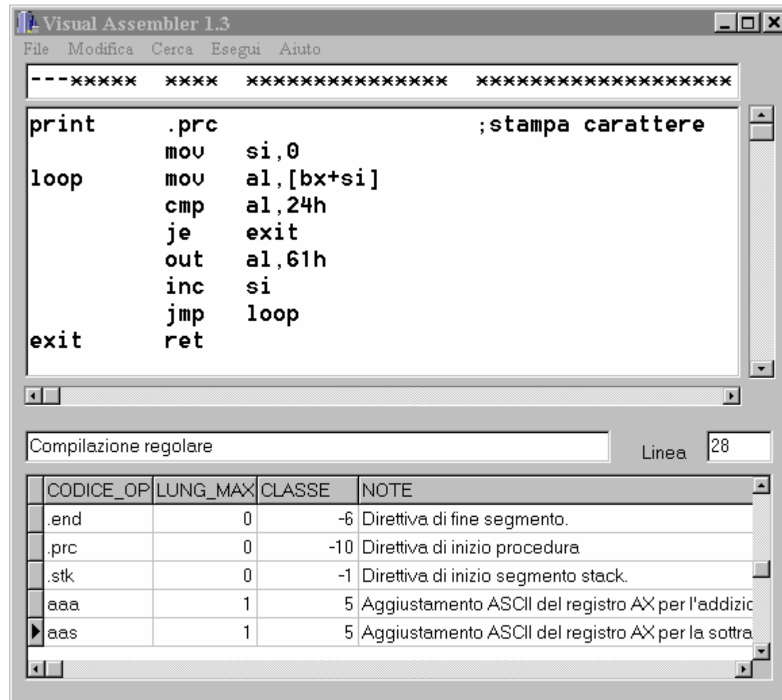
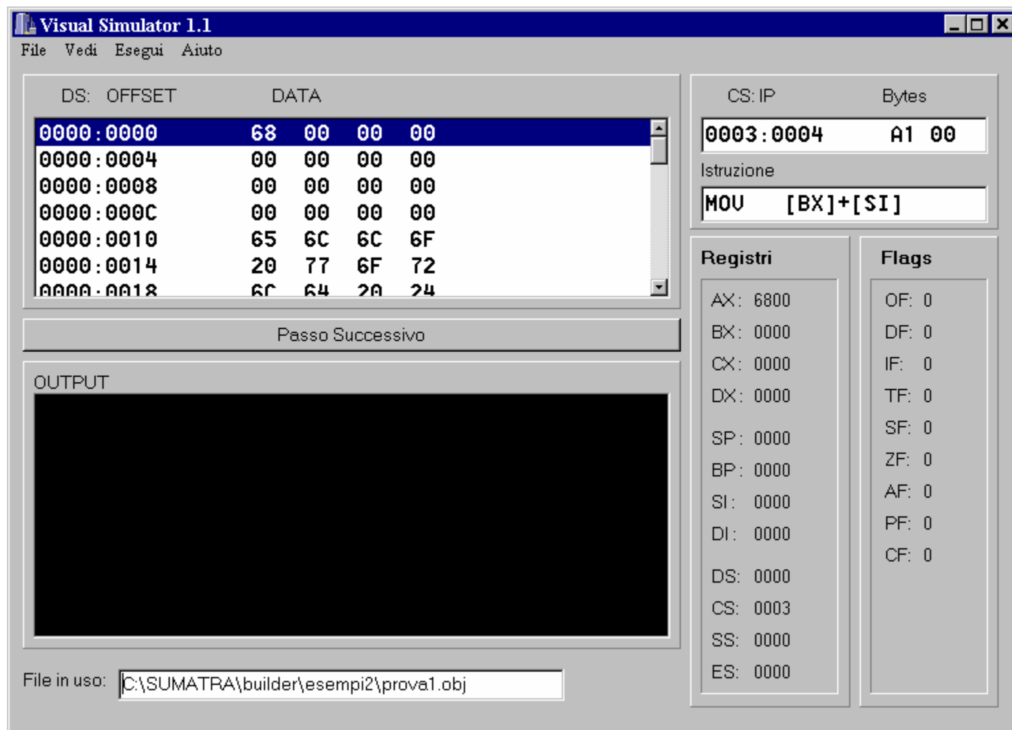
Politecnico di Bari - Sistemi di Elaborazione I

Arcieri Francesco 512155Y

Visual Assembler 1.3 per 8086



Visual Simulator 1.1 per 8086



1 Visual Assembler 1.3

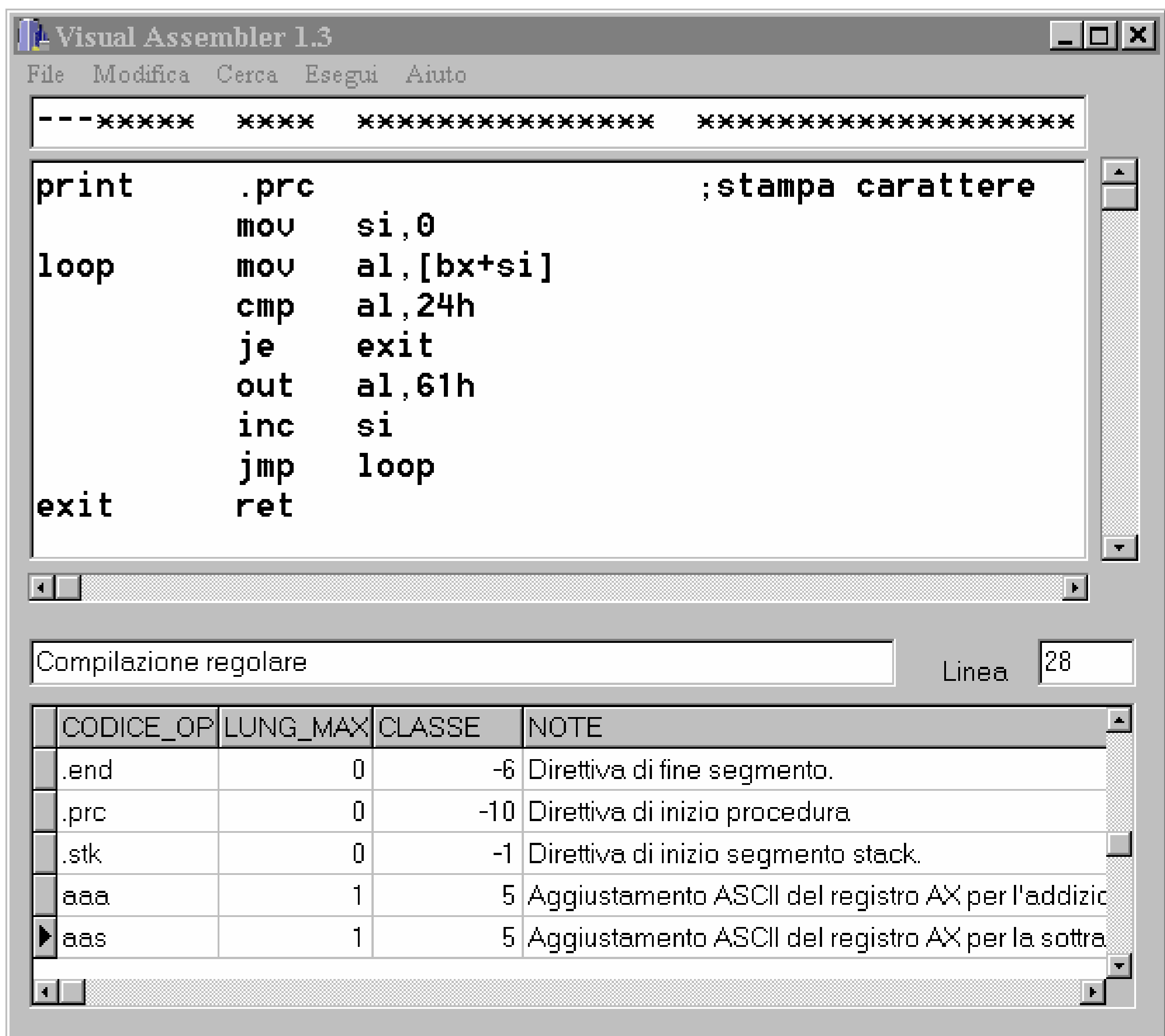


Figura 1: Una schermata dell'assemblatore

1.1 Generalità sull'assemblatore

1.1.1 Caratteristiche principali

L'assemblatore è stato realizzato **per ambiente Windows** tramite il linguaggio:

Borland C++ Builder 3 v1.0

La scelta è ricaduta su questo linguaggio per svariati motivi:

- Il C è un linguaggio che riunisce i migliori elementi dei linguaggi ad alto livello (come il BASIC o il Pascal) con le possibilità di controllo e la flessibilità del linguaggio Assembler.
- Il C++ è un linguaggio di programmazione orientato agli oggetti ed è quindi possibile sfruttare a pieno tutte le caratteristiche di quest'ultimi (Incapsulazione,polimorfismo ed ereditarietà).
- Il C++ Builder è un linguaggio di programmazione "visuale", permette cioè di realizzare complete interfacce grafiche sotto Windows in tempi brevi.
- L'assemblatore è stato realizzato secondo **l'approccio a due passi**, in modo tale da risolvere efficacemente il problema delle referenze a etichette non ancora definite.
- L'assemblatore ha una buona **gestione degli errori** e fornisce sia il tipo di errore verificatosi che la linea di codice dove è stato rilevato.
- L'assemblatore presenta una **lista dei codici operativi** con relativa descrizione.

Esaminiamo le altre caratteristiche descrivendo le funzionalità presenti nel menù a tendine:

Menù FILE

- **Nuovo:** Permette la creazione di un nuovo file assembler.
- **Apri:** Permette l'apertura di un file assembler precedentemente salvato.
- **Salva:** Permette di salvare il file assembler che si sta editando.
- **Stampa:** Permette di stampare il file assembler che si sta editando tramite la stampante di sistema.
- **Stampante:** Permette la modifica delle proprietà di stampa della stampante di sistema.
- **Exit:** Permette l'uscita dall'assemblatore.

Menù MODIFICA

- **Taglia:** Permette di eliminare, per poi inserire in un altro punto tramite il comando Incolla, il testo selezionato.
- **Copia:** Permette di copiare, per poi inserire in un altro punto tramite il comando Incolla, il testo selezionato.
- **Incolla:** Permette di inserire una porzione di testo selezionata precedentemente tramite il comando Copia.
- **Cancella:** Permette di eliminare il testo selezionato.

Menù CERCA

- **Trova:** Permette di trovare, se c'è, all'interno del file assembler che si sta editando, una data parola.

Menù ESEGUI

- **Assembla:** Assembla il file Assembler che si sta editando. Vengono creati il file Listing e il file Oggetto.

Menù AIUTO

- **Aiuto:** Fornisce informazioni sull'utilizzo del programma.
- **Informazioni:** Fornisce informazioni sul programma.

