UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Dispense del corso di Fondamenti di Informatica 1

L'elaboratore elettronico

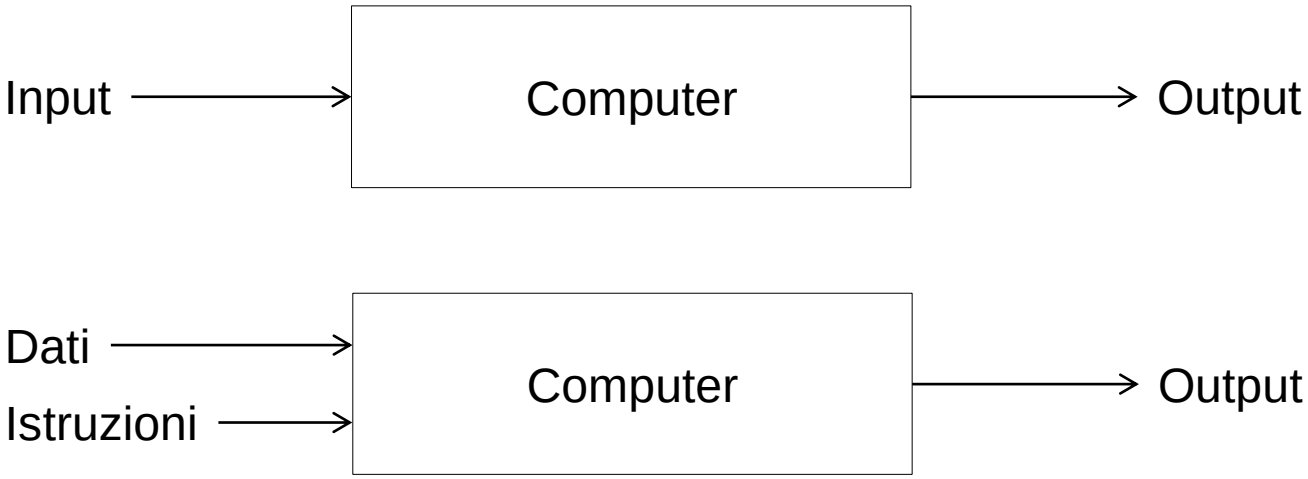
Ultimo aggiornamento: 01/10/2017

Dispense realizzate con il fondamentale contributo del Prof. Roberto Vezzani

GENERALITÀ

Computer o Calcolatore Elettronico

• **Computer**, o **calcolatore elettronico**: è una macchina che produce dati in uscita (output) sulla base delle informazioni che riceve in ingresso (input). Per un computer, sono input tanto i dati sui quali deve lavorare, quanto le istruzioni che deve eseguire per trasformare i dati in output.



Codifica del programma all'interno del calcolatore

- **Programma fisso**
- I primi calcolatori avevano un programma fisso. Alcuni calcolatori usano ancora questo schema, o per semplicità o per motivi didattici.
- Per esempio una calcolatrice è un calcolatore con programma fisso. Può eseguire calcoli matematici, ma non può fare altro (word processing, grafica, ...). Modificare il programma di una macchina a programma fisso richiede di riprogettare e ricostruire la macchina.

- I calcolatori successivi fecero un passo avanti rispetto a questo e vennero creati con *programma cablato*, ovvero si potevano comportare diversamente se si modificavano le connessioni di una serie di cavi. Per impostare un programma sul calcolatore ENIAC (1946) potevano volerci diversi giorni e fino a tre settimane.

- **Programma cablato**: l'unità di esecuzione è strutturata per eseguire una serie di compiti. L'ordine è stabilito tramite connessioni fisiche (cavi o ponticelli).

