 /* dopo questa istruzione R1 contiene N*/
.... /*procedura che fa la somma, non riportata. supponiamo che salvi il risultato in R0*/
SUB 1*W, R3 /* dopo questa istruzione R3 vale A^4 e quindi punta alla cella che contiene l'indirizzo di memoria nel quale salvare il risultato*/
MOVE [R3], R2
MOVE R0, [R2] /*NB: al posto di questa istruzione e della precedente se fosse stato implementato un altro modo di indirizzamento, avremmo potuto scrivere ad esempio solo MOVE R0, [[R3]]*/
POP R0 /*ripristino nello stack, nel corretto ordine, il valore dei registri che mi ero salvato*/
POP R1
POP R2
POP R3 /*dopo questa istruzione SP vale SP^5*/
RETURN

area stack:

SP^1	R0
	R1
	R2
	R3
	PC^{main}
A^4	#SUM
A^3	N
A^2	NUM
SP^{inizio}	...

area dati:

	...
#SUM	SUM
#N	N
#NUM[0]	NUM[0]
	...

...
NUM[N-1]
...

Le memorie

Le memorie si possono suddividere in ROM (Read Only Memory, memoria di sola lettura. NB: anche questa è ad accesso casuale, arbitrario) e RAM (Random Access Memory, memoria ad accesso casuale, arbitrario: il tempo impiegato per accedere ad un dato è indipendente dalla sua posizione). Le RAM si possono ulteriormente scomporre in SRAM (Static RAM) e DRAM (Dynamic RAM).
Le memorie di tipo SRAM sono basate su FF. Sono memorie molto veloci e costose, organizzate per parole.
Le di tipo DRAM invece sono basate su microcondensatori, accumulatori di carica elettrica, ognuno dei quali gestisce un bit di memoria. Sono molto più lente della CPU. Sono impiegate per la memoria centrale di un computer. Una memoria basata su microcondensatori ha bisogno di essere rinfrescata con delle operazioni di refresh, altrimenti dopo un po' di tempo il suo contenuto varia. Un refresh in pratica è un'operazione di lettura senza immagazzinamento del dato letto. Ci sono dei circuiti appositi che si occupano di rinfrescare la memoria, chiedendole periodicamente la lettura di alcune locazioni da refreshare e disabilitando nello stesso momento l'uscita della memoria sul bus dati (con il controllo Output Enable, OE, posto a zero).

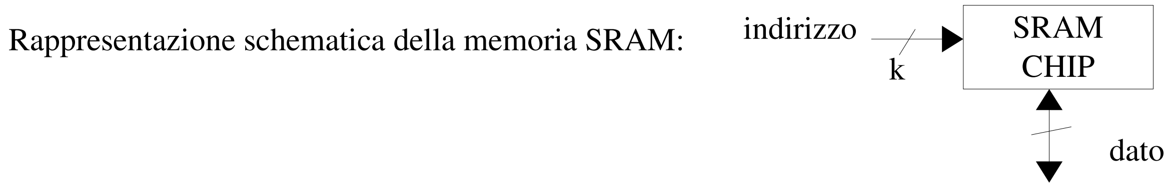
Tabella riepilogativa:

Tipo	*Organizzazione*	*Dimensione*	*Refresh*	*Bus Mux*	*Tempo di accesso*	*Uso*	*Elemento*	*Costo per bit*
SRAM	per word	ridotta	NO	NO	~1ns	Cache memory	FF	elevato
DRAM	per bit	elevata	SI	SI	~10ns	Main Memory	microcondensatore	ridotto

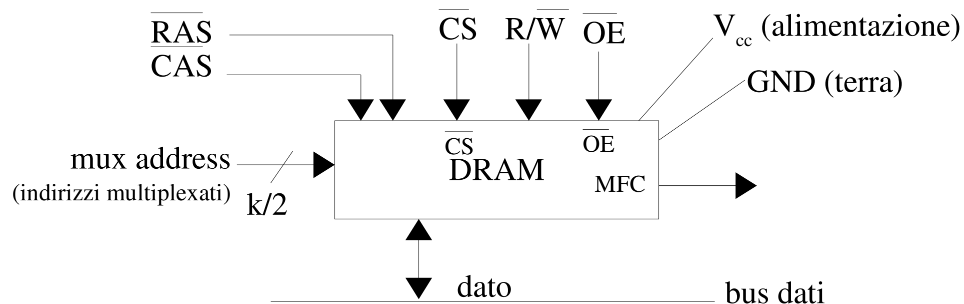
Note:
-Il tempo di accesso è il tempo che intercorre da quando un'operazione (lettura o scrittura) è stata richiesta fino a quando tale operazione è stata eseguita.
-C'è da notare che durante il refresh (che è, in pratica, una lettura interna) non è possibile accedere alla memoria dall'esterno e quindi il tempo di accesso nelle DRAM non è fisso, ma varia (perché se sto facendo il refresh devo attendere che finisca).
-In una memoria organizzata per bit posso indirizzare singoli bit.
-La dimensione di una memoria è $D=2^k \cdot m\, bit$ dove k=numero di linee di indirizzo e m=dimensione della parola.
-La dimensione di un integrato di memoria DRAM (quindi organizzata per bit) è $D=2^k \cdot 1\, bit$, dove con k si indica il numero di bit di indirizzo. Per poterla utilizzare ho quindi bisogno di k+1 piedini (k piedini servono per gli indirizzi, 1 piedino serve per i dati). Una memoria della stessa dimensione, ma

organizzata per word (ad esempio 1 word=8 bit), necessiterebbe di k-3 piedini per l'indirizzo e 8 piedini per i dati, per un totale di k+5 piedini (infatti si avrebbe $D=2^{k-3}\cdot 8=2^{k-3}\cdot 2^3=2^k\cdot 1\,bit$).
-bus MUX indica se è possibile fare il multiplexing degli indirizzi (vedi avanti).

Rappresentazione schematica della memoria SRAM:

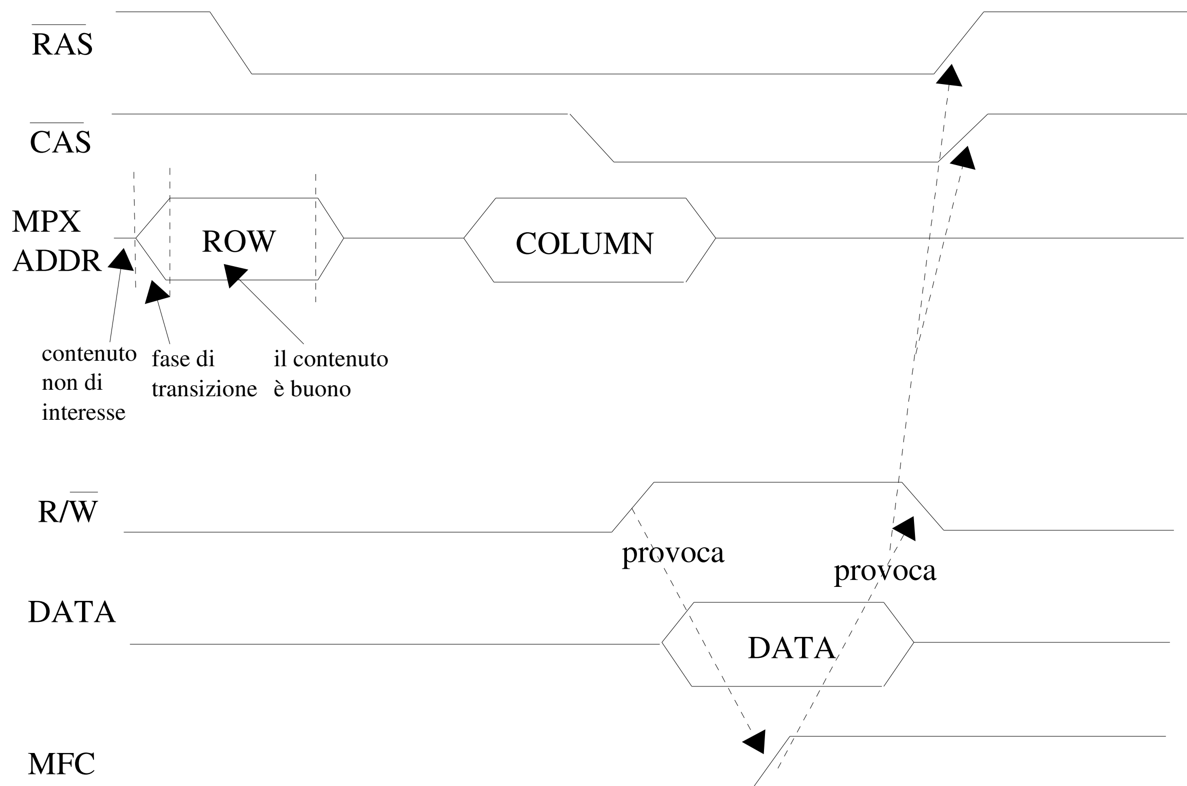


Nelle DRAM l'indirizzo viene mandato in due volte successive, così da risparmiare $\frac{k}{2}$ piedini (multiplexing degli indirizzi). Possiamo vedere la memoria centrale come una tabella, nella quale ogni cella può essere identificata da un indirizzo di riga ed uno di colonna. Quindi posso vedere l'indirizzo di una cella scomposto in due parti, indirizzo di riga e di colonna. Per indicare a quale indirizzo voglio andare a prelevare o scrivere dati prima invio l'indirizzo di riga ed attivo il controllo RAS (Row Address Strobe):
//? l'indirizzo ricevuto viene memorizzato in un registro interno ?//; successivamente invio l'indirizzo di colonna ed attivo il controllo CAS (Column Address Strobe). All'interno l'indirizzo viene ricomposto ed utilizzato. Con questo modo ho risparmiato in circuiteria (la metà dei piedini), ma ho sacrificato le prestazioni, in particolare il tempo di accesso è aumentato (raddoppiato).
Rappresentazione schematica DRAM:



dove (nota: CAS e RAS sono asseriti bassi):
- $\overline{CS}$ (chip select) è un segnale asserito basso (cioè la selezione del chip avviene quanto il segnale è zero) che permette di abilitare un chip. Senza questo segnale il chip non opera.
- $\overline{OE}$ (output enable) è un segnale asserito basso che indica se l'uscita del dato sul bus è abilitata.
- $R/\overline{W}$ indica se effettuare un accesso in lettura o in scrittura.
- MFC (Memory Function Completed), oppure READY, o ACK, indica se la memoria ha già compiuto o se sta ancora compiendo l'ultima operazione richiesta.
I segnali $R/\overline{W}$ e MFC sono detti segnali di handshaking. Viaggiano sul bus controlli.