1.1.2 Regole lessicali

Le righe di codice prodotte nell'assemblatore devono seguire la seguente sintassi:

[<Etichetta>] <Mnem> [<Op>] [<,Op>] [<Commento>]

dove:

- **Etichetta:** è una stringa di caratteri di dimensioni variabili tra 4 e 8.
- **Mnem:** è il nome mnemonico di una qualsiasi istruzione dell'8086.
- **Commento:** è una stringa di caratteri di lunghezza arbitraria.
- **Op** rappresenta l'operando.

Vediamo le regole per specificare operandi e indirizzamento:

1. Nel campo operandi ***non si possono utilizzare caratteri blank*** (spazio, tab,..). Se si vuole specificare per un operando l'indirizzamento immediato bisogna far precedere il valore desiderato con il carattere # (cancelletto).

Esempio:

ADD AX,#5

2. Se si vuole specificare per un operando ***l'indirizzamento diretto*** bisogna scrivere l'identificatore dell'etichetta, o il numero relativo al displacement senza alcun carattere delimitatore.

Esempio:

ADD BX,105

3. Se si vuole specificare per un operando ***l'indirizzamento indiretto con base o indice*** bisogna scrivere tra parentesi quadre [ ] un registro valido (BX,SI,DI).

Esempio:

MOV CX,[SI]

4. Se si vuole specificare per un operando ***l'indirizzamento indiretto con base + indice*** bisogna utilizzare la seguente sintassi:

[RegBase+RegIndice]

Esempio:

ADD CX, [BP+SI]

ADD CX,[SI+BP] **->** ERRORE

ADD CX, [BP+ SI] **->** ERRORE

5. Se si vuole specificare per un operando l'indirizzamento indiretto con base o indice + piazzamento si deve utilizzare la seguente sintassi:

[RegBase/Indice + Spiaz]

dove RegBase/Indice é uno dei registri BX,BP,SI,DI.

Esempio:

MOV AX, [BP+29]

6. Se si vuole specificare per un operando l'indirizzamento indiretto con base + indice + ***spiazzamento*** si deve utilizzare la seguente sintassi:

[RegBase + RegIndice + Spiaz]

Esempio:

SUB DX, [BX+SI+15

1.2 Assemblaggio

1.2.1 Schema Generale

Lo scopo dell'assemblatore è tradurre il codice di un file SORGENTE, scritto in linguaggio assembler, in linguaggio macchina fornendo quindi in uscita un file OGGETTO. L'assemblatore dovrà fornire in uscita anche il file LISTING che conterrà indicazioni sugli indirizzi associati a variabili e dati.

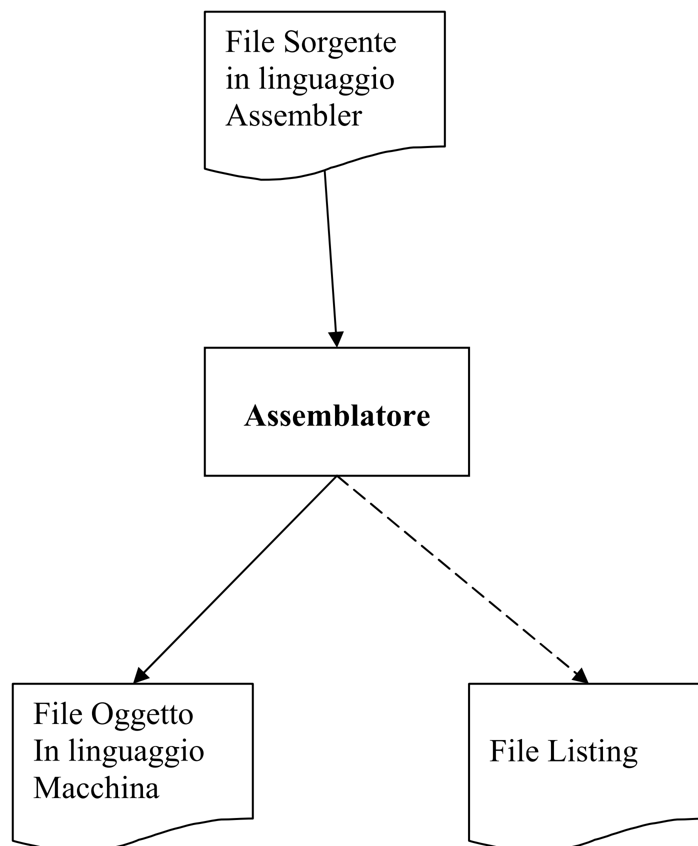


Figura 2: Input/Output dell'Assemblatore

Per risolvere il problema delle referenze a etichette non ancora definite, si è utilizzato all'approccio a due passi nella realizzazione dell'assemblatore.

Il flow Chart seguente mostra il corpo principale del programma:

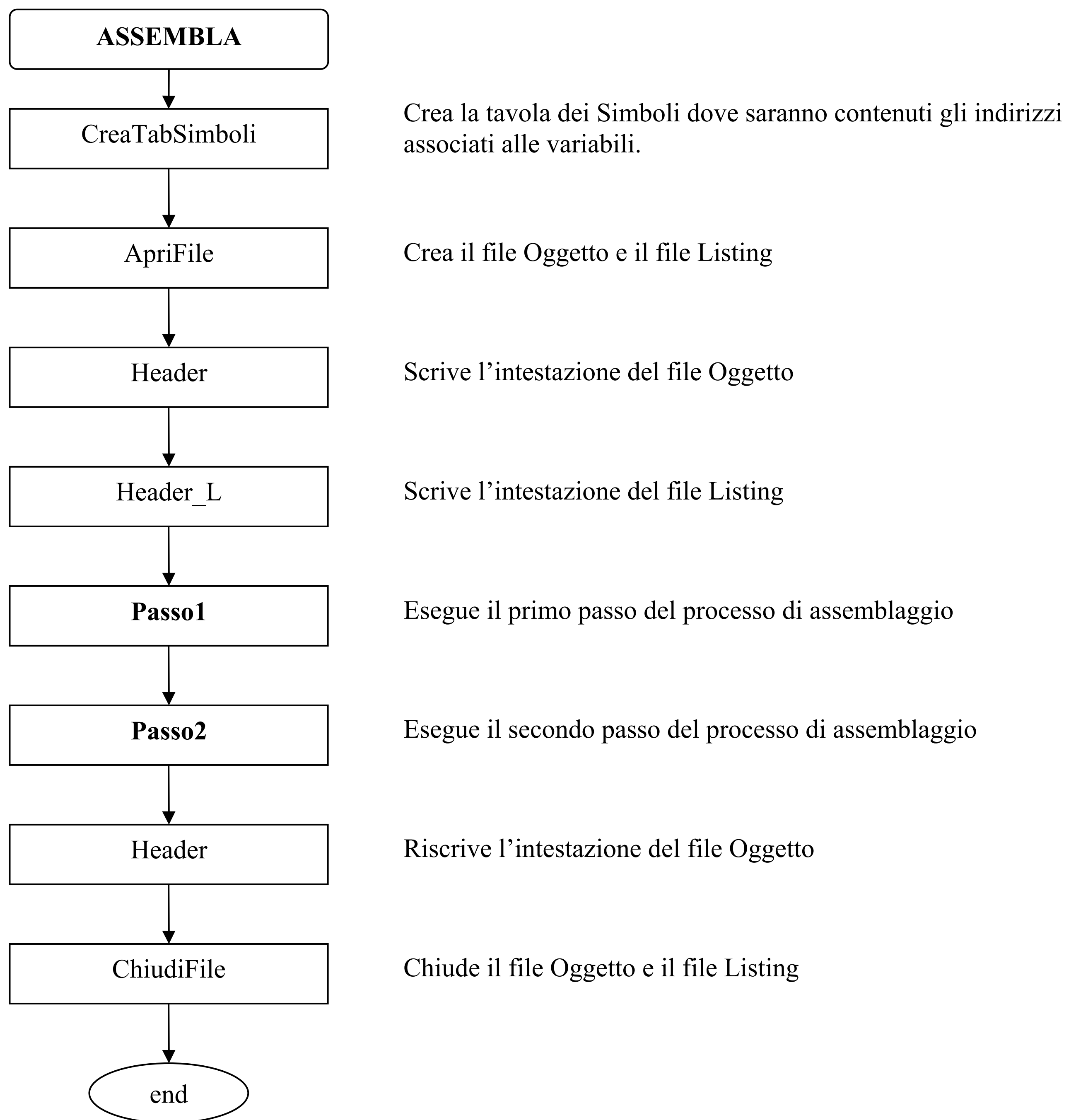


Figura 3: Flow Chart del processo di assemblaggio

1.2.2 Primo Passo

1.2.2.1 La tavola dei simboli

La tavola dei simboli è un array di variabili strutturate come in figura.

Viene riempita durante il primo passo in cui l'assemblatore vi memorizza le etichette che incontra durante la lettura del file sorgente.Ogni variabile è indicizzata mediante il valore restituito dalla funzione di hashing.

La generica variabile del vettore ha struttura:

	simbolo	valore	num_segmento	dopo
Elemento	AnsiString	int	int	*Elemento

Figura 4: Struttura degli elementi della Tavola dei Simboli

Vediamo il significato dei vari campi:

- **simbolo:** etichetta, nome variabile,..
- **valore:** indirizzo associato al simbolo.
- **num_segmento:** segmento di appartenenza del simbolo (il segmento viene identificato tramite un numero)
- **dopo:** punta all'elemento successivo in caso di collisione.

1.2.2.2 La tavola dei Segmenti

La tavola dei segmenti è un array di variabili strutturate con i seguenti campi:

- **Num**: indice che individua il segmento.
- **Tipo**: variabile che specifica il tipo di segmento
  ('d'=segmento dati, 'c'=segmento codice).
- **Dim**: individua le dimensione in byte del segmento.

1.2.2.3 La tavola di Rilocazione

La tavola di rilocazione è un array di variabili strutturate con i seguenti campi:

- **Seg_Iniz**: indice che individua il segmento dell'istruzione da rilocare.
- **Offset**: spiazzamento dell'istruzione da rilocare nel segmento.

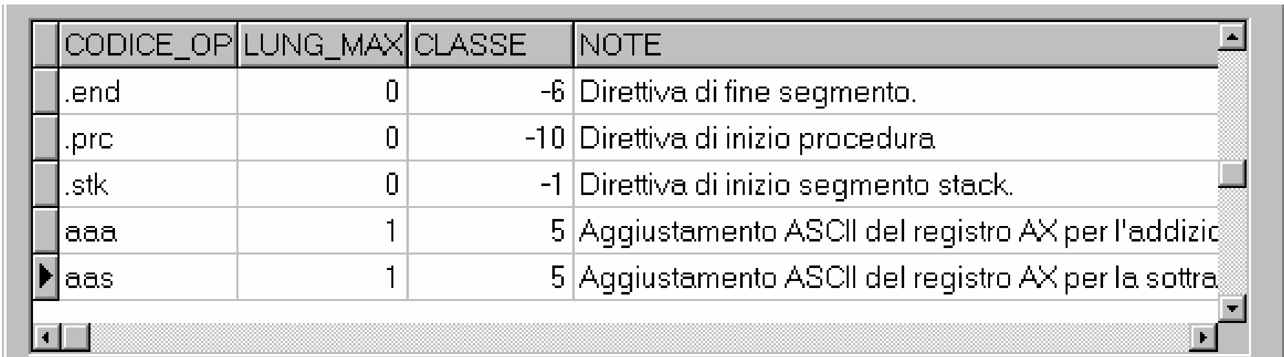
Le indicazioni contenute in questa tabella verranno memorizzate nel file oggetto e serviranno poi per la rilocazione nel momento dell'esecuzione del programma.