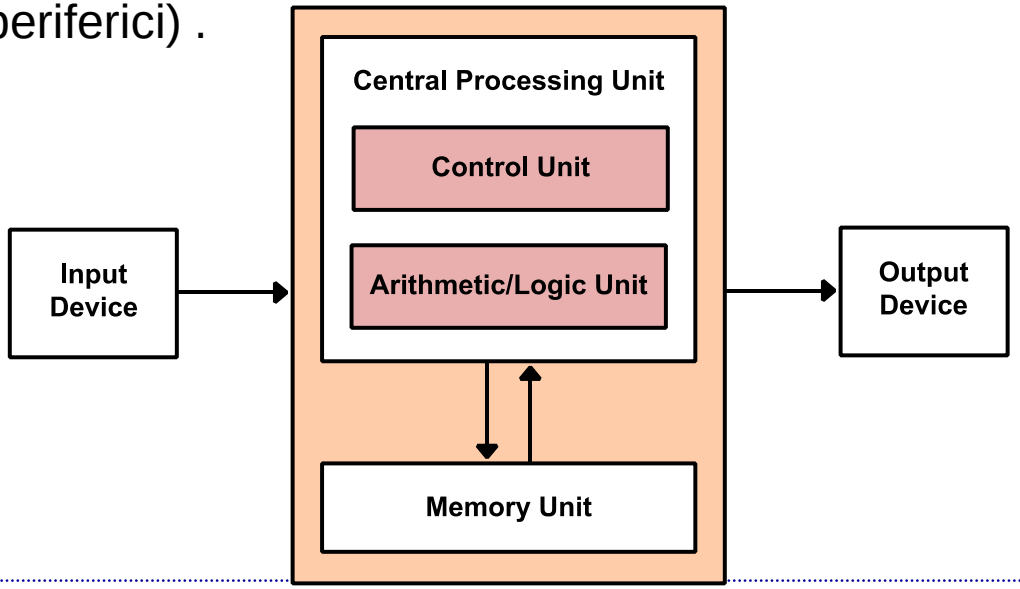
Codifica del programma all'interno del calcolatore

- Con la proposta del *programma memorizzato* questo cambiò radicalmente.
- Un computer a programma memorizzato viene progettato con la capacità di eseguire diverse azioni scelte da un insieme di istruzioni di base.
- Dispone poi di una *memoria* da cui preleva delle istruzioni (un programma) che descrivono quello che deve essere calcolato.

- **Programma memorizzato**
  istruzioni e dati sono codificati e caricati in una memoria, con la possibilità di eseguire programmi diversi → elaboratore elettronico

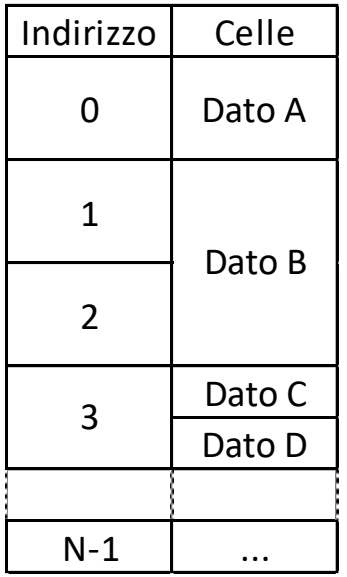
L'architettura di Von Neumann

- L'architettura di Von Neumann (1946) si compone di 3 parti fondamentali:
  - **MEMORIA**: unità che contiene sia le istruzioni che compongono il programma da eseguire, sia i dati che fungono da operandi per tali operazioni
  - **CPU** (Central Processing Unit) o processore: è composto da una unità di controllo per gestire le comunicazioni con tutte le unità del sistema e per gestire il prelievo delle istruzioni (fetch) e la loro esecuzione e da una unità aritmetico logica (ALU) che esegue le istruzioni stesse e che gestisce l'accesso agli operandi.
  - **INPUT/OUTPUT** le porte di interfaccia con l'esterno (e con i corrispondenti dispositivi periferici) .