Diagramma temporale dell'handshaking CPU memoria per la lettura di una locazione (<u>diagramma a caramella</u>):



Note:
-RAS e CAS sono asseriti bassi;
-l'indirizzo di riga viene posto su MPX ADDR e quando è pronto viene asserito RAS; lo stesso per l'indirizzo di colonna e CAS.
-quando entrambi gli indirizzi sono pronti viene generato il segnale di lettura; quando i dati richiesti sono pronti viene generato il segnale MFC e la CPU recupera dal bus dati i dati richiesti

esercizio “chip e banco” (1)

Si vuole costruire un banco di memoria B di dimensioni $D_B = 64\,M \cdot 8\,byte = 2^{26} \cdot 8\,byte$ (quindi il banco deve avere 26 linee di indirizzo e parole dati con 8 linee) a partire da dei chip C di dimensione $D_C = 16\,Mbyte$, con una parola dati di 2 linee.

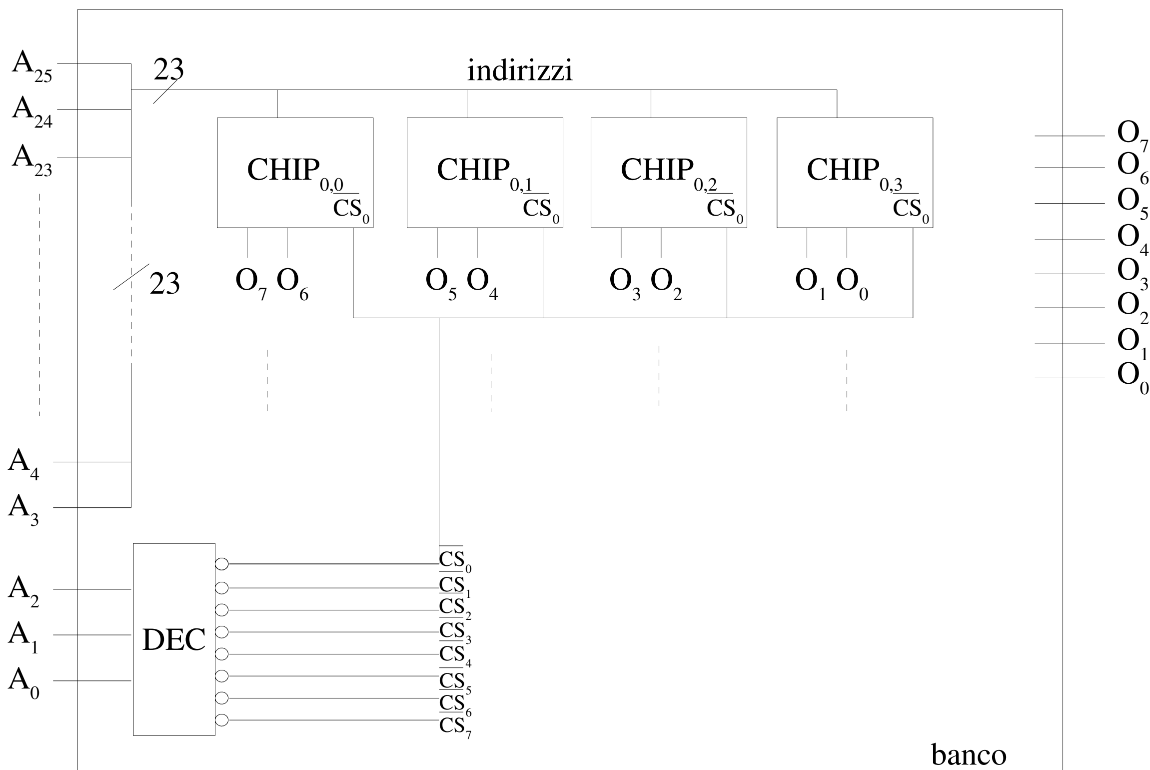
$D_C = 16\,Mbyte = 2^{24} = 2^{23} \cdot 2$, quindi ogni chip ha 23 linee di indirizzo a disposizione e 2 linee dati.

Troviamo il numero di chip necessari facendo

$$N_{chip} = \frac{D_B}{D_C} = \frac{2^{26} \cdot 8}{2^{23} \cdot 2} = 32\,chip$$

IDEA: delle 26 linee di indirizzo a disposizione 23 le collego ad ogni chip. I chip li suddivido in otto gruppi di 4 chip ciascuno. Ciascun gruppo è in grado di fornire in uscita una parola a 8 linee come richiesto (infatti ogni chip ha 2 linee di uscita ed in ogni gruppo ci sono 4 chip). Per selezionare quale gruppo fra gli otto presenti è necessario abilitare (CS) utilizzo le 26-23=3 linee residue disponibili con un decodificatore, con il quale posso abilitare una sola fra le 2^3=8 uscite (del decodificatore) disponibili. Ciascuna di queste otto uscite del decodificatore è collegata ad uno (uno solo!) degli otto gruppi di chip.

La schematizzazione parziale del banco potrebbe essere questa //? ci sarebbero da sistemare anche gli OE ?//:



Controllo se torna tutto:

ho $2^{23}\,indirizzi \cdot 2\,linee\,di\,uscita\,per\,ogni\,chip \cdot 2^5\,chip = 2^{29}\,byte = 2^{26} \cdot 2^3 = 2^{26} \cdot 8\,byte$.

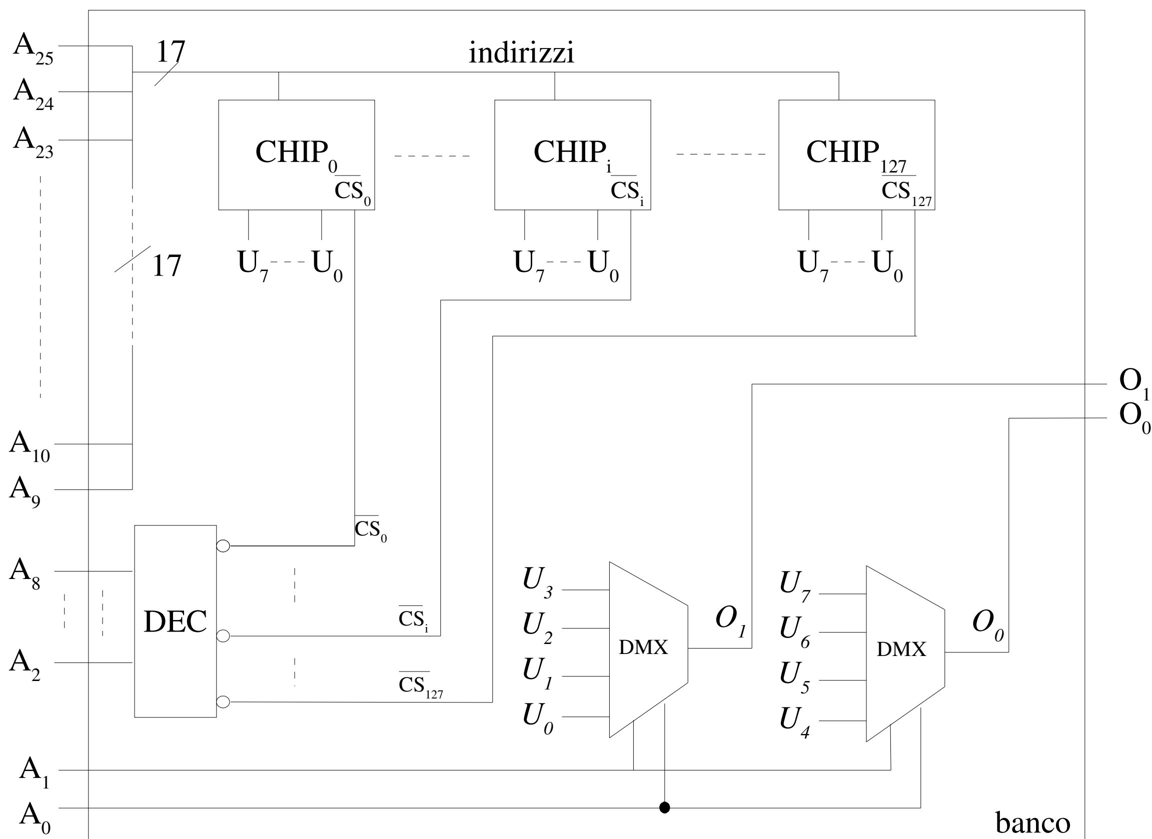
esercizio “chip e banco” (2)

Si vuole costruire un banco di memoria B di dimensioni $D_B = 64\,M \cdot 2\,byte = 2^{26} \cdot 2\,byte$ (quindi il banco deve avere 26 linee di indirizzo e parole dati con 2 linee) a partire da dei chip C di dimensione $D_C = 128\,K \cdot 8\,byte = 2^{17} \cdot 8\,byte$, quindi ogni chip ha 17 linee di indirizzo a disposizione e 8 linee dati. Trovo il numero di chip necessari facendo

$$N_{chip} = \frac{D_B}{D_C} = \frac{2^{26} \cdot 2}{2^{17} \cdot 8} = 128\,chip \quad .$$

IDEA: 17 delle 26 linee del banco le uso come linee di indirizzo da collegare ad ognuno dei 128 chip.
Mi rimangono 26-17=9 linee del banco da utilizzare.
7 di queste 9 linee le attacco ad un decodificatore, in modo da poter avere in uscita dal decodificatore 2^7=128 linee. Collego l'i-esima linea in uscita dal decodificatore all'i-esimo chip, all'ingresso del CS.
Mi avanzano ancora 2 linee, che utilizzo come linee di selezione per due multiplexer distinti. A ciascun multiplexer faccio arrivare 4 delle otto linee in uscita da ciascun chip (e le due linee di selezione), in modo da portare all'uscita di in ogni MPX una sola delle 4 linee entranti, così da avere in uscita dal banco le sole due linee richieste.