1.2.2.4 Tabella dei codici mnemonici

La tabella dei codici mnemonici è una tavola di database esterna al programma.

Questo permette di modificare in qualsiasi momento, senza dover cambiare il codice dell'assemblatore, i codici mnemonici delle istruzioni o delle direttive.

I campi della tabella, come si vede in figura, identificano il codice mnemonico, il codice operativo binario, il postbyte, la lunghezza massima dell'istruzione, la classe a cui appartiene, il tipo e una descrizione del codice mnemonico.



CODICE_OP	LUNG_MAX	CLASSE	NOTE
.end	0	-6	Direttiva di fine segmento.
.prc	0	-10	Direttiva di inizio procedura
.stk	0	-1	Direttiva di inizio segmento stack.
aaa	1	5	Aggiustamento ASCII del registro AX per l'addizic
aas	1	5	Aggiustamento ASCII del registro AX per la sottra

Figura 5: La tabella dei codici mnemonici

Cod. mnemonico	Cod. Operativo	Postbyte	Lung. max	Classe	Tipo	Note
AnsiString	AnsiString	AnsiString	Int	Int	Int	AnsiString

Figura 6: Struttura della tavola dei Codici Mnemonici

Il set di istruzioni implementato è indicato nella tabella seguente:

Cod. Mnemonico	Codice Operativo		Commento
ADC	00010100		Somma con carry Imm con AX
ADD	00000000	00000000	Somma Reg/Mem con Reg ad altro
ADD	10000000	00000000	Somma Imm a Reg/Mem
ADD	00000100		Somma Imm ad AX
AND	00100000	00000000	And Reg/Mem e Reg ad altro
AND	10000000	00100000	And Imm a Reg/Mem
AND	00100100		And Imm ad AX
CALL	11101000		Call dir. all'interno del segmento
CALL	10011010		Call dir. intersegmento
CBW	10011000		Converti AX da byte a parola
CLC	11111000		Azzera il bit di carry
CMP	00111000	00000000	Confronta Reg/Mem e Reg
CMP	10000000	00111000	Confronta Imm con Reg/Mem
CMP	00111100		Confronta Imm con AX
DEC	11111110	00001000	Decrementa Reg/Mem
DEC	01001000		Decrementa Reg$_{16}$
DIV	11110110	00110000	Divide Reg/Mem con Ax
HLT	11110100		Alt
IN	11100100		Input dalla porta specificata
INC	11111110	00000000	Incrementa Reg/Mem
INC	01000000		Incrementa Reg$_{16}$
JCXZ	11100011		Salta se CX=0
JE	01110100		Salta se uguale
JG	01111111		Salta se maggiore
JL	01111100		Salta se minore
JLE	01111110		Salta se minore o uguale
JMP	11101011		Salto incond. diretto nel segmento
JMP	11101010		Salto incond. diretto intersegmento
JO	01110000		Salta se overflow
JP	01111010		Salta se parità
JS	01111000		Salta al segno
LAHF	10011111		Carica AH con flags
LEA	10001101	00000000	Carica EA nel Reg
LOOP	11100010		Loop CX volte
LOPZ	11100001		Loop while zero
MOV	10001000	00000000	Mov Reg/Mem a/da Reg
MOV	11000110	00000000	Mov da imm a Reg/Mem
MOV	10110000		Mov da imm a Reg
MOV	10100000		Mov da Mem a AX
MOV	10100010		Mov da Ax a Mem
MOV	10001110	00000000	Mov da Reg/Mem a segm. Reg
MUL	11110110	00100000	Moltiplicazione
NOP	10010000		Nop
NOT	11110110	00010000	Not
OR	00001000	00000000	Or Reg/Mem e Reg ad altro
OR	10000000	00001000	Or imm e Reg/Mem

OR	0000110**0**		Or Imm ad Ax
OUT	1110011**0**		Output sulla porta specificata
POP	10001111	**00**000**000**	Pop Reg/Mem
POP	01011**000**		Pop Reg
POP	000**00**111		Pop segm. Reg
POPF	10011101		Pop flags
PUSH	11111111	**00**110**000**	Push Reg/Mem
PUSH	01010**000**		Push Reg
PUSH	00**000**110		Push segm. Reg
PUSF	10011100		Push flags
RET	11000011		Ritorno da call all'interno del segmento
RET	11001011		Ritorno da call intersegmento
ROL	110100**00**	**00**000**000**	Ruota a sinistra Reg/Mem
SAHF	10011110		Memorizza AH nel flag
SBB	0001110**0**		Sottrazione (con prestito) imm da AX
SHL	110100**00**	**00**100**000**	Shift logico a sinistra Reg/Mem
SUB	001010**00**	**00000000**	Sottrazione Reg/Mem e Reg ad altro
SUB	001010**00**	**00**101**000**	Sottrazione imm da Reg/Mem
SUB	0010110**0**		Sottrazione imm da AX
TEST	1000010**0**	**00000000**	And per flag no Reg/Mem e Reg
TEST	1010100**0**		And per flag no imm e AX
WAIT	10011011		Wait
XCHG	1000011**0**	**00000000**	Scambio Reg/Mem con Reg
XCHG	10010**000**		Scambio Reg/Mem con AX
XOR	001100**00**	**00000000**	Xor Reg/Mem e Reg ad altro
XOR	1000000**0**	**00**110**000**	Xor imm a Reg/Mem
XOR	0011010**0**		Xor imm ad AX

Figura 7: Tavola dei Codici Operativi

1.2.2.5 La fase di assemblaggio durante il primo passo

Il processo di assemblaggio durante il primo passo può essere descritto dal flow chart seguente:

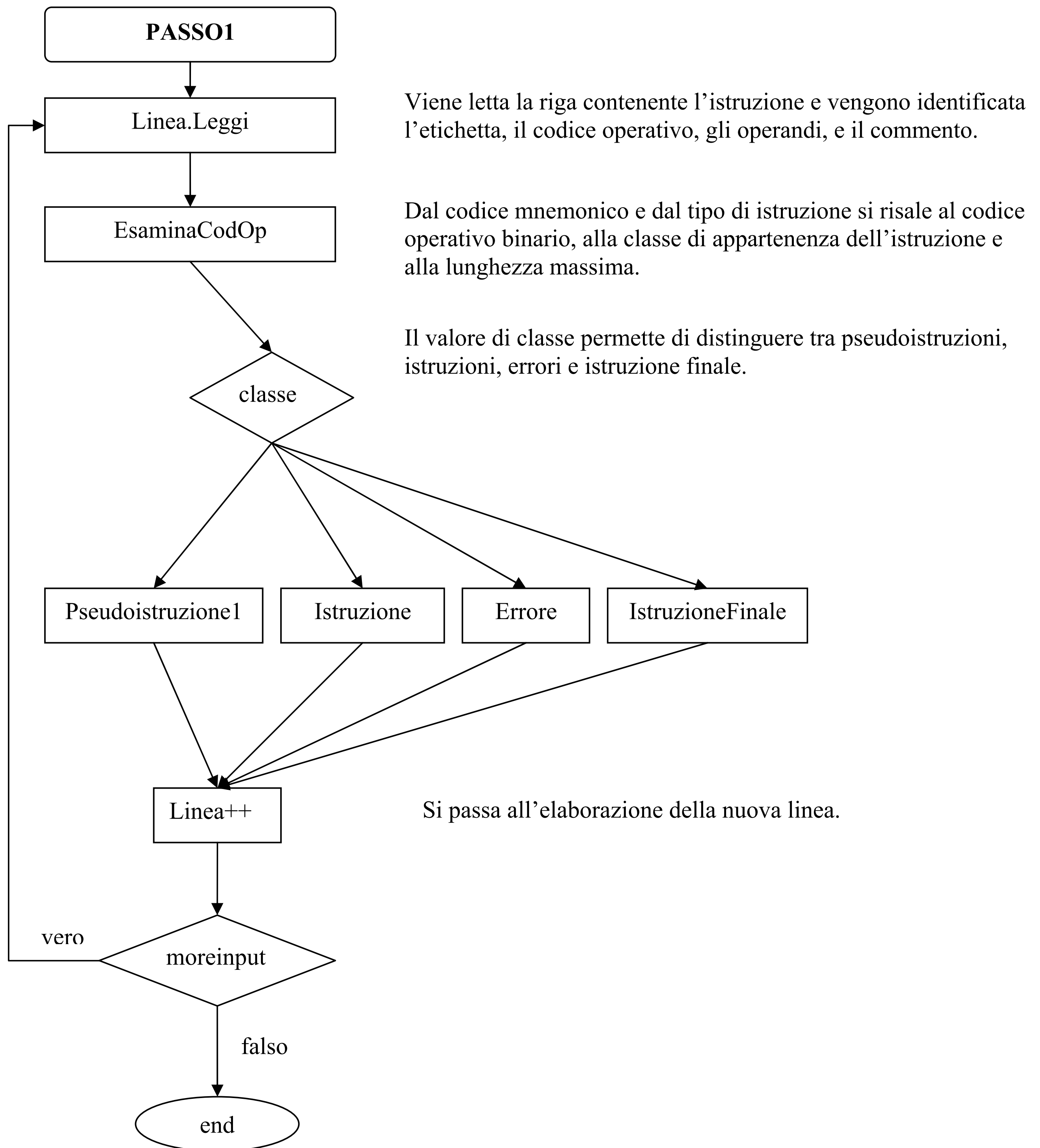


Figura 8: Flow Chart del Primo Passo

Il blocco “Pseudoistruzione1”, in base alla classe, determina una delle possibili pseudoistruzioni:

- .stk
- .dat
- .cod
- .db
- .dw
- .end
- end
- .bgn
- .prc

Le direttive “.db”, “.dw” agiscono sulla Tavola dei Simboli, mentre le restanti direttive salvano le informazioni riguardanti i segmenti stack, dati e codice nella tavola dei Segmenti.

All’interno di questo blocco viene inoltre incrementato il CDL (contatore di linea) usando la lunghezza massima dell’istruzione.

Il blocco “Istruzione” ha due funzioni:

- Se è presente un etichetta, questa viene inserita nella Tavola dei Simboli.
- Viene incrementato il CDL (contatore di linea) usando la lunghezza massima dell’istruzione.

1.2.3 Secondo Passo

Alla fine del primo passo tutte le etichette sono state inserite nella Tavola dei Simboli e ad ognuna di esse è stato associato l'offset che hanno all'interno del segmento in cui si trovano.

A questo punto è possibile assemblare le istruzioni.