La memoria dell'elaboratore

- È un dispositivo suddiviso logicamente in celle, di dimensione fissa, alle quali si può fare riferimento tramite un «indirizzo», ovvero un numero.
- È una struttura monodimensionale.
- Ogni dato può occupare una cella, più celle, o parte di una cella. Si possono quindi combinare più dati assieme e inserirli nella stessa cella, ma la memoria non ha supporto per modificare solo una parte di questa combinazione.



- Operazioni:
  - lettura del contenuto di una cella: il valore viene prelevato dalla memoria e portato nella CPU.
  - scrittura di una cella: il valore viene mandato dalla CPU e sostituisce il contenuto della memoria in quella cella.

IL PROCESSORE ADE8

Un elaboratore di esempio

- Per chiarire meglio che cosa sia un calcolatore elettronico useremo una macchina inventata, realizzabile, ma molto semplice e per questo inefficiente e solo didattica.
- Il processore si chiama **ADE8 (Architettura Di Esempio a 8 bit)** e consente di eseguire operazioni elementari su parole a 8 bit.

- Le CPU, e questa non fa eccezione, dispongono internamente di piccole zone di memoria in cui mantengono alcune informazioni. Queste si chiamano *registri*.

- ADE8 dispone di 4 registri interni:
  - accumulatore (ACC) a 8 bit
  - program counter (PC) a 8 bit
  - stack pointer (SP) a 8 bit
  - flag (FLAG) a 4 bit

Registri di ADE8

- L'**accumulatore** è il registro che partecipa sempre nelle operazioni matematiche e logiche. Uno degli operandi delle diverse operazioni è sempre memorizzato in questo spazio temporaneo. Il risultato delle operazioni viene posto sempre nell'accumulatore
- Il **program counter** è il registro che contiene l'indirizzo in memoria dove è presente la prossima istruzione da eseguire.
- Lo **stack pointer** è il registro che contiene l'indirizzo in memoria dove possono essere memorizzati dati o indirizzi temporanei o parametri per le funzioni.
  - In inglese *stack* significa «pila», nel senso di «pila di piatti» o «pila di giornali», e vuole ricordare il fatto che gli oggetti nella pila (nel nostro caso i dati) possono essere aggiunti o tolti dalla cima della pila.
  - Questo registro viene utilizzato nelle operazioni più complesse e quindi **per ora** non serve capirne il significato.
- Il registro **flag** viene utilizzato per mantenere lo stato del processore e avere informazioni sul risultato dell'ultima operazione aritmetica o logica eseguita.

Il registro FLAG

- Il registro FLAG contiene 4 bit che fungono da «bandiera» per segnalare che si è verificato un certo evento.
- In particolare in ADE8 solo le operazioni aritmetiche aggiornano questo registro.

3	2	1	0
Carry	Overflow	Negative	Zero

- Se il flag Zero vale 1, significa che l'operazione aritmetica o logica ha prodotto un risultato uguale a zero.
- Se il flag Negative vale 1, significa che l'operazione aritmetica ha prodotto un risultato negativo.
- Se il flag Overflow vale 1, significa che l'operazione aritmetica (tra numeri con segno) ha prodotto un overflow, cioè che il risultato non è rappresentabile a 8 bit.
- Se il flag Carry vale 1, significa che l'operazione aritmetica ha prodotto un riporto sul bit più significativo.

Istruzioni di ADE8

- Il programmatore ha a disposizione un insieme di 52 istruzioni con le quali è possibile effettuare alcune semplici operazioni sull'accumulatore, tra l'accumulatore e la memoria o modificare il punto di esecuzione.

- Esistono istruzioni che non richiedono informazioni aggiuntive e lavorano solo con l'accumulatore (ad esempio quella per calcolare l'opposto di un numero).

- Altre istruzioni invece richiedono di specificare, ad esempio, quale sia il secondo operando. Queste devono consentire al programmatore di fornire questa informazione, detta *parametro*.