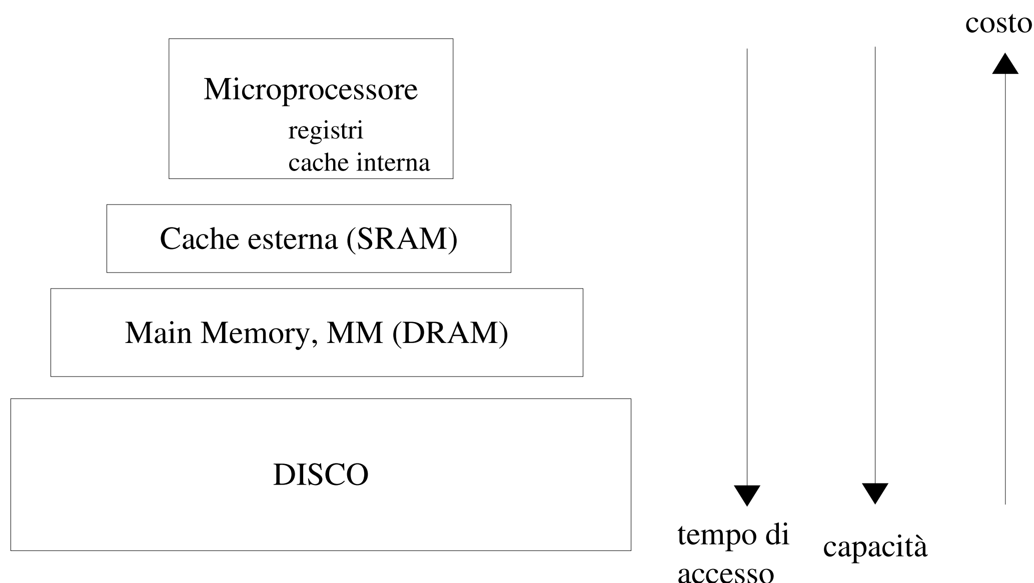
La schematizzazione parziale del banco potrebbe essere questa //? ci sarebbero da sistemare anche gli OE ?//:



quindi ho $2^{17}\,indirizzi \cdot 8\,linee\,dati\,per\,ogni\,chip \cdot 128\,chip = 2^{27} = 2^{26} \cdot 2\,byte$.

Gerarchie di memoria

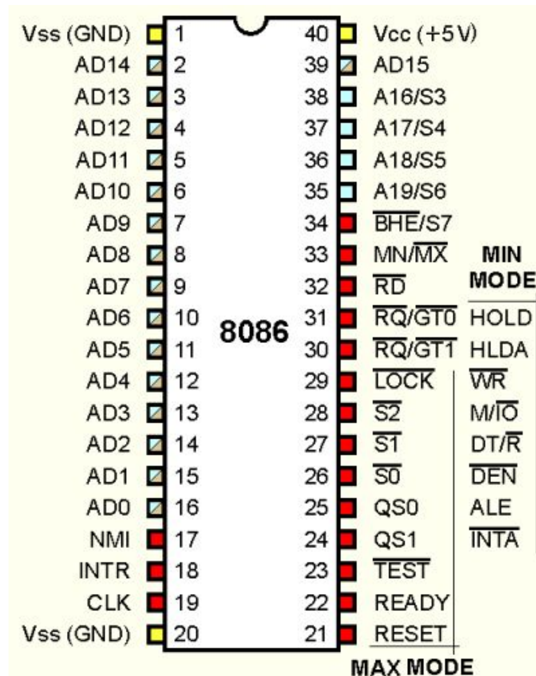
Le memorie possono essere viste in modo gerarchico:



Nell'esecuzione delle istruzioni è opportuno considerare due “fenomeni”:
-località spaziale: generalmente la successiva istruzione da eseguire è contigua all'istruzione attuale (lo stesso vale per i dati);
-località temporale: se ho fatto accesso ad un certo dato in un certo momento è probabile che faccia nuovamente riferimento a tale dato in un tempo breve (ad esempio nei cicli).
Per questi due fenomeni quando vado a prendere qualcosa che sta in memoria mi conviene prendere anche ciò che gli sta intorno. Queste informazioni aggiuntive vengono messe nella cache esterna, così se in breve tempo viene fatto riferimento a qualcosa presente in cache, lo prendo da lì.
CACHE MISS: accade quando richiedo un dato che non è presente nella cache;
CACHE HIT: accade quando richiedo un dato che è presente nella cache.
Con questi accorgimenti il tempo di accesso complessivo sarà:
$T_{acc}=h\cdot t_{cache}+m\cdot t_{mm}$, con h+m=1
dove h è la probabilità di un cache hit ed m è la probabilità di un cache miss.
Lo stesso ragionamento vale per cache esterna – memoria e disco vale anche per cache interna – cache esterna. In questo modo il tempo di accesso è percepito dal processore più basso di quello effettivo, con a disposizione una capacità più elevata.

Architettura 8086

L'8086 è un processore di tipo CISC realizzato da intel. Da esso discendono anche i moderni processori della intel.
L'8086 ha il bus indirizzi a 20 bit ed il bus dati a 16 bit.
L'8086 ha 40 piedini in tutto.
Per risparmiare piedini l'8086 condivide i 16 piedini destinati ai dati con i primi 16 dei 20 piedini destinati agli indirizzi (AD0 – AD15), quindi non può generare simultaneamente indirizzi e dati (ad esempio in fase di scrittura in memoria).
Al piedino INTR arrivano le richieste di interruzione (vedere I/O).
Il clock (CLK) dell'8086 è a 5MHz.



Modello di programmazione 8086

Il modello di programmazione è ciò che un programmatore assembler deve sapere su una certa macchina.
Bus dati: 16 bit => 1 word=2byte=16bit
Bus indirizzi: 20 bit => 1M di indirizzi
Registri:

registri per i dati

AX (registro accumulatore)	AH	AL
BX (base register)	BH	BL
CX (registro contatore)	CH	CL
DX (data register)	DH	DL

Questi 4 registri (AX, BX, CX, DX) sono a 16 bit, ma vi si puà accedere anche un byte alla volta, usandoli nella loro parte bassa (ad esempio AL) o nella loro parte alta (ad esempio AH), entrambe quindi di 8 bit.
In generale ogni registro può essere usato per un sacco di diverse istruzioni. È vero però che alcuni registri sono sfruttati meglio se usati con alcune istruzioni; ci sono inoltre altre istruzioni che richiedono obbligatoriamente l'uso di certi registri (ad esempio l'istruzione LOOP che fa uso di CX).
BX può essere usato insieme ad un registro di indice come base+offset per indirizzare un certo dato di memoria.
Esempi: mov AL,17 mette in AL il numero binario 00010001, che si indica come 10001b. Se cerco di spostare in AL un numero codificato su un numero maggiore di 8 bit l'assemblatore mi dà un errore.

registri puntatori o indici
SP (stack pointer)
BP (base pointer)
SI (registro indice)
DI (registro indice)

Anche questi 4 registri sono a 16 bit, ma non possono essere utilizzati a mezzo, cioè non si può utilizzare la loro parte bassa o la loro parte alta.
Il registro SP, se non viene modificato, punta all'ultima cella nello stack che contiene un valore utile e significativo. Il registro BP può essere utilizzato per accedere direttamente ad una certa locazione di memoria dello stack, senza bisogno di fare operazioni di push e di pop (quindi come una normale locazione di memoria dell'area dati). I registri SI e DI invece possono essere utilizzati come registri indice. esempi: mov ax, vect[si] sposta in AX quello che c'è nella locazione di memoria di indirizzo ds:vect+si (vedi avanti)

registri di segmento
CS (code segment register)
DS (data segment register)
ES (extra segment register)
SS (stack segment register)

Anche questi sono registri a 16 bit che non sono divisi in parte bassa e parte alta. Contengono l'indirizzo dell'area di memoria indicata dal nome di ciascun registro (quindi CS punta al segmento del codice, DS punta al segmento dati, SS punta al segmento stack; ES può essere usato ad esempio per puntare al PSP).

Altri due importanti registri sono IP e FLAGS (PSW). Sono entrambi a 16 bit. IP (instruction pointer) punta alla prossima istruzione da eseguire (è un offset rispetto a CS), mentre la PSW contiene vari bit di stato del processore, molti dei quali vengono modificati durante le operazioni aritmetiche e di comparazione per essere utilizzati nelle istruzioni di salto condizionato.

ancora sulla segmentazione

siccome il bus indirizzi è a 20 bit ed i registri sono tutti a 16 bit, mancano 4 bit per poter avere un indirizzo completo all'interno di un registro! il “problema” è risolto così: quando ci si deve riferire ad un dato che se ne sta in memoria è necessario specificarne l'indirizzo con due valori, l'indirizzo dell'inizio del segmento e l'offset (chiamato EA, Effective Address, forse impropriamente) del dato rispetto al segmento. Con questi due valori si può ottenere l'indirizzo fisico, PA (Physical Address) del dato desiderato. Indicando con SR il registro che contiene il segmento e con EA l'offset, l'operazione per trovare PA si scrive così: PA=SR:EA e significa PA=SR*16 + EA
Notare che i valori di SR e EA stanno su 16 bit. Moltiplicare SR per 16 (in base decimale) equivale a

shiftarlo a sinistra di 4 posizioni (2^4=16). Quindi viene fuori una cosa del genere:
SR 0000 +
0000 EA=
PA su 20 bit
Spesso EA deve essere calcolato, in base al modo di indirizzamento che si sta utilizzando.
Una volta che è stato fissato uno SR iniziale posso quindi individuare un segmento che inizia a SR*16 e finisce a SR * 16 + (2^{16} – 1).
Notare che i segmenti possono essere tranquillamente sovrapposti.
Con lo schema di memoria segmentata il programma in linguaggio macchina ha indirizzi relativi, che non devono così essere modificati dal loader del sistema operativo: il SO basta infatti che carichi i registri di segmento con i giusti valori, senza modificare il codice del programma.
L'assemblatore nell'assegnare gli indirizzi ipotizza che il programma venga caricato dall'inizio della memoria.
Se invece l'indirizzamento fosse assoluto il loader dovrebbe, al caricamento del programma, modificare tutti gli indirizzi che ci sono.

Con la memoria segmentata il PC diventa quindi PC=CS:IP.

I registri SP e BP sono associati a SS (ad esempio mov ax, [bx]); IP è associato a CS; le altre operazioni sui dati in genere sono associate con DS (ad esempio mov ax, [bx] oppure mov ax, var[di]).

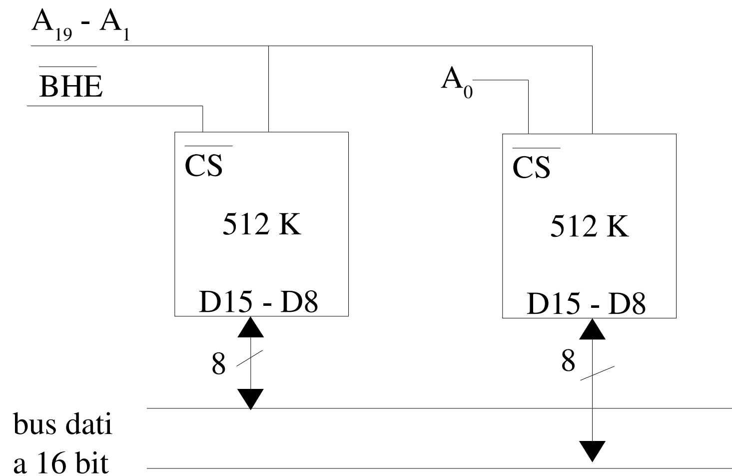
Notare che, ad esempio, posso avere all'interno del mio programma anche più di un segmento di stack (anche se solo uno sarà "attivo" in un certo istante): per passare da un segmento all'altro devo copiare in SS l'indirizzo dello stack desiderato. Lo stesso vale per i dati.