Lo schema di assemblaggio del secondo passo è descritto nel flow-chart seguente (formalmente simile a quello del primo passo):

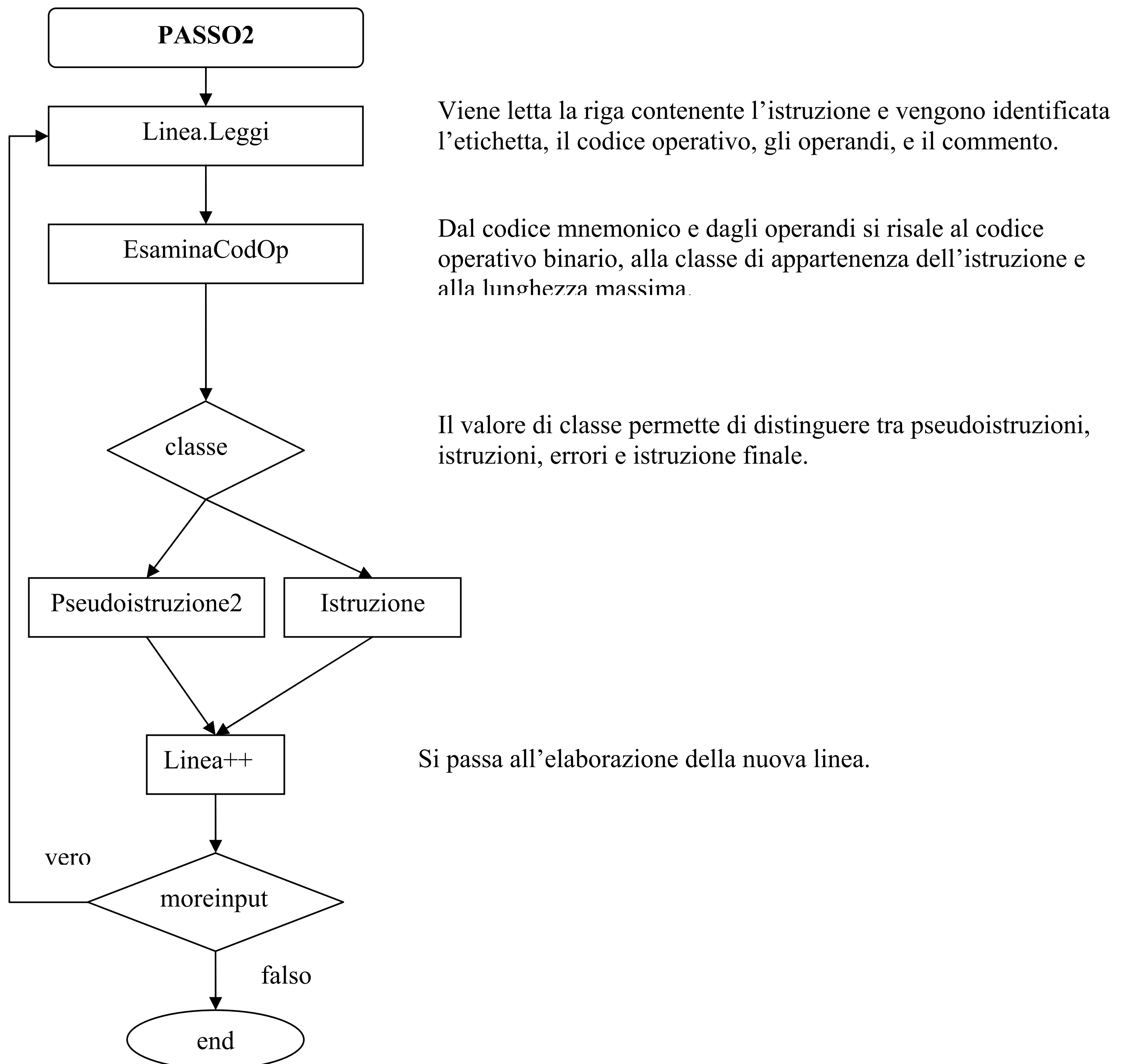


Figura 9:Flow Chart del Secondo Passo

Nel blocco “Pseudoistruzione2” vi sono le funzionalità necessarie affinché si possa scrivere sul file listing le informazioni relative alle pseudoistruzioni. All’interno di questo blocco viene inoltre incrementato il CDL (contatore di linea) usando, questa volta, la lunghezza reale dell’istruzione e colmando la differenza di byte con la lunghezza massima impostata nel primo passo con delle NOP.

Nel blocco “Istruzione” si eseguono le seguenti operazioni:

- Uno switch sulla variabile “classe” permette di avviare la funzione relativa alla classe di appartenenza dell’istruzione. La funzione restituisce la lunghezza reale dell’istruzione e determina tutti i dati necessari per assemblare l’istruzione. Controlla, inoltre, l’eventuale presenza di errori.
- Viene assemblata l’istruzione mediante la funzione **Oggetto().** L’istruzione assemblata viene scritta sul file oggetto.
- L’istuzione, con le relative informazioni, viene scritta sul file listing mediante la funzione **Listing().**
- Viene incrementato il CDL (contatore di linea) usando la lunghezza reale dell’istruzione.
- Vengono inserite delle NOP per colmare la differenza di byte tra la lunghezza massima dell’istruzione impostata nel primo passo e la lunghezza reale dell’istruzione determinata nel secondo passo.

1.2.3.1 Traduzione delle istruzioni

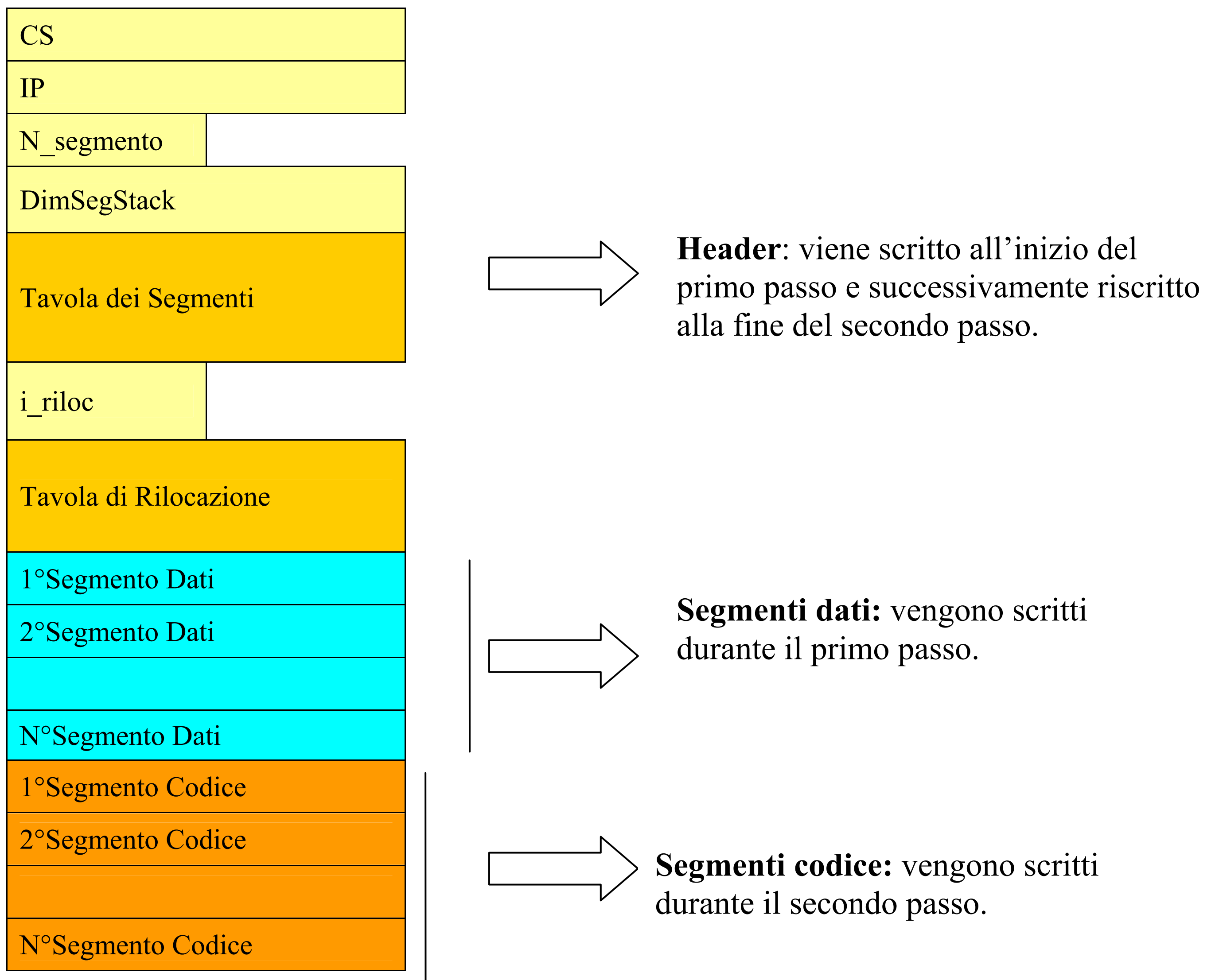
La traduzione delle istruzioni è attuata grazie alla chiamata di una delle funzioni **Funz*N*.** Le istruzioni del processore 8086 sono infatti raggruppabili in categorie, ognuna delle quali viene tradotta da una funzione specifica. La funzione restituisce la lunghezza reale dell’istruzione e determina tutti i dati necessari per assemblare l’istruzione. Per la determinazione dei vari campi dell’istruzione l’assemblatore utilizza le procedure e le funzioni:

- void Codice_w();
- void Codice_s();
- void Codice_v();
- void Codice5_w();
- void Codice_d();
- void Codice_reg();
- void Codice_sr();
- int Campi_mod_rm_spiazz(AnsiString Opx);
- int Campi_mod_rmreg(AnsiString Opx);
- void Campi_reg_w(AnsiString Opx);

- void Campo_sr(AnsiString Opx);
- void Campi_imm(AnsiString Opx);

1.2.4 File Oggetto

Il file oggetto è stato organizzato come in figura:



2 Visual Simulator 1.1

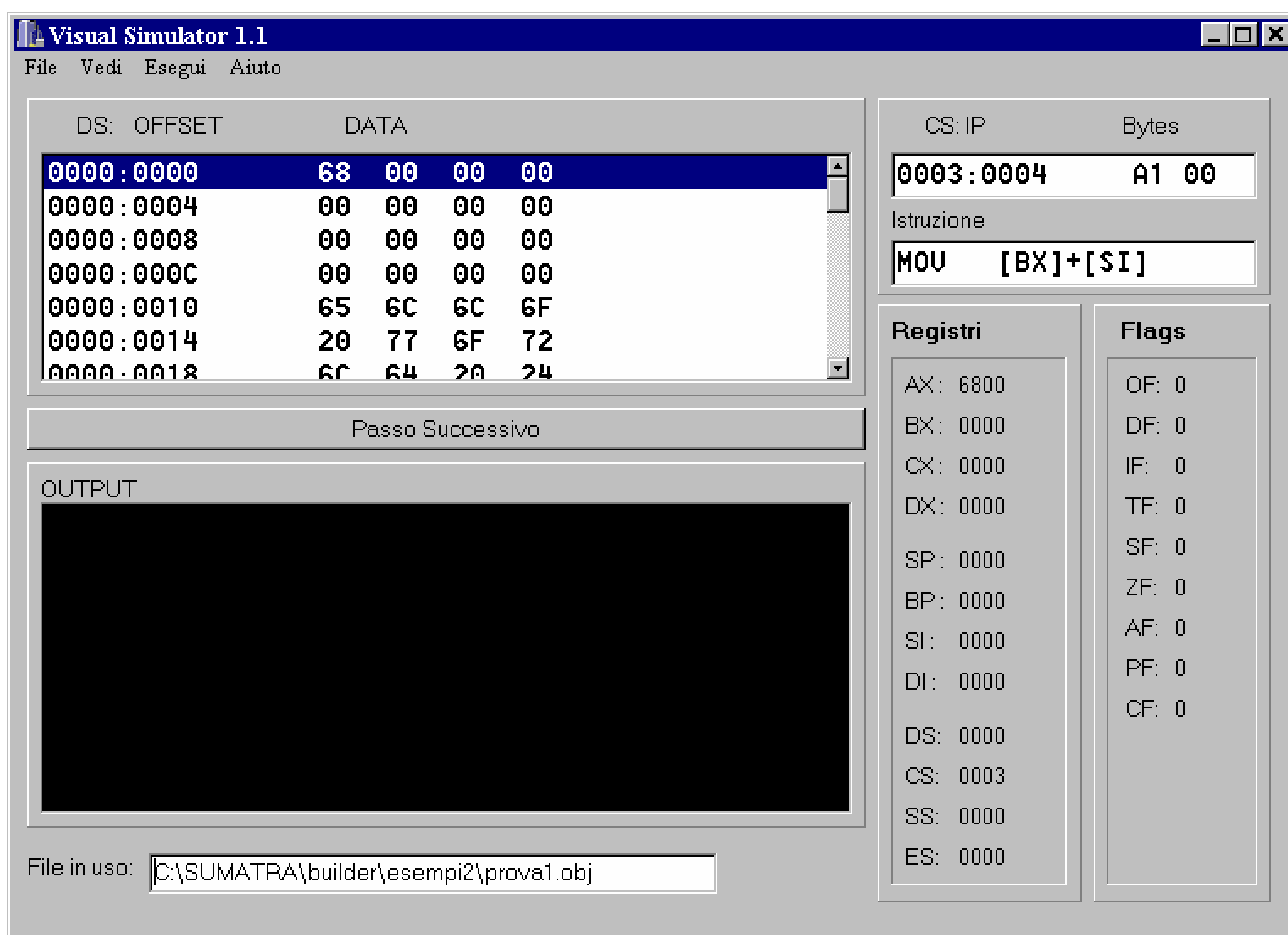


Figura 10: Una schermata del simulatore

2.1 Ciclo Fetch-Decode-Execute

Una qualsiasi istruzione che viene eseguita dall'8086 deve seguire tre fasi sempre nello stesso ordine:

- Fase di Fetch: in questa fase viene prelevata l'istruzione dalla memoria (dal segmento codice puntato da CS)
- Fase di Decode: in questa fase il controllore dell'8086 determina il tipo di operazione, gli eventuali registri impiegati dall'istruzione, e l'eventuale indirizzo di memoria implicato nell'esecuzione.
- Fase di Execute: in quest'ultima fase l'istruzione decodificata viene eseguita e viene inoltre incrementato l'IP.

2.1.1 La fase di Fetch

La fase di prelievo dell'istruzione consiste nel determinare l'indirizzo assoluto dell'istruzione da eseguire in base ai dati contenuti in IP e CS e nel caricare il CIR con il contenuto della cella di memoria puntata da EA.

La fase di fetch dell'istruzione, in RTL, si può riassumere in:

$CIR \leftarrow_{16} MEM[IP+CS<<4]$

2.1.2 La fase di Decode

La fase di decode consiste nel determinare tutte le informazioni necessarie per la completa decodifica dell'istruzione. Gli 8 bit meno significativi contenuti nel CIR contengono la maggior parte delle informazioni necessarie alla decodifica.

Le istruzioni dell'8086 si possono suddividere in due grandi categorie:

Istruzioni la cui decodifica è completamente determinata dagli 8 bit meno significativi del CIR.

Istruzioni la cui decodifica è determinata anche dalla parte alta del CIR (campo postbyte).

Per determinare la categoria dell'istruzione basta decodificare il primo byte del CIR.

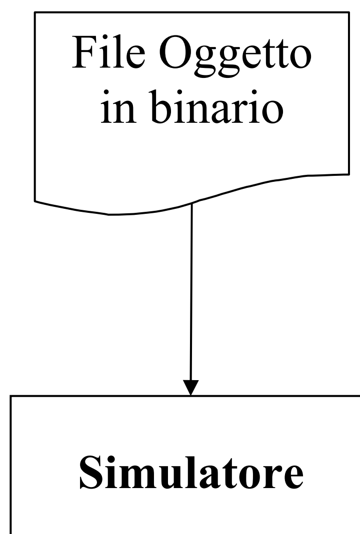
2.1.3 La fase di Execute

Nella fase di execute, infine, la CPU può operare sui dati e produrre risultati in funzione del codice operativo contenuto nel CIR. In questa fase viene utilizzata l'unità aritmetico-logica (ALU) e vengono settati i bit del registro dei flags a seconda dell'esito dell'istruzione.

2.2Il software di Simulazione

2.2.1 Schema Generale

Lo scopo del simulatore è eseguire il codice binario di un file OGGETTO realizzato tramite l'assemblatore Visual Assembler 1.3.



2.2.2 Caratteristiche Principali

Il simulatore è stato realizzato **per ambiente Windows** tramite il linguaggio:

Borland C++ Builder 3 v1.0

Il simulatore può funzionare secondo due differenti modalità:

- **Esecuzione:** Il simulatore esegue una dopo l'altra le istruzioni presenti nel codice oggetto caricato.
- **Debug:** Il simulatore esegue un' istruzione e aspetta la pressione di un tasto per l'esecuzione di un'istruzione successiva. In questa modalità il simulatore visualizza il codice operativo in esecuzione e il contenuto della memoria, dei registri e dei flags.

Esaminiamo le caratteristiche del simulatore descrivendo le funzionalità presenti nel menù a tendine:

Menù FILE

- **Apri:** Permette l'apertura di un file oggetto (in formato Visual Assembler 1.3)
- **Exit:** Permette l'uscita dall'assemblatore.

Menù VEDI

- **Registri:** Permette di visualizzare/non visualizzare il contenuto dei registri.
- **Memoria:** Permette di visualizzare/non visualizzare il contenuto della memoria.
- **Flags:** Permette di visualizzare/non visualizzare il contenuto dei flags.

Menù ESEGUI

- **Esegui:** Avvia il simulatore in modalità **Esecuzione.**
- **PassoPasso:** Avvia il simulatore in modalità **Debug.**

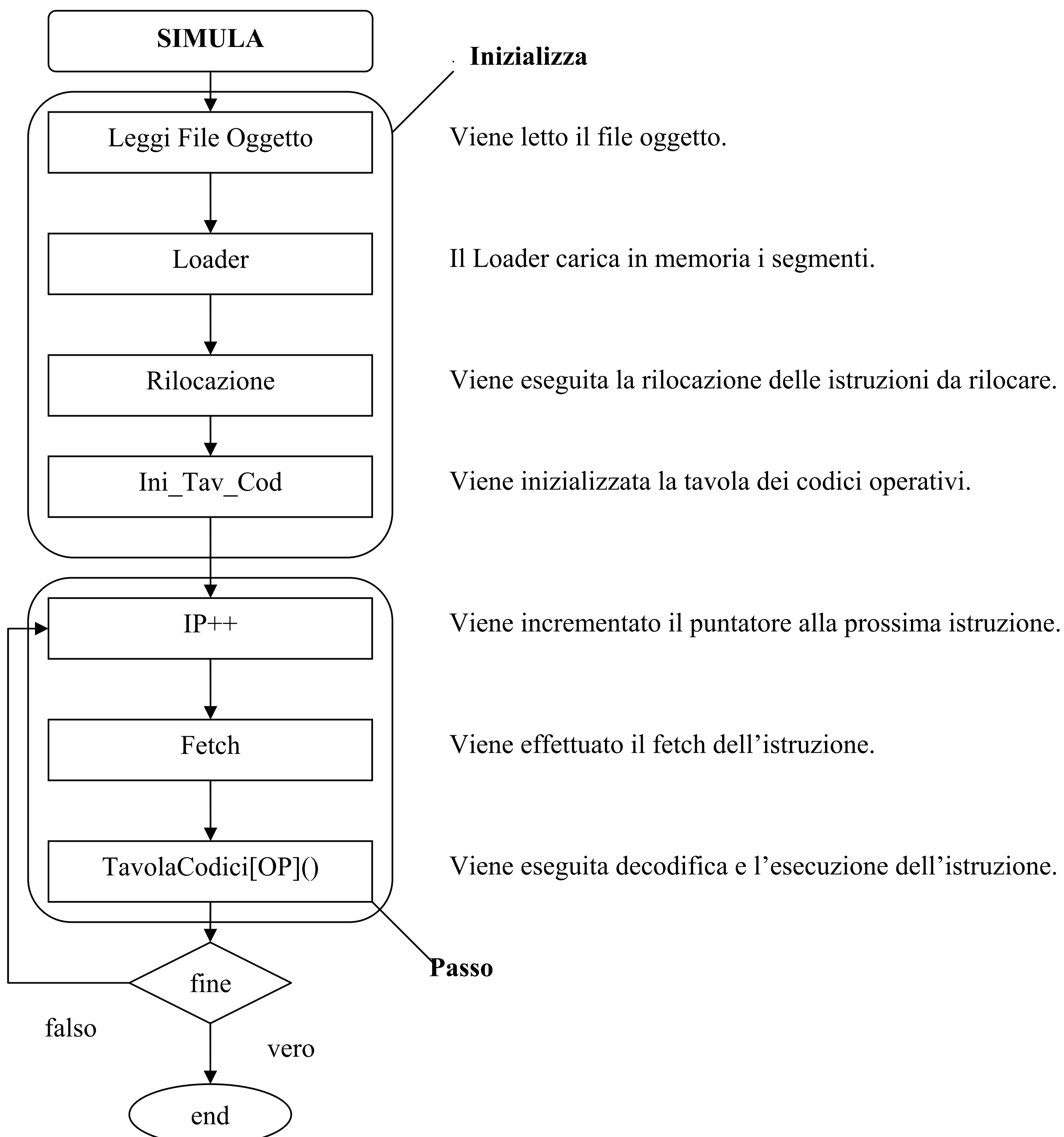
Menù AIUTO

- **Aiuto:** Fornisce informazioni sull'utilizzo del programma.
- **Informazioni:** Fornisce informazioni sul programma.