- **Se ve lo state chiedendo, vi confermo che le istruzioni vanno saputе a memoria.**

Formato delle istruzioni

- Le istruzioni sono di lunghezza fissa (16 bit) e composte da due numeri a 8 bit:

codice operativo (*opcode*)	valore/indirizzo (*param*)

- Il *codice operativo* (*opcode*) è un numero che dice al processore che operazione deve eseguire.
- Il secondo campo *(param)* può essere o un valore (detto *immediato*) da utilizzare nell'operazione, oppure un indirizzo di memoria (*address*). Il significato può essere compreso solo dopo aver *decodificato* il codice operativo.

- Nelle operazioni che non richiedono parametri, di norma il campo *param* viene lasciato a 0, ma in ogni caso il processore lo ignora e quindi potrebbe valere qualsiasi valore.

- Ogni istruzione occupa quindi 2 byte. Se il codice operativo è all'indirizzo x, il corrispondente parametro si trova all'indirizzo $x + 1$.

Memoria di ADE8

- Nel progetto di ADE8, puntando alla semplicità, si sono utilizzati solo registri e dati a 8 bit.
- Siccome il PC è a 8 bit, come anche il campo *param*, gli indirizzi a cui ci si può riferire sono solo quelli da 0 a 255.
- Per simmetria, ogni cella di memoria (come accade anche in molti processori reali) può contenere un valore a 8 bit.
- Il processore può quindi interfacciarsi con una memoria che dispone di 256 celle da 1 byte. Per un totale di 256 byte di memoria (notate che sono byte e non Megabyte!).

Il ciclo delle istruzioni di ADE8

- Il processore lavora continuamente in questo modo:

1. Accede in memoria all'indirizzo specificato dal PC e legge il numero lì contenuto. Questo è (si spera) un codice operativo.
2. Incrementa il PC di 1.
3. Accede in memoria all'indirizzo PC (1 dopo il precedente) e legge il numero lì contenuto. Questo è il parametro dell'istruzione.
4. Incrementa nuovamente il PC di 1 (in modo da essere pronto a eseguire la prossima istruzione).
5. Guarda il valore dell'opcode e in base a quello esegue una determinata azione.
6. Ricomincia dalla fase 1.

- Le fasi da 1 a 4 sono dette (assieme) fase di *fetch*.
- La fase 5 viene detta fase di *execute*.

Reset

- Che cosa succede quando «accendiamo» il processore?
- Inizia una fase detta **reset**, in cui il sistema hardware si porta in uno stato iniziale noto e ben definito.
- Quando il processore ADE8 viene acceso (quando viene quindi fornita l'alimentazione elettrica), tutti i registri vengono azzerati e quindi sia l'accumulatore (ACC), sia il program counter (PC), sia lo stack pointer (SP) valgono 0.
- Cosa implica questo?
- Il processore inizia a caricare i due byte presenti agli indirizzi 0 e 1 in memoria e prosegue facendo quello che viene specificato dal codice operativo dell'istruzione.

Descrizione delle istruzioni

- Le istruzioni hanno codici numerici e ognuna ha un significato (usiamo il suffisso *h* per indicare che i numeri sono in esadecimale)
- Ad esempio:
  - L'istruzione 70h esegue il complemento a 2 del contenuto dell'accumulatore e il campo *param* non viene utilizzato.
  - L'istruzione 01h prende il valore di *param* e lo copia nell'accumulatore.
  - L'istruzione 00h prende l'indirizzo contenuto in *param*, va in memoria, preleva il contenuto della cella a quell'indirizzo e lo carica nell'accumulatore.
- Questo modo di specificare il comportamento di ogni istruzione è molto scomodo e incomprensibile. Quindi useremo una notazione più compatta e formale per chiarire il significato delle istruzioni.
- Ad esempio:
  - Istruzione 70h: ACC ← -ACC
  - Istruzione 01h: ACC ← *param*
  - Istruzione 00h: ACC ← mem[*param*]

«←» significa «assume il valore», il segno meno ha il suo significato matematico

«mem[...]» indica il contenuto della cella di memoria all'indirizzo indicato tra parentesi quadre

Mnemonici

- Per poter fare riferimento alle istruzioni in modo più comprensibile, si utilizzano delle sequenze di lettere dette *codici mnemonici* (o semplicemente *mnemonici*). Gli opcode sono tutti in esadecimale.