Little-endian, big-endian

Nel momento in cui voglio accedere ad un singolo byte della memoria non ho alcun problema: ogni byte ha un determinato indirizzo. Quando invece voglio accedere ad una parola, cioè a 2 byte insieme, è necessario stabilire una convenzione: qual è l'indirizzo della parola? è l'indirizzo del primo byte o quello del secondo byte?
Ci sono due alternative: la convenzione little-endian (usata nell'8086) nella quale l'indirizzo di una word è l'indirizzo del suo byte meno significativo; la convenzione big-endian, nella quale l'indirizzo di una word è l'indirizzo del suo byte più significativo.
Quindi nell'8086 (little endian) per scrivere la parola
A2B3 (esadecimale)
devo scrivere
B3 A2
in due locazioni di memoria consecutive, supponendo che gli indirizzi della memoria crescano verso destra. Mentre nel caso big-endian, per la stessa parola avrei scritto
A2 B3
sempre supponendo che gli indirizzi crescano verso destra.
La convenzione little-endian è adottata anche nella codifica degli operandi nelle istruzioni di macchina (ad esempio per gli indirizzi e per i valori).

Accesso alla memoria ad indirizzi pari e ad indirizzi dispari

La memoria nell'8086 è organizzata così:



BHE sta per Byte High Enable.
Leggere un singolo byte, sia che si trovi ad indirizzo pari che ad indirizzo dispari, richiede ovviamente un solo ciclo di bus.
Anche per leggere una parola ad indirizzo pari basta un solo ciclo di bus, per il fatto che l'indirizzo dei due byte da leggere varia solo per il bit A_0, quindi basta che A_0 e BHE siano messi a zero (e quindi entrambi i $\overline{CS}$ asseriti) per prendere contemporaneamente i due byte consecutivi richiesti.
Se invece la parola da leggere si trova ad indirizzo dispari, allora occorrono due cicli di bus, per il fatto che gli indirizzi dei due byte da leggere differiscono sia per A_0 che per A_1, bit che è in comune con entrambi i chip. Nel primo ciclo di bus si ha A_0 posto a 1 e BHE posto a 0, nel secondo ciclo di bus si ha A_0 posto a 0 e BHE posto a 1.
Tabella riassuntiva:

azione	$\overline{BHE}$	A_0
Trasferimento di un byte di indirizzo pari	1	0
Trasferimento di un byte di indirizzo dispari	0	1
Trasferimento di una parola di indirizzo pari	0	0
Trasferimento di una parola di indirizzo dispari	0	1
	1	0

Quindi per trasferire una parola di indirizzo dispari impiego il doppio del tempo impiegato per trasferire una parola di indirizzo pari.

Struttura interna dell'8086

L'8086 può essere visto come suddiviso in due parti: la BIU (Bus Interface Unit) e la EU (Execution Unit). Schema: Bucci, pg 338.
La EU si occupa dell'esecuzione delle istruzioni.
La BIU si occupa del (pre)fetch delle istruzioni: nella BIU è contenuta la coda di prefetch, composta da 6 byte (nell'8086 le istruzioni possono essere lunghe da 1 a 6 byte) all'interno dei quali vengono caricate le istruzioni successive a quella correntemente in esecuzione. In questo modo viene realizzato una specie di parallelismo, perché la BIU agisce quando il bus non è impiegato dalla EU, rendendole disponibile una nuova istruzione da eseguire non appena è stata eseguita l'istruzione corrente.
Ovviamente se siamo in presenza di una istruzione di salto la coda di prefetch deve essere svuotata totalmente al fine di caricarci le istruzioni giuste.
Tenuto conto del fatto che l'8086 può leggere una parola di indirizzo pari in un solo ciclo di bus, nel caso in cui la BIU si trovi a leggere istruzioni da un indirizzo dispari (ad esempio a seguito di un'istruzione di salto), in tale occasione viene fatta la lettura di un solo byte, così da poter poi continuare a leggere parole intere di indirizzo pari in un solo ciclo di bus.

Assembler 8086

nota preliminare: questa parte, data la complessità del linguaggio, è particolarmente soggetta ad imprecisioni/errori/mancanze.
Il linguaggio assembler è strettamente legato all'architettura dell'elaboratore con il quale si deve lavorare. Il programma assembler scritto in forma testuale viene tradotto ("assemblato") in codice macchina da un assemblatore: ad ogni istruzione scritta in assembly corrisponde una istruzione di macchina.
Nell'8086 il sorgente di un programma assembler si divide in segmenti, che rispettano la segmentazione prevista nella memoria (dati, stack, codice). Se si vuole, un programma può essere diviso in moduli (file diversi). La divisione in moduli non è presa in considerazione in questa mini-guida.
All'interno di ciascun segmento ci possono stare direttive (comandi per l'assemblatore) ed istruzioni vere e proprie.
I commenti sono preceduti da un punto e virgola.
Il linguaggio non è sensibile a maiuscole e minuscole, quindi i nomi di parole chiave ed identificatori possono essere usati indistintamente con caratteri minuscoli, maiuscoli o misti.

Per definire un segmento si scrive:

```
nomesegmento segment [altra roba]
        ....
nomesegmento ends
```

dove nomesegmento è un identificatore per il segmento (per gli identificatori sono possibili solo caratteri alfanumerici ed il tratto basso "_", ed inoltre il primo carattere non può essere un numero).
[altraroba] è facolativa: ci si può mettere, ad esempio, l'allineamento (per specificare se il segmento deve essere allocato a partire da un indirizzo multiplo di 1, di 2, di 4, ...), la combinabilità (per

specificare se e come il segmento deve essere combinato con altri segmenti presenti in altri moduli [file] del programma; per lo stack in genere si mette “stack”, senza le virgolette), il nome della classe (tutti i segmenti non lo stesso nome di classe [che deve essere racchiuso fra apici singoli] vengono messi in posizioni di memoria contigue).

All'interno del segmento dati

La definizione di variabili si fa all'interno del segmento dati, in questo modo:
identificatore dimensione valore
dove:
-identificatore (facoltativo) sarà utilizzato per referenziare la variabile;
-dimensione è una parola chiave scelta fra DB (Define Byte, alloca dati di un byte), DW (Define Word, alloca dati di 2 bye), DD (Define Doubleword, alloca dati di 4 byte), DQ (Define Quadword, alloca dati di 8 byte), DT (Define Tenbyte, alloca dati di 10 byte).
-valore può essere un numero oppure una stringa oppure altre parole chiave per effettuare delle ripetizioni di valori. Per quanto riguarda i numeri, se non è indicato niente vengono interpretati come numeri in base 10; se invece alla fine del numero si mette una “h” il numero viene interpretato come numero in base 16 (nota: se il numero inizia con una lettera [A, ..., F] è necessario anteporre uno zero, per evitare ambiguità con gli identificatori); con una “b” il numero viene interpretato come numero binario. Per le stringhe è necessario racchiuderle fra apici singoli. Per effettuare delle ripetizioni si usa questa sintassi:
numero_di_ripetizioni dup (valore)
dove valore può essere un numero o una stringa.
Esempi:
var1 db 'ciao' ;alloca 4 byte, nel primo c'è la codifica ASCII di 'c', nel secondo la codifica di 'i', ...
var2 db 0ah ;alloca 1 byte contenente 00001010 (binario), con EA=4
var3 dw 12 ;alloca 2 byte, il primo contenente 00001100, il secondo 00000000 (per via del little endian), con EA=5
var4 db 10 dup ('a') ;alloca 10 byte, tutti e dieci contenenti la codifica ASCII del carattere 'a', con EA=7

La definizione di costanti potrebbe essere fatta in qualsiasi parte del codice, non necessariamente all'interno del segmento dati. Per definire una costante si scrive
identificatore equ valore
L'assemblatore ogni volta che trova identificatore lo sostituisce con valore, e poi inizia ad assemblare.
NOTA BENE: definendo una costante non viene allocato spazio in memoria!! di fatto la costante è un dato immediato!

All'interno del segmento di stack

Nel segmento di stack, se serve, si allocano i byte necessari (in genere senza una etichetta)
    db 10 dup ('stack')
Inizializzando lo stack con la parola 'stack' è più facile riconoscerlo poi in fase di debug.
NOTA: nell'8086 lo stack cresce verso indirizzi più bassi. Quindi alla definizione di uno stack di un

materiale sotto licenza FDL – Free Documentation License - si veda la prima pagina e l'ultima sezione

tot parole SS punta alla prima parola mentre SP punta all'ultima; via via che faccio operazioni di push SP decresce fino ad arrivare a zero, mentre invece aumenta facendo delle operazioni di pop.

All'interno del segmento di codice

È necessario istruire l'assemblatore su quali siano i segmenti di codice, dati e stack. Per fare questo è necessario usare la direttiva assume, in questo modo:
assume cs:nome_segmento_codice, ds:nome_segmento_dati, ss:nome_segmento_stack
Una delle prime istruzioni da inserire deve spostare nei registri DS e SS l'indirizzo del rispettivo segmento, in questo modo:

```
inizio:
        mov ax, nome_segmento_dati
        mov ds, ax
```

e lo stesso per ss. Non è possibile fare il passaggio diretto mov ds, nome_segmento.
"inizio:" è un'etichetta, e servirà successivamente per la direttiva END.
Le etichette sono un identificatore seguito dai due punti. Identificano l'indirizzo della successiva istruzione. Si posso usare, ad esempio, nelle istruzioni di salto.
A seconda del tipo, NEAR o FAR, è possibile riferirsi ad una etichetta solo all'interno del segmento di codice in cui è definita (near) oppure da tutto il modulo (far). Se non è specificato niente, le etichette vengono definite di tipo near. Per riferirsi ad una etichetta di tipo near l'assemblatore codifica nell'istruzione ad esempio una istruzione di salto incondizionato) solo il suo scostamento all'interno del segmento corrente, mentre per riferirsi ad una etichetta di tipo far l'assemblatore codifica nell'istruzione sia l'indirizzo del segmento che il suo scostamento. Inoltre per ogni etichetta è possibile specificarne la visibilità, che indica se è possibile riferirsi all'etichetta anche da altri moduli. Se niente è specificato la visibilità è solo interna al modulo.
Per specificare tutte queste opzioni le etichette si devono indicare in un altro modo, così:
nome_etichetta label distanza
dove nome_etichetta è il nome che sarà utilizzato per riferirsi all'etichetta, label è una parola chiave, distanza è NEAR oppure FAR.