2.2.3 La Simulazione

Il simulatore realizzato è di tipo interpretativo: infatti il programma di simulazione legge il codice istruzione per istruzione e per ognuna di esse vengono eseguite, nell'ambiente virtuale, esattamente le stesse operazioni che la CPU originale avrebbe eseguito. Questo sistema ha il vantaggio di consentire in maniera molto semplice un controllo fine sui risultati prodotti, ma è anche piuttosto lento. Questo però, in questo caso, non è un problema data la bassa velocità di esecuzione della CPU 8086 da simulare. Il simulatore opera a livello di macchina base, ma sono state incluse delle funzionalità del Sistema Operativo come ad esempio il LOADER (per il caricamento in memoria del programma oggetto).

Da un punto di vista algoritmico, il simulatore si comporta come descritto nel flow chart seguente:



2.2.3.1 Descrizione del software di simulazione

Nella lettura del file Oggetto vengono utilizzate le seguenti strutture:

La tavola dei segmenti:

E' un array di variabili strutturate con i seguenti campi:

- **Num**: indice che individua il segmento.
- **Tipo**: variabile che specifica il tipo di segmento ('d'=segmento dati, 'c'=segmento codice).
- **Dim**: individua le dimensione in byte del segmento.
- **Inizio**: Variabile che specifica il paragrafo di inizio del segmento.Questo campo verrà utilizzato dal loader per il caricamento in memoria dei segmenti.

La tavola di rilocazione:

E' un array di variabili strutturate con i seguenti campi:

- **Seg_Iniz**: indice che individua il segmento dell'istruzione da rilocare.
- **Offset**: spiazzamento dell'istruzione da rilocare nel segmento.

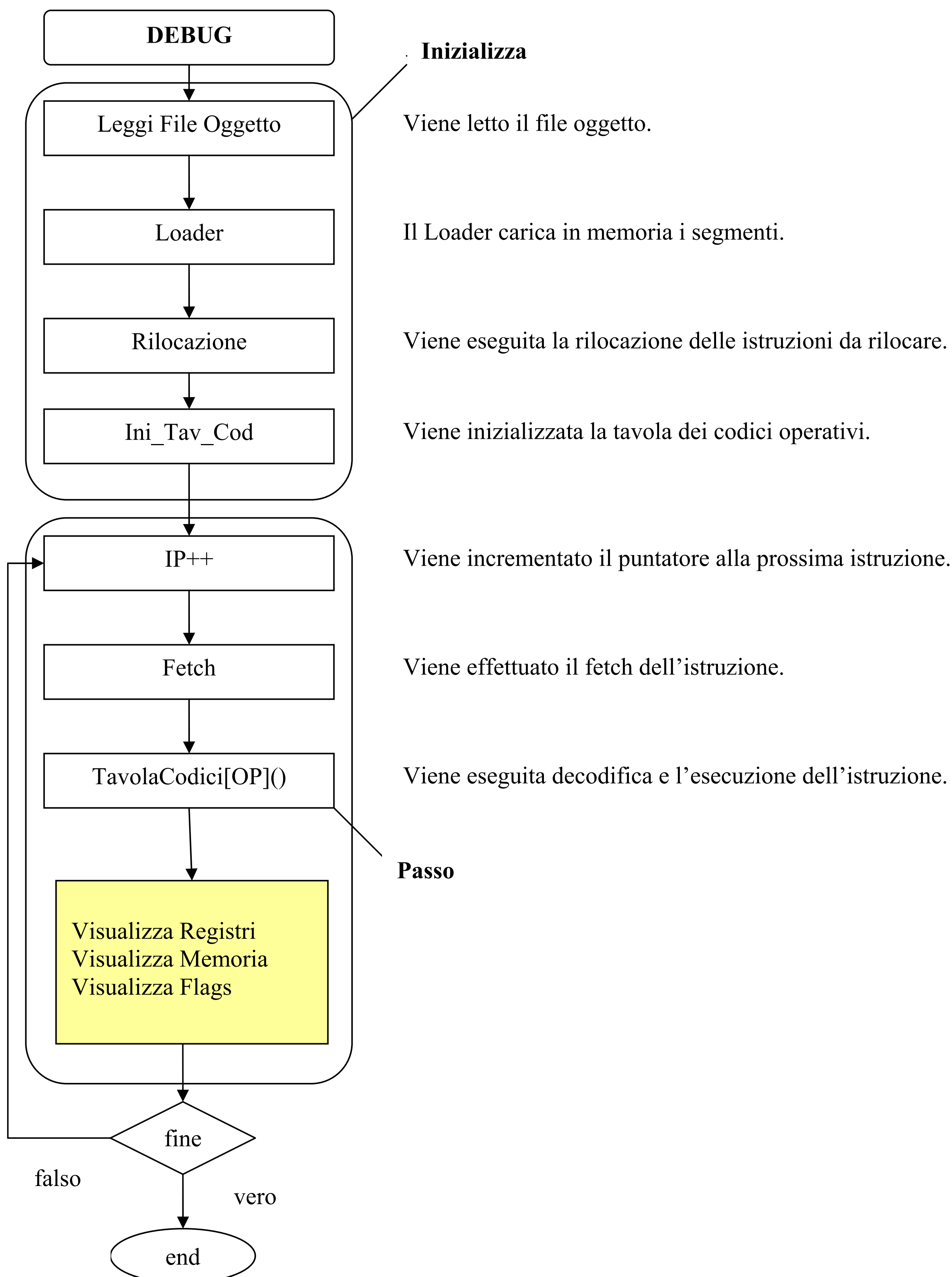
Per l'esecuzione delle istruzioni sono state definite:

- Le variabili char per la simulazione dei registri.
- Un array di 1MB per la simulazione della memoria.
- Un array di puntatori a funzione per i codici operativi implementati.

2.2.4 Il Debugger

Lo schema del Debugger è molto simile a quello del simulatore. In effetti, il debugger esegue le stesse operazioni del simulatore e in più mostra l'istruzione corrente e il contenuto della memoria, dei registri e dei flags.

Il flow-chat del Debugger è:



3 Il processore 8086

3.1 Caratteristiche del processore

Elenchiamo le caratteristiche principali del processore:

- Usa la tecnologia HMOS.
- Il chip è composto da circa 29,000 transistor ed ha 40 pin.
- A seconda dei modelli può lavorare ad una frequenza di 5 MHz (8086), 8 MHz (8086-2) oppure 10 Mhz (8086-1).
- Richiede un'unica tensione di alimentazione (+5V).
- Spazio di indirizzamento pari a 2 20 =1Mbyte.
- 16 tra i 20 pin di indirizzo fungono anche da pin di dato.
- È orientato alla multiprogrammazione ed al multiprocessing.

Modello Architetturale

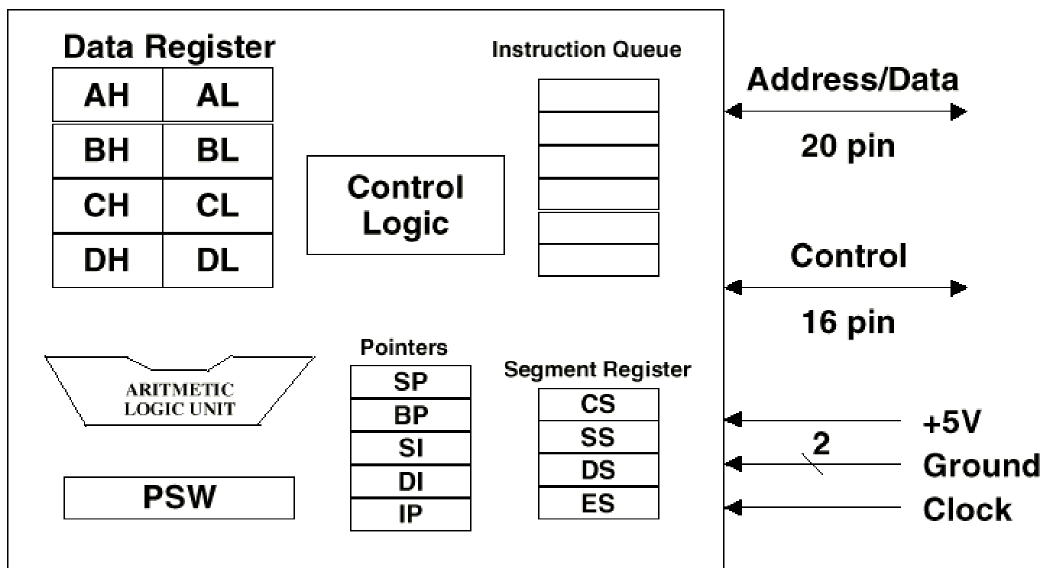


Figura 11: Il Modello Architetturale dell'8086

3.2Registri

Possono essere suddivisi in 3 gruppi:

- registri di dato
- registri puntatore
- registri di segmento.

3.2.1 Registri di Dato

I 4 registri di dato sono:

- AX (Accumulator Register),
- BX (Base Register)
- CX (Count Register)
- DX (Data Register).

Sono utilizzati per memorizzare operandi e risultato delle operazioni. Possono essere utilizzati come registri da 16 bit oppure come coppie di registri da 8 bit. BX può anche essere utilizzato nel calcolo di indirizzi. CX viene anche utilizzato come contatore da talune istruzioni. DX contiene l'indirizzo di I/O in alcune istruzioni di I/O.

3.2.2 Registri Puntatore

I 5 registri puntatore sono:

- IP
- SP
- BP
- SI
- DI

IP (*Instruction Pointer)* contiene il puntatore alla prima istruzione da eseguire. IP non può comparire esplicitamente come operando di una istruzione.

SP (*stack pointer*) contiene il puntatore alla testa dello stack.

BP (*Base Pointer*) viene utilizzato come base per fare accesso all'interno dello stack.

SI (*Source Index*) e DI (*Destination Index*) vengono utilizzati come registri indice.

3.2.3 Registri di Segmento

I 4 registri di segmento sono:

- CS
- DS
- ES
- SS

Vengono utilizzati per costruire gli indirizzi fisici con i quali fare accesso in memoria. Contengono i puntatori all'inizio dei segmenti di codice, di dato, di dato supplementare e di stack, rispettivamente.

3.3 Calcolo degli indirizzi

Ogni volta che l'8086 deve generare un indirizzo da mettere sull'A-bus (physical address), esso esegue una operazione di somma tra il contenuto di un registro puntatore oppure di BX (effective address o offset) ed il contenuto di un registro di segmento (segment address). La somma avviene dopo aver moltiplicato per 16 (shift di 4 posizioni) il contenuto del registro di segmento:

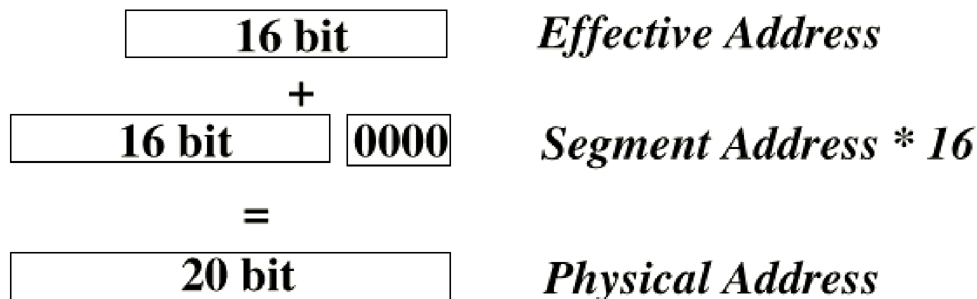


Figura 12: Calcolo degli indirizzi

3.4 Segmenti

La memoria può essere considerata come organizzata in segmenti, ognuno di dimensione pari a 64 Kbyte. Tutti i segmenti cominciano ad indirizzi multipli di 16.