Gruppo 1	
NEG	70
NOT	71
LDv	01
ANDv	21
ORv	31
ADDv	41
SUBv	51
CMPv	61
LD	00
ST	10
AND	20
OR	30
ADD	40
SUB	50
CMP	60

Gruppo 2	
JMP	90
JZ	91
JN	92
JO	93
JC	94
JE	91
JLs	95
JLu	96

Gruppo 3	
LDi	02
STi	12
ANDi	22
ORi	32
ADDi	42
SUBi	52
CMPi	62

Gruppo 4	
PUSH	80
POP	81
CALL	82
RET	83
LDs	03
STs	13
ANDs	23
ORs	33
ADDs	43
SUBs	53
CMPs	63

Gruppo 5	
LA	01
LAs	86
RES	84
REL	85
LDsi	04
STsi	14
ANDsi	24
ORsi	34
ADDsi	44
SUBsi	54
CMPsi	64

Mnemonici

- Nel seguito analizzeremo prima le istruzioni «facili» (Gruppi 1 e 2). A seguire quelle del Gruppo 3 (accesso indiretto alla memoria) e infine quelle del Gruppo 4 (utilizzo dello stack). Il Gruppo 5 non è da sapere.

Gruppo 1	
NEG	70
NOT	71
LDv	01
ANDv	21
ORv	31
ADDv	41
SUBv	51
CMPv	61
LD	00
ST	10
AND	20
OR	30
ADD	40
SUB	50
CMP	60

Gruppo 2	
JMP	90
JZ	91
JN	92
JO	93
JC	94
JE	91
JLs	95
JLu	96

Gruppo 3	
LDi	02
STi	12
ANDi	22
ORi	32
ADDi	42
SUBi	52
CMPi	62

Gruppo 4	
PUSH	80
POP	81
CALL	82
RET	83
LDs	03
STs	13
ANDs	23
ORs	33
ADDs	43
SUBs	53
CMPs	63

Gruppo 5	
LA	01
LAs	86
RES	84
REL	85
LDsi	04
STsi	14
ANDsi	24
ORsi	34
ADDsi	44
SUBsi	54
CMPsi	64

Le istruzioni dell'ADE8 (1)

- Istruzioni aritmetiche senza parametri (il campo *param* non viene utilizzato)

NEG (70h): ACC ← **-**ACC (aggiorna i FLAG)

NOT (71h): ACC ← **not** ACC (aggiorna i FLAG)

- Istruzioni con un valore immediato (la *v* alla fine ricorda questo)

LDv (01h): ACC ← *param*

ANDv (21h): ACC ← ACC **and** *param* (aggiorna i FLAG)

ORv (31h): ACC ← ACC **or** *param* (aggiorna i FLAG)

ADDv (41h): ACC ← ACC **+** *param* (aggiorna i FLAG)

SUBv (51h): ACC ← ACC **-** *param* (aggiorna i FLAG)

CMPv (61h): ACC **-** *param* (aggiorna i FLAG)

L'istruzione CMPv esegue la differenza tra l'accumulatore e il valore nell'istruzione solo per aggiornare i FLAG, ma non memorizza il risultato da nessuna parte.

Le istruzioni dell'ADE8 (2)

- Istruzioni per lettura e scrittura in memoria

LD (00h): ACC ← mem[*param*]

ST (10h): mem[*param*] ← ACC

- L'istruzione *load* (LD) legge dalla memoria il dato all'indirizzo specificato come parametro e lo mette nell'accumulatore.