Alla fine del programma, sempre all'interno del segmento di codice, è bene ricordarsi di scrivere le istruzioni per far tornare il controllo al sistema operativo, altrimenti continua l'esecuzione con tutti i byte che vengono trovati in memoria nelle posizioni successive.
Con il DOS ci sono due possibilità:
-l'uso di un'interruzione, con le istruzioni:

```
        mov ah, 4ch ;sposta in ah il valore 4ch
        int 21h
```

La prima istruzione mette il codice dell'operazione richiesta nel registro ah, la seconda istruzione chiama l'interruzione 21h che, fra le altre funzioni (specificate con il contenuto di alcuni registri, in questo caso di ah), permette di fare il ritorno al sistema operativo.
-con l'uso dell'istruzione ret: in questo caso sono necessari diversi accorgimenti:
riferendosi a quello scritto sopra, al posto dell'etichetta "inizio:" si deve mettere una procedura di tipo FAR (vedi più avanti per le procedure), in questo modo
nome_procedura proc far

dove nome_procedura è il nome che si vuol dare alla procedura (ad esempio “main”). Subito dopo è necessario fare il push nello stack dell'indirizzo del segmento nel quale si trova il codice da eseguire per fare il ritorno al so e poi (in questo ordine) dell'offset (all'interno del segmento) dell'istruzione da eseguire. In pratica si tratta di mettere nello stack l'indirizzo di memoria, diviso in CS e IP, dell'istruzione che deve essere eseguita per fare il ritorno al so. Questa istruzione si trova nel primo byte del PSP (vedere più avanti per il PSP), quindi è necessario inserire subito dopo la dichiarazione della procedura

```
push ds
xor ax,ax
push ax
```

Questo perché al caricamento del programma in memoria il registro DS (o anche il registro ES) contiene l'indirizzo a partire dal quale il programma è stato caricato (quindi punta al primo byte del PSP). A questo punto per far tornare il controllo al so è sufficiente usare l'istruzione

```
ret
```

che, siccome la procedura è stata definita FAR, prende dallo stack (con due istruzioni di pop), nell'ordine, il valore di IP ed il valore di CS. NOTA: lo stack deve essere nelle stesse condizioni di partenza, altrimenti vengono prelevati dei valori sbagliati (quindi se nel programma sono stati fatti 2 push, devono esserci anche 2 pop). In questo modo l'istruzione eseguita successivamente è quella che si trova nel primo byte del PSP, che finalmente provvede a far tornare il controllo al so.

la direttiva END

in fondo al programma, dopo la chiusura del segmento di codice, si deve indicare all'assemblatore qual è la prima istruzione che deve essere eseguita all'avvio del programma. Questo si fa con la direttiva END, in questo modo:

```
end punto_di_partenza
```

dove punto_di_partenza è un identificatore (ad esempio un'etichetta o il nome di una procedura) che è stato definito da qualche altra parte nel programma e a partire dal quale il programma deve iniziare. Nota: inserire un carattere di ritorno a capo (premere invio...) dopo questa direttiva, altrimenti il masm non capisce.

macro e procedure

Quando del codice si deve ripetere più volte all'interno del programma è opportuno usare una macro o una procedura.

Una macro è un pezzo di codice al quale ci si può riferire con un identificatore. Prima di assemblare il tutto, l'assemblatore sostituisce ogni chiamata a macro incontrata con il codice della macro, ricopiandocelo, e sostituendo al posto dei parametri formali i parametri attuali (per accorgersi di questo basta dare un'occhiata al file .LST che viene creato dopo aver assemblato). Quindi la definizione di una macro non occupa spazio in memoria, ma lo spazio occupato sarà dato dal numero di chiamate moltiplicato per la dimensione del codice della macro, proprio per il fatto che tutto il codice della macro viene ripetuto interamente per ogni chiamata (un po' come accade con le costanti [EQU]).

Per definire una macro si fa così:

nome_macro macro [elenco_parametri_formali]

```
        ; qui ci va il codice della macro
        endm
```

dove nome_macro è il nome che servirà per utilizzare la macro, mentre elenco_parametri_formali è l'elenco dei parametri da utilizzare per la macro, separati da una virgola, che vengono utilizzati all'interno della macro.
Per utilizzare una macro invece si fa così (supponiamo ad esempio di essere dentro al segmento di codice):

```
        nome_macro [elenco_parametri_attuali]
```

Quando l'assemblatore incontra questa riga la cancella e al suo posto ci mette tutto il codice contenuto nella macro, sostituendo ogni parametro formale che incontra con il rispettivo parametro attuale.
Ad esempio, la macro per mostrare del testo a video può essere fatta così.

```
mostra_testo macro str
        push dx         ;salvo i registri nello stack per
        push ax         ;       non sporcarli
        mov dx, offset str      ;metto in dx l'offset della stringa da stampare
        mov ah, 9h      ;codice dos per mostrare del testo
        int 21h         ;interruzione dos che si occupa di varie cose, tra cui mostrare del testo a video
        pop ax          ;recupero i registri
        pop dx          ;       dallo stack
        enm
```

Quando all'interno del segmento di codice chiamo la macro con

```
        mostra_testo stringa_da_stampare
```

l'assemblatore sostituisce questa riga con il codice contenuto nella macro, sostituendo inoltre str con stringa_da_stampare. Vista questa sostituzione "grezza" fatta dall'assemblatore c'è da stare attenti ad esempio ad utilizzare delle etichette all'interno di una macro, perché si rischia di ripetere la stessa etichetta più di una volta (se la macro viene utilizzata più di una volta) e questo dà errore.
nota: la stringa di testo da stampare deve obbligatoriamente terminare con il carattere dollaro '$'.

Per quanto riguarda le procedure invece la cosa è diversa: non ne viene ricopiato tutto il codice nel punto in cui vengono chiamate, come invece accade con le macro, bensì viene passato il controllo al codice della procedura, controllo che viene poi restituito all'istruzione successiva a quella che ha fatto la chiamata.
Per definire una procedura (//? si deve essere dentro un segmento di codice ?//), si fa così:

```
nome_procedura proc [distanza]
        ; qui ci va il codice della procedura
        ret
nome_procedura endp
```

dove nome_procedura è il nome che si userà per chiamare la procedura, distanza (facoltativo) può essere NEAR oppure FAR (di default è NEAR). Se la procedura è NEAR per effettuare il ritorno (con l'istruzione RET, che viene codificata dall'assemblatore come RETN) dallo stack verrà prelevato (con un pop) solo l'IP; se invece la procedura è FAR per effettuare il ritorno (con l'istruzione RET, che viene codificata dall'assemblatore come RETF) dallo stack viene prelevato l'IP ed il CS, in questo ordine.

Per chiamare una procedura, all'interno del segmento di codice, si usa l'istruzione CALL in questo modo:

    call nome_procedura

Se la procedura prevede dei parametri (passati in particolari registri oppure nello stack, dipende dalla funzione), vanno indicati prima della chiamata. Nel caso di parametri passati nello stack c'è da star attenti al fatto che la call esegue, prima di passare il controllo alla procedura, il push dell'IP oppure, se la procedura è FAR, esegue il push del CS e dell'IP (in questo ordine). Quindi all'interno della procedura al momento di prelevare i parametri occorre fare attenzione di non prelevare per sbaglio gli indirizzi di segmento o di offset inseriti nello stack dalla CALL.

modi di indirizzamento

Per indicare i dati in una istruzione ci sono varie possibilità, chiamate modi di indirizzamento:
-indirizzamento di registro: le operazioni coinvolgono due registri, ad esempio

    mov ax, bx

che sposta il contenuto di bx nel registro ax.
-indirizzamento immediato: il dato viene codificato come parte dell'istruzione stessa, ad esempio in

    mov ax, 20h

20h è un dato immediato. La stessa cosa accade se si usano delle costanti [EQU]. Per questo dato sono necessari 2 byte (per il fatto che la destinazione, AX, è un registro a 16 bit).
- indirizzamento diretto di memoria: nell'istruzione viene indicato l'indirizzo di memoria nel quale si trova il dato, ad esempio nell'istruzione

    mov ah, var   (o anche mov var, ah che fa l'inverso)

var è stata definita ad esempio così: var db 104 e rappresenta l'indirizzo (anzi, l'EA rispetto al segmento dati) nel quale si trova il valore 104. Quindi nell'istruzione viene codificato l'EA (NON il valore 104!!!), usando 16 bit.
-indirizzamento di memoria indiretto usando un registro

    mov ax, [bx]   (o anche mov [bx], ax che fa l'inverso)

che sposta in ax il contenuto della parola il cui EA è scritto in bx.
-indirizzamento di memoria relativo usando un registro

    mov ax, var[si]   (o anche mov var[si], ax che fa l'inverso)

che sposta in ax il contenuto della parola il cui EA è dato da var+il contenuto del registro si. Nell'istruzione viene memorizzato l'EA (16 bit) di var.
-indirizzamento di memoria con registro di base e registro di offset

    mov ax, [bx][si]   (o anche mov [bx][si], ax che fa l'inverso)

che sposta in ax la parola di EA dato da bx+si.
-indirizzamento di memoria relativo con registro di base ed indice

    mov ax, var[bx][si]   (o anche mov var[bx][si], ax che fa l'inverso)

che sposta in ax la parola di EA dato da var+bx+si. Nell'istruzione viene memorizzato anche l'EA di var //? (16 bit) ?//.

Quando c'è da usare un registro di base si può usare BX o BP. In genere per gli indici invece si usa SI, DI, CX. Quando viene utilizzato un registro “illegale” l'assemblatore dà errore.
Nel memorizzare gli EA ed i dati immediati viene adottata la convenzione little endian (prima il byte

meno significativo e poi quello più significativo).