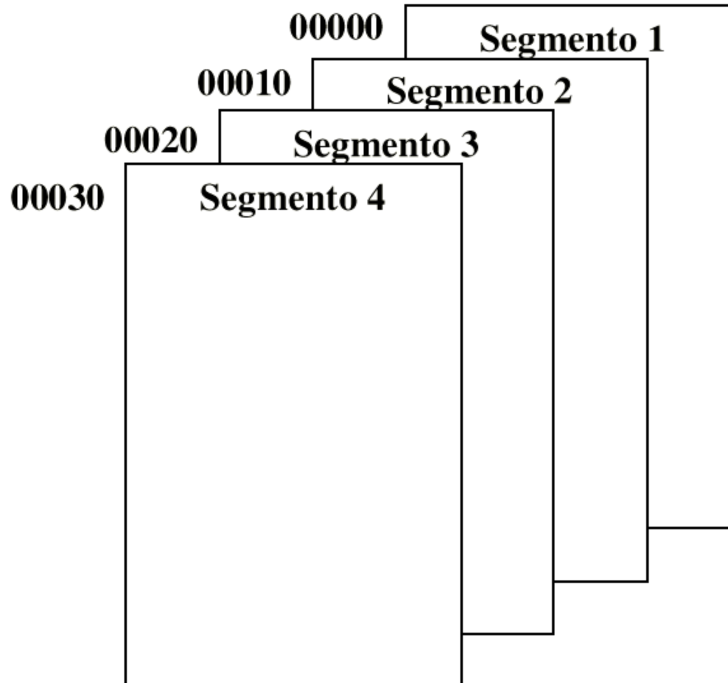


Figura 13: I segmenti

3.4.1 Uso dei Segmenti

Una volta caricato l'indirizzo di testa del segmento in un segment register, tutti gli indirizzi all'interno del segmento sono esprimibili attraverso un registro contenente l'offset.

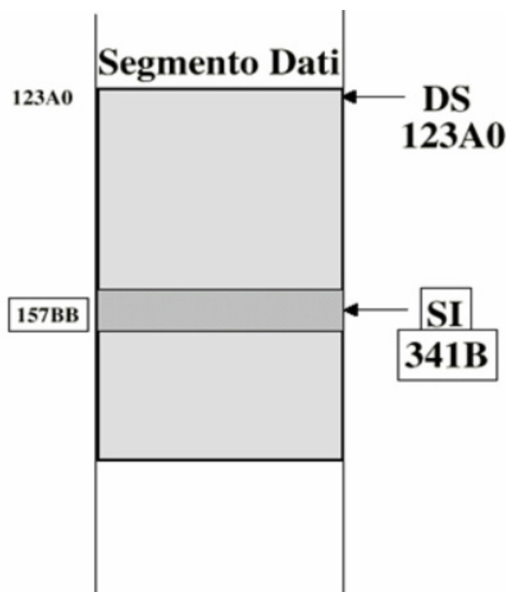


Figura 14:Uso dei Segmenti

3.4.2 Paragrafi

I gruppi di 16 byte che iniziano ad indirizzi multipli di 16 si definiscono paragrafi. La memoria è quindi organizzata in paragrafi.

Paragrafo 0		00000h
Paragrafo 1		00010h
Paragrafo 2		00020h
Paragrafo 3		00030h
Paragrafo 4		00040h

Figura 15: I paragrafi

3.4.3 Vantaggi della Segmentazione

La segmentazione permette di ottenere i seguenti vantaggi:

- Spazio di indirizzamento pari a 2^{20}, ma indirizzi a 16 bit.
- Separazione tra dati, codice e stack.
- Possibilità di avere più segmenti dello stesso tipo (dati,codice o stack).
- Possibilità di sovrapposizione tra segmenti, con minimizzazione della memoria sprecata.
- Rilocabilità.

3.4.4 Segmentazione della Memoria

Una volta caricato l'indirizzo di testa del segmento in un segment register, tutti gli indirizzi all'interno del segmento sono esprimibili attraverso un registro contenente l'offset.

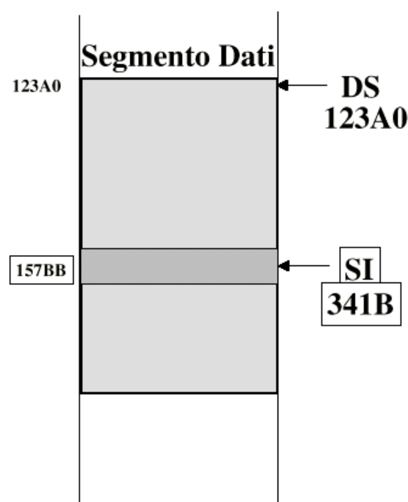


Figura 16: Segmentazione della Memoria

3.5Process Status Word (PSW)

È composta da 16 bit, ma solo 9 di questi sono usati. Ogni bit corrisponde ad un flag.

I flag si dividono in:

- flag di condizione
- flag di controllo.

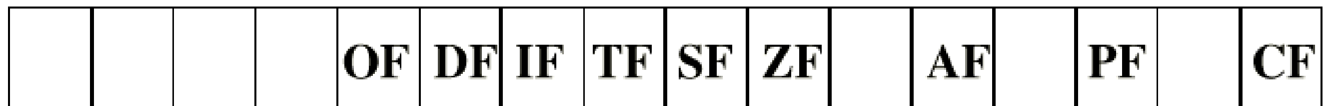


Figura 17: La Process Status Word

3.5.1 Flag di Condizione

Vengono automaticamente scritti al termine di varie operazioni:

- SF (Sign Flag): coincide con il MSB del risultato dopo una operazione aritmetica
- ZF (Zero Flag): vale 0 se il risultato è nullo, 1 altrimenti
- PF (Parity Flag): vale 1 se il numero di 1 negli 8 bit meno significativi del risultato è pari, 0 altrimenti
- CF (Carry Flag): dopo le operazioni aritmetiche vale 1 se c'è stato riporto (somma) o prestito (sottrazione); altre istruzioni ne fanno un uso particolare
- AF (Auxiliary Carry Flag): usato nell'aritmetica BCD; vale 1 se c'è stato riporto (somma) o prestito (sottrazione) dal bit 3
- OF (Overflow Flag): vale 1 se l'ultima istruzione ha prodotto un overflow.

3.5.2 Flag di Controllo

Possono venire scritti e manipolati da apposite istruzioni, e servono a regolare il funzionamento di talune funzioni del processore:

- DF (Direction Flag): utilizzato dalle istruzioni per la manipolazione delle stringhe; se vale 0 le stringhe vengono manipolate partendo dai caratteri all'indirizzo minore, se vale 1 a partire dall'indirizzo maggiore
- IF (Interrupt Flag): se vale 1, i segnali di Interrupt mascherabili vengono percepiti dalla CPU, altrimenti questi vengono ignorati
- TF (Trap Flag): se vale 1, viene eseguita una trap al termine di ogni istruzione.

3.6 Execution Unit (EU)

Provvede alla decodifica ed alla esecuzione delle istruzioni. Riceve byte per byte le istruzioni dalla BIU, le decodifica, genera gli indirizzi degli operandi (se necessario), li passa alla BIU; una volta ricevuti tutti gli operandi esegue l'istruzione, testa ed aggiorna i flag.

3.7 Bus Interface Unit (BIU)

Gestisce tutte le operazioni da e per l'esterno:

- Fetch delle istruzioni
- Lettura e scrittura operandi e risultati di istruzioni
- Generazione degli indirizzi
- Accodamento delle istruzioni

La BIU lavora in parallelo con la EU.

3.8 Coda delle Istruzioni

La BIU gestisce una struttura FIFO di dimensioni pari a 6 byte in cui vengono accumulate le istruzioni che si prevede dovranno essere eseguite. La Coda delle Istruzioni viene caricata dalla BIU ogni qual volta vi è una word libera, ed il bus è libero; in tal caso viene letta dalla memoria la word successiva nel Code Segment. Se viene eseguita una istruzione di salto, la Coda delle Istruzioni viene azzerata, e la EU deve attendere il tempo necessario per il fetch di una nuova istruzione prima di poter lavorare.

3.9 Accesso alla Memoria

L'8086 ha uno spazio di indirizzamento pari a 1 Mbyte. L'8086 è in grado di accedere in un solo passo ad un byte oppure ad una word di memoria, purchè questa inizi ad un indirizzo pari. L'accesso a word allineate su indirizzi dispari richiede due cicli di memoria. Fisicamente la memoria è organizzata come due banchi da 512Kbyte:

- il banco dispari (D_{15}-D_8)
- il banco pari (D_7-D_0)

I due banchi sono indirizzati in parallelo dalle linee di indirizzo A_{19}-A_1.

I dati con indirizzo pari sono trasferiti sui pin D 7 -D 0 mentre i dati ad indirizzi dispari sono trasferiti sui pin D 15 -D 8.

Il processore fornisce due segnali di enable (BHE e A_0) per gestire l'accesso alla memoria.

BHE	A_0	*Operazioni in memoria*
0	**0**	**Word Intera**
0	**1**	**Byte alto (dall'indirizzo dispari)**
1	**0**	**Byte basso (dall'indirizzo pari)**
1	**1**	**Nessuna**

Figura 18: Accesso alla Memoria

Rappresentazione delle word

Si presuppone che i dati memorizzati in una word abbiano il byte meno significativo memorizzato nel byte con indirizzo minore.

Nell'esempio si vede come sia memorizzato il numero FF00.

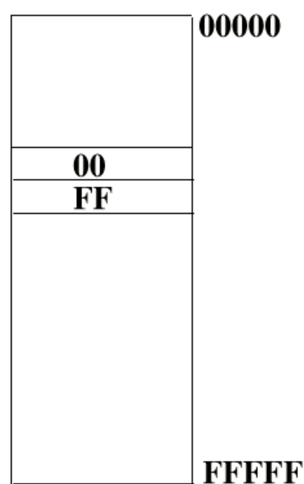


Figura 19: Rappresentazione delle word

Parti Riservate della Memoria

Alcune parti della memoria non sono libere, ma *dedicate*, oppure *riservate.*

Ad esempio la memoria dall'indirizzo 00000H allo 003FFH è riservata, in quanto contiene la *Interrupt Vector Table*.