- L'istruzione *store* (ST) scrive il dato contenuto nell'accumulatore nella cella di memoria all'indirizzo specificato come parametro.

Le istruzioni dell'ADE8 (3)

- Istruzioni aritmetiche e logiche con la memoria

AND (20h):	ACC ← ACC **and** mem[*param*]	(aggiorna i FLAG)
OR (30h):	ACC ← ACC **or** mem[*param*]	(aggiorna i FLAG)
ADD (40h):	ACC ← ACC **+** mem[*param*]	(aggiorna i FLAG)
SUB (50h):	ACC ← ACC **-** mem[*param*]	(aggiorna i FLAG)
CMP (60h):	ACC **-** mem[*param*]	(aggiorna i FLAG)

- Il salto incondizionato

JMP (90h): PC ← *param*

Sostituendo il contenuto del PC, la prossima istruzione che verrà caricata sarà quella indicata dal programmatore, che ha quindi fatto «saltare» l'esecuzione al punto desiderato.

Le istruzioni dell'ADE8 (4)

- I salti condizionati dai flag

JZ (91h): se Z=1, allora PC ← *addr*
JN (92h): se N=1, allora PC ← *addr*
JO (93h): se O=1, allora PC ← *addr*
JC (94h): se C=1, allora PC ← *addr*

- Questi salti sono eseguiti solo se una condizione è vera e quindi consentono di regolare l'esecuzione in base al risultato di operazioni precedenti.

Le istruzioni dell'ADE8 (5)

- Esiste anche una interpretazione logica dei flag, che diventa chiara se osserviamo il loro valore dopo aver eseguito una delle istruzioni CMP

- *jump if equal*

JE (91h): se i due valori confrontati sono uguali allora PC ← *addr*
In realtà JE è un altro nome per JZ.

- *jump if lower*

JLs (95h): (per numeri signed) se il primo valore è minore del secondo allora PC ← *addr*

JLu (96h): (per numeri unsigned) se il primo valore è minore del secondo allora PC ← *addr*

Cosa possiamo fare con questa macchina?

- Proviamo a usarla come calcolatrice a 8 bit.
- Mettiamo due numeri in memoria agli indirizzi 08h e 09h, ad esempio i valori 23 (17h) e 48 (30h). Vogliamo programmare questa macchina in modo che esegua la somma e metta il risultato (47h) in 0Ah.
- Per eseguire questo compito quindi dobbiamo:
  - leggere il dato all'indirizzo 08h e metterlo nell'accumulatore
  - sommare all'accumulatore il dato all'indirizzo 09h
  - scrivere il valore dell'accumulatore in memoria all'indirizzo 0Ah
- Le istruzioni che ci servono sono LD, ADD e ST.
- Il programma che scriveremo in memoria (dall'indirizzo 0) sarà:

Indirizzo	**Valore**	**Note**
00h	00h	opcode: 00h (LD), param: 08h
01h	08h	
02h	40h	opcode: 40h (ADD), param: 09h
03h	09h	
04h	10h	opcode: 10h (ST), param: 0Ah
05h	0Ah	

Questo è il programma in *linguaggio macchina*:

`00 08 40 09 10 0A`

(i valori sono in esadecimale, non metto la h)

In modo più simbolico

- Possiamo visualizzare il programma in modo più leggibile:

```
00h:      00h 08h
02h:      40h 09h
04h:      10h 0Ah
```

- In questa rappresentazione abbiamo indicato l'indirizzo in memoria in cui è caricata l'istruzione. Ognuna è composta da due byte: opcode e param.

- Ancora più leggibile è la sostituzione del codice operativo con il codice mnemonico che abbiamo definito:

```
00h:      LD    08h
02h:      ADD   09h
04h:      ST    0Ah
```

- Questo è il cosiddetto *linguaggio assembly* o, in un poco comune italiano, *linguaggio assemblativo* o *assemblato*.

Dopo l'esecuzione che cosa accade?