PSP (program segment prefix)

È un blocco di 100h posizioni che viene fatto precedere dal dos, durante il caricamento in memoria, al programma vero e proprio. Nel PSP ci vengono messe alcune informazioni utili al programma, ad esempio nella prima posizione c'è l'istruzione per far tornare il controllo al so, mentre nelle posizioni da 81h a 0FFh ci sono i caratteri digitati nel prompt del dos dopo il nome del programma. L'ultimo carattere presente è sempre il carattere di invio (CR, di codice ASCII 0Dh).

alcune istruzioni, lunghezza/codifica delle istruzioni

alcune istruzioni

Un elenco delle più comuni istruzioni, con una breve ed incompleta descrizione, è il seguente:

-istruzioni di trasferimento dati:

mov destinaizione, sorgente ;copia sorgente in destinazione. sorgente e destinazione possono essere registri, dati di memoria oppure (solo la sorgente...) dati immediati. Non tutte le combinazioni potrebbero essere ammesse.

push sorgente ;mette nello stack il dato indicato

pop destinazione ;estrae un dato dalla testa dello stack e lo mette in destinazione

in al, porta ;preleva da una periferica di input un dato e lo mette in al (la periferica è identificata da porta, che può essere anche un dato immediato)

out porta, ah ;scrive su una periferica di output il dato presente in ah

-istruzioni aritmetiche:

add destinazione, sorgente ;memorizza in destinazione il risultato di destinazione+sorgente

sub destinazione, sorgente ;memorizza in destinazione il risultato di destinazione-sorgente

inc dato ;incrementa di 1 il dato e sovrascrive il risultato

dec dato ;decrementa di 1 il dato e sovrascrive il risultato

cmp op1, op2 ;compara i due operandi, andando a modificare i bit della PSW (ad esempio l'overflow flag, il carry flag, il parity flag, il sign flag, lo zero flag). Di fatto fa op1-op2 ma non salva il risultato in op1. utile prima di una operazione di salto condizionato

mul dato ;esegue una moltiplicazione. a seconda del tipo di operando cambia il registro in cui viene preso l'altro fattore ed il registro in cui va il risultato. il risultato è a 8 o a 16 bit, sempre a seconda dell'operando. consultare il manuale.

div dato ;esegue una divisione. a seconda del tipo di operando cambiano i registri in cui viene messo il risultato ed il resto. consultare il manuale

shl dato, num ;esegue lo shift a sinistra di dato di num posizioni. uno shift equivale a moltiplicare per due,

shr dato, num ;esegue lo shift a destra di dato di num posizioni. uno shift a destra equivale a dividere per due.

-operazioni logiche (sono molto efficienti)

not dato ;esegue il NOT, bit a bit, del dato e sovrascrive il risultato

and op1, op2 ;esegue l'AND, bit a bit, di op1 e op2 e salva il risultato in op1

or op1, op2 ;esegue l'OR, bit a bit, di op1 e op2 e salva il risultato in op1

xor op1, op2 ;esegue l'XOR, bit a bit, di op1 e op2 e salva il risultato in op1

(molto utile per azzerare un registro: xor ax, ax)
test op1, op2 ;esegue l'AND, bit a bit, di op1 e op2 ma non salva il risultato, modifica soltanto i bit della PSW
-istruzioni di controllo
jmp etichetta ;(salto incondizionato) passa il controllo al codice che si trova subito dopo etichetta
le istruzioni di salto condizionato esaminano i bit della PSW e se la condizione che esprimono è verificata passano il controllo all'etichetta indicata, altrimenti all'istruzione successiva. alcuni esempi di queste istruzioni sono je (jump if equal), jz (jump if zero), jng (jump if not grater), jle (jump if less or equal), jne (jump if not equal), ecc..., ognuna seguita dal nome dell'etichetta a cui saltare se la condizione è vera.
loop etichetta ;decrementa il registro cx, se il risultato è diverso da zero fa un salto ad etichetta, altrimenti passa all'istruzione successiva
call nome_procedura ;passa il controllo a alla procedura indicata. se la procedura in questione è di tipo near allora prima di passare il controllo la call salva nello stack l'offset dell'istruzione successiva, alla quale dovrà essere ridato il controllo alla fine della procedura, se invece è di tipo far allora la call salva nello stack, in questo ordine, l'indirizzo dell'attuale segmento di codice e l'offset dell'istruzione successiva.
ret ;fa tornare il controllo al programma chiamante eseguendo il pop dallo stack dell'ip (se la procedura attuale è di tipo near), oppure di ip e cs (se la procedura attuale è di tipo far)
iret ;ritorna il controllo quando è finita una routine che gestisce una interruzione
int numero_interruzione ;chiama l'interruzione indicata
nota: dove è possibile inserire un operando di tipo immediato è possibile mettere anche un carattere (indicato fra singoli apici, sempre in modo immediato), anche in operazioni aritmetiche o logiche, per il fatto che la stringa viene considerata dall'assemblatore semplicemente come un numero binario composto delle cifre che corrispondono alla sua codifica ascii. Ad esempio questo codice

```
mov ax, 'a'
sub ax, 'A'
add ax, 'R'
```

alla fine mette in ax il carattere 'r'. Notare inoltre che nella tabella ascii i caratteri minuscoli differiscono da quelli maiuscoli per un solo bit.

Quando ci si riferisce ad un dato di memoria tramite un EA viene di solito scelto automaticamente, sulla base dell'istruzione e dei registri coinvolti, qual è il registro che contiene l'indirizzo del segmento e con il quale trovare l'indirizzo fisico del dato. Ad esempio se ci si riferisce ad un dato del segmento dati l'associazione viene fatta con il registro DS. È possibile fare anche altre associazioni: per farlo è necessario indicare il registro contenente l'indirizzo del segmento prima dell'EA, facendolo seguire dai due punti, ad esempio mov es[di], al .

<u>codifica delle istruzioni</u>
Nell'8086 le istruzioni sono codificate su un numero di byte variabile da 1 a 6. Ogni istruzione ha una

diversa codifica in base al modo di indirizzamento degli operandi. Già dal primo byte dell'istruzione è possibile determinarne la lunghezza complessiva.
Esempio: l'istruzione add ha questa codifica

campo	opcode	***d***	***w***	***mod***	***reg***	***r/m***	***displacement***
num. di bit	6	1	1	2	3	3	8 oppure 16

-l'opcode è 000000;
-d è 1 quando reg è destinazione;
-gli altri campi sono scelti in base alla tabella riportata qui sotto;
-se uno dei due operandi è di memoria allora è presente anche l'ultimo campo;
-//? se c'è qualche operando immediato allora è presente un altro campo ancora, di 1 o 2 byte a seconda del tipo. ?//

r/m	***mod***					
	00	01	10	11		reg
				w=0	w=1	
000	(bx)+(si) ds	(bx)+(si)+(d8) ds	(bx)+(si)+d16 ds	al	ax	000
001	(bx)+(di) ds	(bx)+(di)+d8 ds	(bx)+(di)+d16 ds	cl	cx	001
010	(bp)+(si) ss	(bp)+(di)+d8 ss	(bp)+(si)+d16 ss	dl	dx	010
011	(bp)+(di) ss	(bp)+(di)+d8 ss	(bp)+(di)+d16 ss	bl	bx	011
100	(si) ds	(si)+d8	(si)+d16 ds	ah	sp	100
101	(di) ds	(di)+d8 ds	(di)+d16 ds	ch	bp	101
110	d16 ds	(bp)+d8 ss	(bp)+d16 ss	dh	si	110
111	(bx) ds	(bx)+d8 ds	(bx)+d16 ds	bh	di	111

ad esempio l'istruzione add bx, var[si] viene codificata così:

campo	opcode	***d***	***w***	***mod***	***reg***	***r/m***	***disp-lo***	***disp-hi***
binario	000000	1	1	10	011	100	xxxxxxxx	xxxxxxxx
esadecimale	03			9C			xx	xx

mentre l'istruzione add bx, cx si codifica così:

campo	opcode	***d***	***w***	***mod***	***reg***	***r/m***
binario	000000	1	1	11	011	001
esadecimale	03			D9		

parole chiave offset e seg

La parola chiave offset serve a riferirsi allo scostamento di una certa variabile (//? in generale di un certo identificatore ?//) all'interno del suo segmento. Ad esempio:

```
mov bx, offset var
```

sposta in bx l'offset di var.
notare che scrivere

```
mov ax, var
```

fa la stessa cosa che scrivere le due istruzioni

```
mov bx, offset var
mov ax, [bx]
```

Quando il riferimento a var deve essere espresso più volte è meglio fare nel secondo modo, si hanno istruzioni più corte e minori cicli di bus.
La parola chiave seg permette di riferirsi all'indirizzo del segmento di una certa variabile (//? e in generale di un certo identificatore ?//). Ad esempio

```
mov bx, seg var
```

sposta in bx l'indirizzo del segmento in cui è definita var.

assemblare e debuggare

Per assemblare un programma è necessario prima di tutto procurarsi un assemblatore. Quello di microsoft è il masm.
Una volta preso il masm.exe ed il linker (link.exe) per assemblare un programma si deve scrivere
masm nomeprogramma
senza l'estensione, che deve essere .asm. Se dopo il nome si mette qualche virgola masm non chiederà altre informazioni per i nome degli altri file. Una volta fatto questo (se tutto è andato bene) si deve fare il link con
link nomeprogramma
senza l'estensione, seguito anche questo da qualche virgola per non essere interpellati su altri nomi.
Se anche il link va a buon fine tutto è pronto per lanciare il programma digitandone il nome.
Se ci sono problemi durante una delle due fasi viene riportato il tipo di errore ed il numero della riga sulla quale presumibilmente si trova l'errore.
Per il debug di può usare il programma “debug” già incluso nel dos.
Per avviarlo è necessario digitare
debug nomeprogramma.exe
Per eseguire una istruzione alla volta il comando è “t”. Per ogni istruzione eseguita viene visualizzato il contenuto dei registri e, sulla destra, il bit di stato, mentre in basso viene indicata la prossima

istruzione da eseguire, la sua posizione nella memoria e la sua codifica in macchina. Per eseguire un insieme di istruzioni (ad esempio una procedura intera od una interruzione oppure tutto un ciclo che usa l'istruzione “loop”) il comando è “p”. Per visualizzare la memoria si deve invece digitare “d segmento:offset”. Per far completare l'esecuzione del programma il comando è “g”. Per uscire il programma è “q”. Tutte i dati riportati sono in esadecimale.

Input / Output

Il colloquio fra la CPU ed un certo dispositivo di input/output avviene per mezzo di una interfaccia (in parte hardware ed in parte software) di I/O.
L'interfaccia realizza la comunicazione fra due dispositivi hardware: è composta da una parte operativa, che si occupa della temporizzazione e della memorizzazione dei dati, e della parte di controllo (driver), realizzata via software (utilizzando le istruzioni di macchina del processore, gira nell'elaboratore), che si occupa di gestire la parte operativa
In generale una interfaccia può essere di tipo seriale (se i bit sono trasmessi uno alla volta) o di tipo parallelo (se i bit sono trasmessi in parole).
Un esempio di interfaccia è l'interfaccia USB (Universal Serial Bus).