Gli indirizzi da FFFF0H a FFFFBH sono utilizzati per contenere un'istruzione di salto alla routine di caricamento di programma di *bootstrap*.

Le locazioni da FFFFCH a FFFFFH sono invece riservate per usi futuri.

Area	Indirizzo
Interrupt Vector Table	00000 – 003FF
Liberi	
Reset Bootstrap	FFFF0
Riservati	FFFFC

Figura 20: Parti riservate della Memoria

3.10 Lo Stack

L'8086/8088 prevede alcune strutture e meccanismi hadware per la gestione di uno stack.

Lo stack corrisponde al segmento di memoria la cui testa è puntata da SS. Il top dello stack (locazione riempita per ultima) è puntato da SP. Lo stack cresce dalle locazioni di memoria con indirizzo maggiore verso quelle ad indirizzo minore.

Ogni operazione di PUSH decrementa di 2 unità SP e scrive una word nella locazione da questo puntata. Ogni operazione di POP estrae una word dalla locazione puntata da SP, e successivamente incrementa SP di 2 unità.

3.10.1 Esempio di Stack

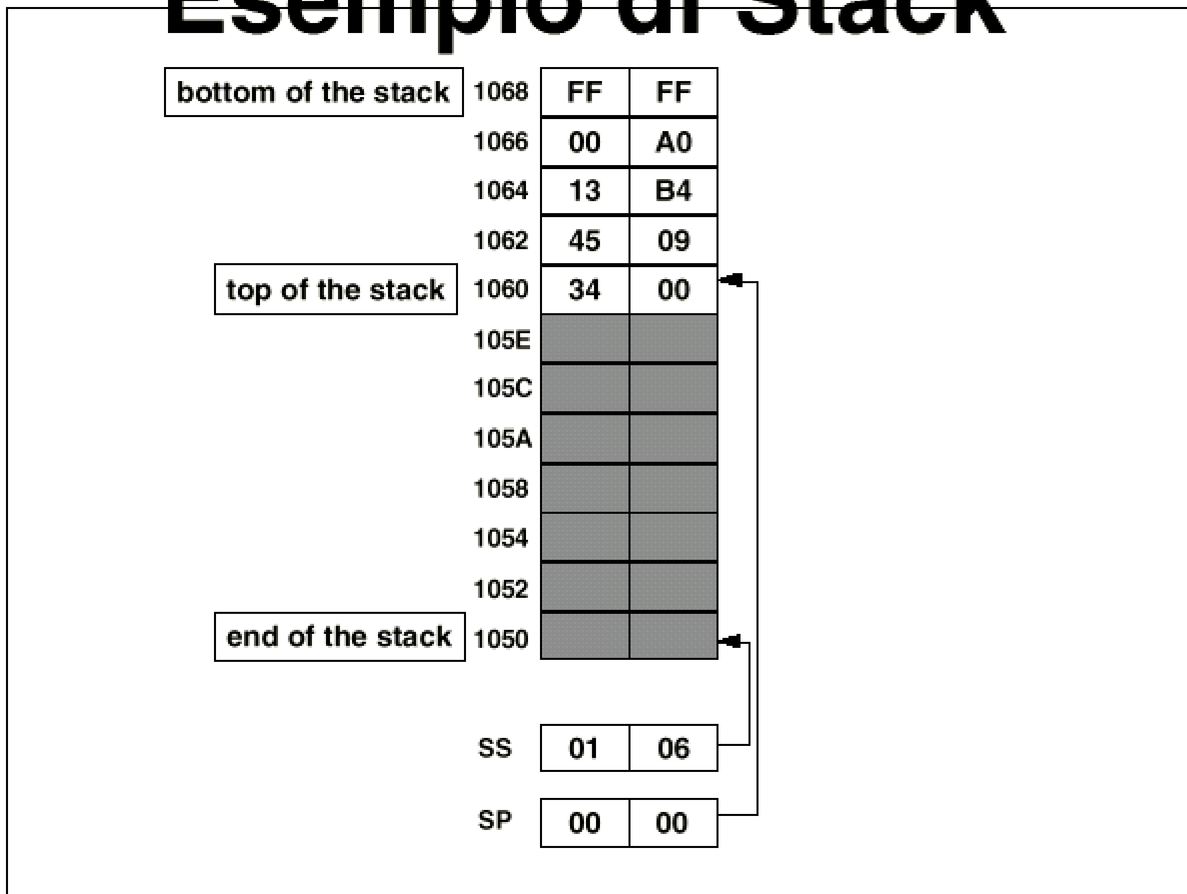


Figura 21: Esempio di Stack

Operazioni sullo stack

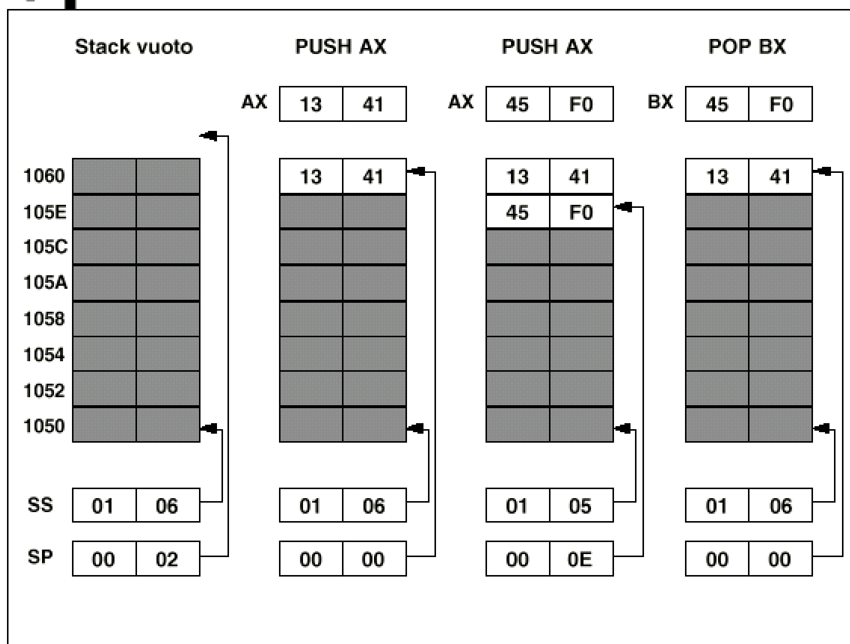


Figura 22: Operazioni sullo Stack

3.10.2 Uso dello Stack

Oltre che a seguito delle istruzioni che esplicitamente lo manipolano (PUSH e POP), lo stack viene modificato in altri due casi:

- all’atto della chiamata (CALL) o del ritorno (RET) da una procedura: nel primo caso viene automaticamente eseguita la PUSH dell’IP, nel secondo caso l’analoga POP

all’atto dell’attivazione di un interrupt, e del ritorno dalla corrispondente routine di servizio (IRET); nel primo caso vengono eseguite automaticamente le PUSH del program counter (registri CS e IP) e della PSW, nel secondo caso vengono eseguite le corrispondenti POP.

3.11 Input/Output

L’accesso alle periferiche avviene spesso attraverso speciali locazioni associate ad un certo indirizzo. L’accesso a tali locazioni può avvenire nell’8086 sia in modo memory mapped sia in isolated I/O. Nel primo caso l’accesso alla periferica avviene attraverso una normale istruzione, nel secondo attraverso speciali istruzioni di I/O. Lo spazio di indirizzamento dell’I/O è pari al più a 64K.

4 Sommario

1 Visual Assembler 1.3 ... 2
1.1 Generalità sull'assemblatore ... 3
1.1.1 Caratteristiche principali ... 3
1.1.2 Regole lessicali ... 4
1.2 Assemblaggio ... 6
1.2.1 Schema Generale ... 6
1.2.2 Primo Passo ... 8
1.2.3 Secondo Passo ... 14
1.2.4 File Oggetto ... 16
2 Visual Simulator 1.1 ... 17
2.1 Ciclo Fetch-Decode-Execute ... 18
2.1.1 La fase di Fetch ... 18
2.1.2 La fase di Decode ... 18
2.1.3 La fase di Execute ... 18
2.2 Il software di Simulazione ... 19
2.2.1 Schema Generale ... 19
2.2.2 Caratteristiche Principali ... 19
2.2.3 La Simulazione ... 21
2.2.4 Il Debugger ... 23
3 Il processore 8086 ... 24
3.1 Caratteristiche del processore ... 24
3.2 Registri ... 25
3.2.1 Registri di Dato ... 25
3.2.2 Registri Puntatore ... 25
3.2.3 Registri di Segmento ... 26
3.3 Calcolo degli indirizzi ... 26
3.4 Segmenti ... 27
3.4.1 Uso dei Segmenti ... 27
3.4.2 Paragrafi ... 28
3.4.3 Vantaggi della Segmentazione ... 28
3.4.4 Segmentazione della Memoria ... 28
3.5 Process Status Word (PSW) ... 29
3.5.1 Flag di Condizione ... 29
3.5.2 Flag di Controllo ... 29
3.6 Execution Unit (EU) ... 30
3.7 Bus Interface Unit (BIU) ... 30
3.8 Coda delle Istruzioni ... 30
3.9 Accesso alla Memoria ... 30
3.10 Lo Stack ... 32
3.10.1 Esempio di Stack ... 33
3.10.2 Uso dello Stack ... 34
3.11 Input/Output ... 34
4 Sommario ... 35
5 Indice delle Figure ... 36
6 Bibliografia ... 37

5 Indice delle Figure

Figura 1: Una schermata dell'assemblatore 2
Figura 2: Input/Output dell'Assemblatore 6
Figura 3: Flow Chart del processo di assemblaggio 7
Figura 4: Struttura degli elementi della Tavola dei Simboli 8
Figura 5: La tabella dei codici mnemonici 9
Figura 6: Struttura della tavola dei Codici Mnemonici 9
Figura 7: Tavola dei Codici Operativi 11
Figura 8: Flow Chart del Primo Passo 12
Figura 9:Flow Chart del Secondo Passo 14
Figura 10: Una schermata del simulatore 17
Figura 11: Il Modello Architetturale dell'8086 24
Figura 12: Calcolo degli indirizzi 26
Figura 13: I segmenti 27
Figura 14:Uso dei Segmenti 27
Figura 15: I paragrafi 28
Figura 16: Segmentazione della Memoria 28
Figura 17: La Process Status Word 29
Figura 18: Accesso alla Memoria 31
Figura 19: Rappresentazione delle word 31
Figura 20: Parti riservate della Memoria 32
Figura 21: Esempio di Stack 33
Figura 22: Operazioni sullo Stack 34

6 Bibliografia

Tanembaum, *Architettura del Computer,* Ed. Prentice Hall-Jackson, 1991

Herbert Schildt, *C/C++ La guida Completa,* Ed. McGra-Hill, 1995

Anna Gentile, *Architetture dei Microprocessori 8086,* Ed. Laterza

C.Morgan, M.Waite, *Il manuale 8086/8088,* Ed. McGraw-Hill, 1985

Intel, *8086 16-bit Hmos microprocessor 8086/8086-2/8086-1,* Ed. Intel,1990.