Un esempio di interfaccia potrebbe essere composta così:

Il componente che decodifica gli indirizzi riconosce gli indirizzi asseriti sul bus indirizzi ai quali la periferica deve rispondere. Questi particolari indirizzi si chiamano “porte”. Ci sono due diverse modalità per accedere ai dispositivi di I/O: se l'architettura prevede l'ingresso/uscita “mappato in memoria” allora per l'accesso ai dispositivi non sono previste istruzioni specifiche, ma si usano quelle utilizzate per la memoria (con gli indirizzi dei dispositivi: quindi memoria e I/O condividono il range di indirizzi disponibili); nell'altra modalità, con l'ingresso/uscita isolato (più diffusa, usata nell'8086), invece ci sono delle istruzioni specifiche per accedere ai dispositivi di I/O, con le quali vengono asserite determinate linee del bus controlli (IOWC oppure IORC, vedi avanti).
Tutti i dati scambiati fra la cpu e il dispositivo di I/O passano per il registro dati, quindi la cpu per scrivere un dato su una periferica di output non deve far altro che indirizzare il dato all'indirizzo (porta) giusto e scriverlo nel registro dati; sarà l'interfaccia a preoccuparsi del trasferimento del dato alla periferica. In un interfaccia di sola uscita il registro dati si può solo scrivere, in una interfaccia di sola entrata invece si può solo leggere, mentre se l'interfaccia è sia di entrata che di uscita allora si può sia leggere che scrivere, a seconda dei casi.

Il registro di stato, di sola lettura per la cpu, fornisce lo stato attuale della periferica. Lo stato, quando richiesto, in genere viene posto sull'LSB del bus dati (se si tratta di un solo bit). In genere per una periferica di solo input lo stato 1 indica alla cpu che è disponibile un nuovo dato da acquisire (ad esempio premendo un tasto sulla tastiera lo stato diventa 1 e la codifica del dato viene posta nel registro dati), mentre per una periferica di solo output in genere lo stato 1 indica che la periferica è pronta per ricevere un nuovo dato da trasferire (ad esempio un nuovo carattere da stampare, se si tratta di una stampante). Il registro di stato si può solo leggere.
Il registro controlli serve a settare il funzionamento generale del colloquio processore-interfaccia, tramite dei bit che indicano, ad esempio, la velocità di trasferimento dei dati oppure la modalità di funzionamento (a controllo di programma o basata sulle interruzioni). Il registro controlli in genere si può solo scrivere.

Inizialmente possiamo pensare di associare ogni registro ad una porta diversa. Questo però porterebbe ad uno spreco di porte, per il fatto che non tutti i registri hanno la necessità di essere sia letti che scritti. Nel caso dello schema sopra possono bastare due sole porte: 1 per il registro dati (che deve essere sia letto che scritto) ed un'altra per i controlli e per lo stato (uno deve essere solo letto e l'altro solo scritto, per cui basta una sola porta: si interviene su un registro o sull'altro a seconda del fatto che si richieda la lettura o la scrittura del registro).
In genere gli indirizzi di porta sono consecutivi.

Istruzioni assembler tipiche

Le istruzioni assembler per leggere e scrivere su dispositivi di input e output sono queste due (cpu 8088, dati a 8 bit):

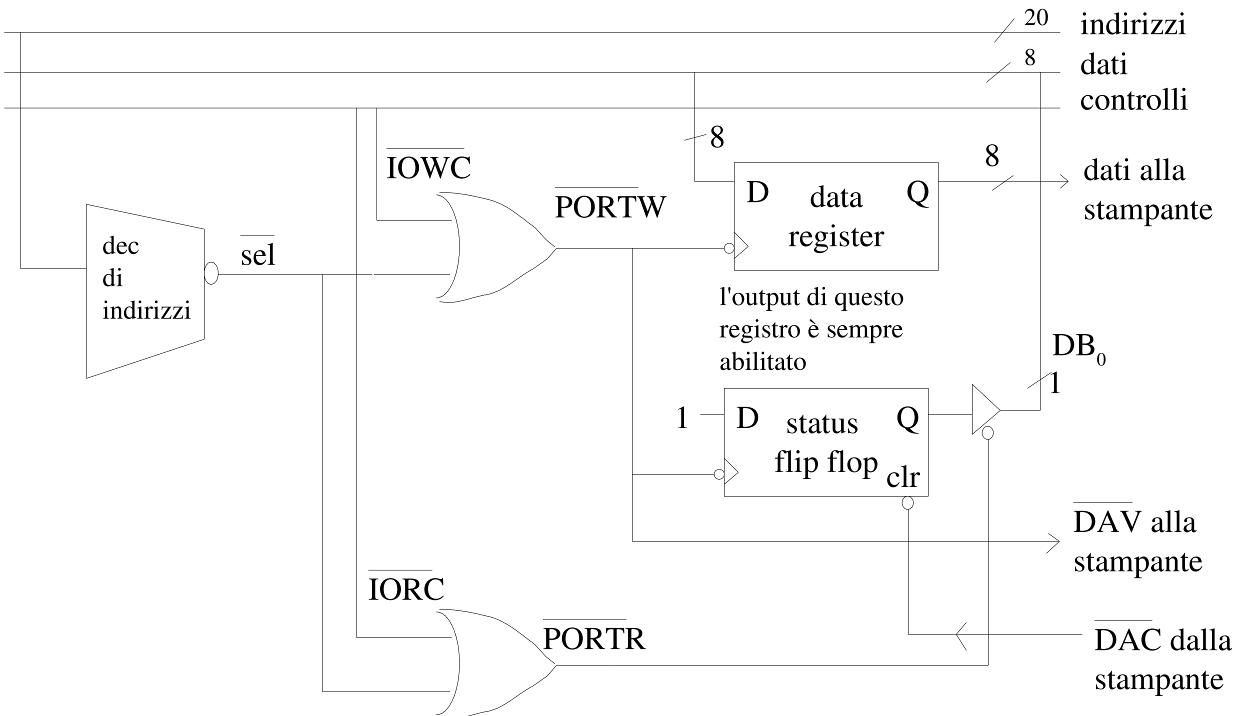
in al, porta ;legge in al dall'interfaccia che si trova all'indirizzo porta (immediato)

e

out porta, ah ;scrive sull'interfaccia che si trova all'indirizzo porta (immediato) il contenuto di ah

dove porta è l'indirizzo dal quale leggere/scrivere.
Altre due istruzioni assembler legate alle interruzioni sono CLI e STI.
L'istruzione CLI va a resettare un bit il bit IF (interrupt flag) della PSW. Quando IF è a zero il processore non serve la maggior parte delle richieste di interruzione (le interruzioni mascherabili). Le interruzioni non mascherabili (che giungono al piedino NMI) sono servite in ogni caso.
L'istruzione STI invece fa il contrario di CLI, settando a 1 il bit IF della PSW riabilitando quindi anche le interruzioni mascherabili.
Un altro bit della PSW è il bit TF. Se settato a 1 alla fine dell'esecuzione di ogni istruzione viene generato un interrupt, che provoca il salto ad una certa routine di interruzione, che in genere è un debugger.

Esempio di interfaccia di sola uscita a controllo di programma, senza il registro controlli (ad esempio verso una stampante che riceve un carattere alla volta; lo stato = 1 significa che il dispositivo è occupato), 8088, bus dati a 8 bit:



dove IOWC sta per I/O write control, IORC sta per I/O read control, PORTW sta per port write, PORTR sta per port read, DAV sta per data valid, DAC sta per data acknowledgement, DB_0 è il bit meno significativo del bus dati.
Funziona così:
-quando la cpu vuole stampare un carattere pone sul bus dati il codice del carattere da stampare, sul bus indirizzi l'indirizzo dell'interfaccia e asserisce (è asserito basso) il controllo IOWC. Così il dato presente sul bus dati viene memorizzato nel data register ed inviato alla stampante. Allo stesso tempo nel flip flop di stato viene memorizzato 1 (= dispositivo occupato) ed alla stampante si invia il segnale di DAV. A questo punto la stampante, ricevuto DAV, stampa il carattere e quando è pronta asserisce DAC, che ha l'effetto di porre a zero lo stato, così che la cpu possa inviare un altro carattere.
-quando la cpu vuole leggere lo stato della stampante (per sapere se è pronta o meno a ricevere un altro carattere) pone sul bus indirizzi l'indirizzo dell'interfaccia ed asserisce IORC. Così il dato presente nel flip flop viene messo sul bus dati e viene ricevuto dalla cpu, che lo legge facendo un AND del dato con 1.
In questo esempio l'interfaccia è mappata su un unico indirizzo di porta.

La parte di controllo (software) di questa stampante potrebbe essere ad esempio così:
---------------chiamata alla funzione di stampa
```
        mov si, offset buffer   ;mette in si l'offset dell'area di memoria contenente i caratteri da
stampare
        mov cx, n               ;mette in cx il numero di caratteri da stampare
```

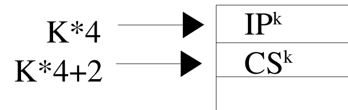
```
        call stampa           ;chiama la procedura di stampa

---------------funzione di stampa
stampa proc far

polling: in al, port
        and al, 1             ;mantengo il solo bit dello stato (LSB), tutti gli altri li azzero
        jnz polling           ;interroga l'interfaccia fino a quando è pronta per ricevere dati

        mov ah, [si]          ;mette in ah il dato da stampare
        out port, ah          ;stampa il dato
        inc si
        loop polling          ;continua fino a quando sono stati stampati tutti i caratteri del buffer
```

Questa modalità di controllo si chiama modalità di trasferimento a controllo di programma. Ha un grave difetto: il processore se ne sta nel polling (interroga la periferica per sapere se è pronta per ricevere un altro dato) per la maggior parte del tempo.

Una alternativa migliore è usare una **gestione sotto controllo di interruzione (interrupt esterni)**, in modo che sia la stampante ad avvertire la cpu quando ha finito di stampare un carattere; nel frattempo la cpu, invece che attendere la fine della stampa, fa altro. Quindi in generale con lo schema ad interrupt è il dispositivo che avverte il processore quando ha finito la scrittura di un dato o quando c'è un nuovo dato pronto da trasferire. Quando un dispositivo genera una richiesta di interruzione il processore può decidere cosa fare: servire l'interruzione o lasciarla in attesa. In generale l'interruzione viene servita non appena è terminata l'esecuzione della corrente istruzione di macchina. Per adattare l'interfaccia esemplificata sopra con lo schema ad interrupt è sufficiente collegare la linea di controllo INTR all'uscita $\bar{Q}$ (che nella figura non è rappresentata) del flip flop di stato. È anche possibile, se si vuole, aggiungere un flip flop di controllo (identificato da un altro numero di porta) di sola scrittura che permetta di scegliere se il funzionamento dell'interfaccia deve essere ad interrupt o meno (tramite un bit, messo in AND con l'uscita di $\bar{Q}$: se il bit vale zero allora il controllo ad interrupt è disabilitato, se vale 1 è abilitato).
Per sapere se c'è da servire una interruzione oppure no, la UC ogni volta che ha terminato l'esecuzione di una istruzione di macchina va a controllare il piedino INTR per vedere se qualcuno ha richiesto una interruzione; nel caso che INTR sia asserito la UC controlla qual è il dispositivo che ha chiesto l'interruzione e decide se servirla o meno. Il segnale di interruzione viaggia sul bus controlli (dall'interfaccia verso il processore).
Quando il processore riceve ed accetta la richiesta di interruzione interrompe la routine che sta eseguendo, passa il controllo alla routine che si occupa di gestire l'interruzione e alla fine riprende il lavoro che era stato interrotto, in modo trasparente per la routine interrotta (mantenendo il contenuto di tutti i registri).

Per stabilire l'identità del dispositivo che ha richiesto l'interruzione il dispositivo pone sul bus dati un codice k.

Il **vettore delle interruzioni** è un vettore che viene caricato nella prima parte della memoria; contiene gli indirizzi (segmento e offset) delle routine che si occupano della gestione delle interruzioni di ogni dispositivo. Usando il vettore delle interruzioni l'interfaccia che richiede l'interruzione deve porre il proprio tipo k (interrupt type, una sequenza 8 bit) sul bus dati. Da questa sequenza, con una semplice formula, si può trovare l'indirizzo in cui si trovano il CS e l'IP della routine che si occupa di gestire l'interruzione. La locazione in cui si trova l'IP della routine che gestisce l'interruzione è data da 4*k, mentre il CS si trova nella locazione di indirizzo 4*k+2. Le routine di interruzione sono di tipo FAR.

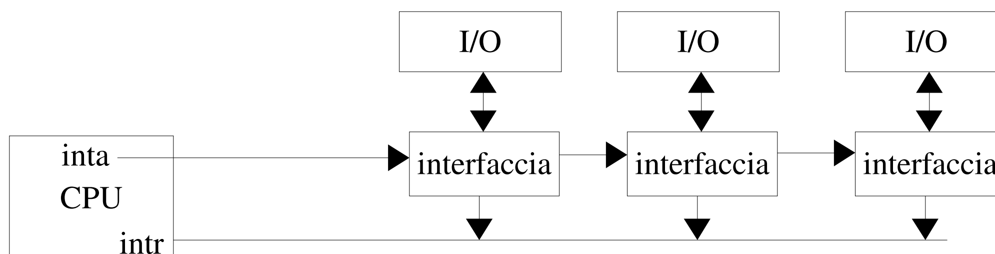


<u>Schema generale di una interruzione</u>:
-l'interfaccia asserisce la richiesta di interruzione
-dopo l'istruzione corrente è stata completata la cpu invia l'ack
-l'interfaccia che ha richiesto l'interruzione, tenuto conto di eventuali priorità, pone il proprio tipo k sul bus dati
-psw, cs e ip sono salvati nello stack
-i bit IF e TF sono messi a zero, per evitare che qualcun altro interrompa in questo momento
-ip e cs vengono aggiornati con gli indirizzi contenuti rispettivamente nelle locazioni di indirizzo 4*k e 4*k+2
-ha inizio la routine che gestisce l'interruzione
-viene riabilitata la possibilità di servire altre interruzioni (con STI)
-l'istruzione IRET (ritorno da interruzione) esegue il pop di ip, cs e psw dallo stack
-il controllo viene riportato al programma interrotto

Una routine di interruzione può essere interrotta a sua volta da qualche altra interruzione.

Con il metodo delle interruzioni vettorizzate è semplice sostituire una routine di interruzione con un'altra, oppure spostarla in un'altra locazione di memoria: è sufficiente aggiornare solo il suo indirizzo all'interno del vettore delle interruzioni.

<u>Gestione delle priorità (interruzioni vettorizzate)</u>
Può accadere che due o più dispositivi richiedano un'interruzione nello stesso momento. Sorge il problema di quale delle due richieste gestire. Questo problema è risolto adottando una politica delle priorità. Un modo di gestire le priorità è quello con lo schema **daisy-chain**, nel quale il dispositivo a più alta priorità è quello più vicino alla cpu:



funziona così: una periferica richiede una interruzione tramite intr; se la cpu vuole gestire l'interruzione asserisce la linea INTA. Il segnale inta viene "filtrato" da tutte le interfacce, a partire da quella più a sinistra (a maggiore priorità, più vicina alla cpu) verso quella più a destra (a minore priorità): quando la prima interfaccia riceve inta controlla se è stata lei a richiedere l'interruzione o meno. Se è stata lei allora non fa propagare inta, altrimenti lascia propagare inta alla periferica di destra, che fa la stessa cosa, e così via.
La gestione delle priorità con la daisy-chain è poco flessibile (è hardware), perché si basa sulla vicinanza dei dispositivi alla cpu.

Una volta che la catena è arrivata all'interfaccia che ha fatto richiesta dell'interruzione, questa pone il proprio tipo k sul bus dati. La cpu riceve il tipo k e, con questo, trova le due celle del vettore delle interruzioni che contengono l'indirizzo (segmento e offset) della procedura che gestisce l'interruzione e, dopo aver salvato il contenuto di alcuni registri nello stack (psw, cs e ip), la chiama. Una volta che la routine di interruzione è stata completata viene ridato il controllo al programma interrotto. La macchina viene riportata allo stato precedente e l'esecuzione continua dal punto in cui era stata interrotta, in modo trasparente.
Gli altri registri che non sono memorizzati automaticamente (quindi tutti i registri tranne cs, ip e la psw) devono essere salvati e ripristinati dalla routine di interruzione.

Un altro modo, più flessibile, per gestire la priorità e l'identità delle richieste di interruzione è tramite il componente hardware **PIC** (programmable interrupt control). Il PIC riceve tutte le singole richieste di interruzione dalle interfacce e invia alla cpu il segnale intr. Quando la cpu risponde con l'inta, il pic sceglie il dispositivo con più alta priorità, in base alle regole fornite dal programmatore che lo ha programmato. Quindi la gestione delle priorità diventa programmabile via software, mentre rimane di tipo hardware la gestione dell'identità dei dispositivi.

Un altro modo per gestire le priorità delle interruzioni è chiamato **polled interrupt**. Funziona così: la richiesta di interruzione di un dispotivo alla cpu viene fatta normalmente, sulla linea INTR (come nella daisy-chain). In questo caso però il processore ha una sola routine di interruzione, che si occupa di interrogare in sequenza tutti i dispositivi in grado di richiedere un'interruzione, al fine di trovare l'interfaccia che effettivamente ne ha fatto richiesta. Quando la cpu trova il dispositivo che ha richiesto l'interruzione allora chiama la procedura corretta. La priorità di un certo dispositivo è quindi determinata dall'ordine con il quale la routine che gestisce le interruzioni interroga i vari dispositivi (quello interrogato prima ha una priorità maggiore).

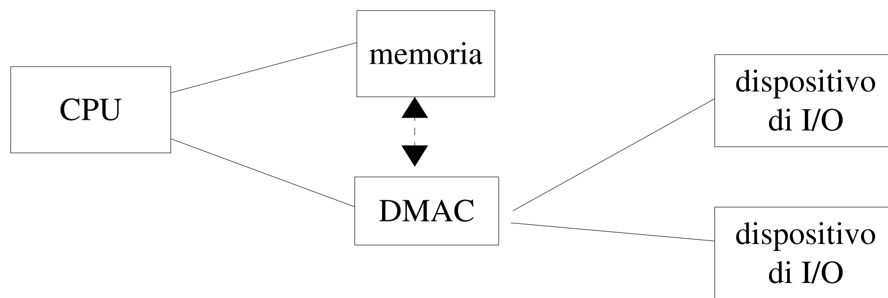
nota sulla simultaneità delle interruzioni: la cpu campiona il valore di INTR a tempi fissi (alla fine dell'esecuzione di ogni istruzione), quindi vengono considerate simultanee quelle richieste che arrivano fra un campionamento ed il successivo, anche se di fatto arrivano in due istanti diversi.

<u>nota: classificazione delle interruzioni</u>
Le interruzioni si possono suddividere in tre categorie
-interruzioni esterne: sono causate dai dispositivi che colloquiano con la cpu; si verificano in modo del

tutto asincrono rispetto all'esecuzione del programma; sono utilizzate per gestire le operazioni di I/O.
-eccezioni: sono causate da situazioni anomale che accadono durante l'esecuzione di un programma. Un esempio tipico è il tentativo di dividere un numero per zero, oppure di eseguire del codice macchina non permesso. Ovviamente non sono predicibili, in quanto dipendono dallo stato della macchina al momento dell'esecuzione; si verificano in modo sincrono rispetto al programma in esecuzione.
-trappole: sono generate da istruzioni apposite presenti nel programma (ad esempio l'istruzione INT); si verificano in modo sincrono e sono predicibili. Le trappole sono quindi particolari istruzioni di salto.

DMA (direct memory access)

Per rendere più efficienti i trasferimenti fra memoria centrale e periferiche di input/output (ad esempio l'hard disk) si può prevedere l'accesso diretto alla memoria di alcuni dispositivi, rendendoli in grado di generare indirizzi sul bus indirizzi (il dispositivo funziona da bus mater), sotto il controllo di un dispositivo (il controllore DMA, DMAC).
Il dispositivo che vuole funzionare come bus master deve ottenere l'autorizzazione dalla cpu, che ha un ruolo predominante nell'accesso al bus. Quando il dispositivo richiede l'accesso al bus il DMAC fa la richiesta del bus alla cpu; se la cpu autorizza l'utilizzo del bus al DMAC allora gli invia un apposito segnale; a questo punto il DMAC effettua i trasferimenti richiesti dalla periferica e, quando ha finito, disasserisce la linea di richiesta del controllo del bus, che torna alla cpu.



GNU Free Documentation License

GNU Free Documentation License
Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been

arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated

version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the

original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of thisLicense "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.