- Il processore non conosce le nostre intenzioni e **continua a leggere e ad eseguire il contenuto della memoria** senza "capire" che il programma è finito.
- Per forzare uno «stop», non esiste in questo processore (e in nessun altro) un'istruzione apposita, semplicemente perché i processori non si fermano mai! C'è sempre qualche programma in esecuzione, di solito il sistema operativo.
- Per fare in modo che il nostro programma «si fermi», possiamo utilizzare un salto. Un salto alla stessa istruzione è quello che genera un ciclo infinito:

```
00h:      LD    08h
02h:      ADD   09h
04h:      ST    0Ah
06h:      JMP   06h
```

Assemblatore

- L’operazione di traduzione dalla forma simbolica dell’assembly al linguaggio macchina può essere realizzata tramite un programma che automaticamente traduce gli mnemonici in codici operativi.
- Ogni volta che legge una «parola», ovvero una sequenza di caratteri diversa da spazio, tab o a capo, se questa è NEG, NOT, ldv, eccetera, viene sostituita dal byte corrispondente all’*opcode* corretto.

- L’altra funzione che l'assemblatore può eseguire è quella di consentire la definizione di *etichette simboliche* o *label*, che devono essere tradotte in indirizzi.
- In pratica possiamo dare un nome alla cella di memoria in cui il numero verrà poi inserito. In questo modo invece che riferirci alle celle con il loro indirizzo, possiamo utilizzare un nome ed esprimere meglio il significato di quel valore.

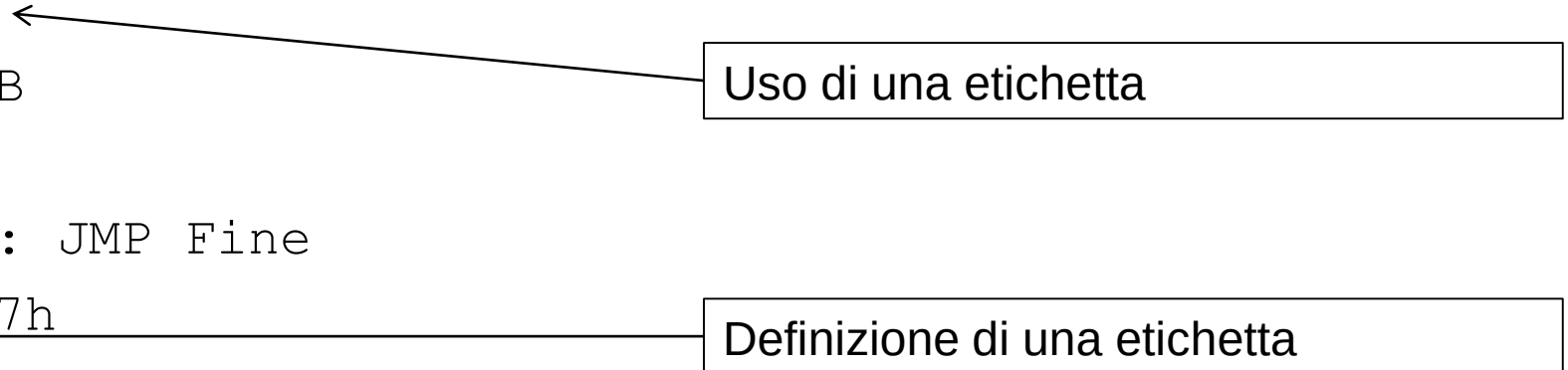
Assemblatore

- Inoltre l'assemblatore può essere in grado di capire se in una linea sto scrivendo un valore, un'istruzione o un'etichetta. Per fare questo è sufficiente vedere se il «pezzo di testo» che sto considerando (delimitato da spazi o tabulazioni o a capo, detto anche *token*) inizia con una lettera o una cifra:
  - 04h → è certamente un valore.
  - ADD → potrebbe essere una istruzione o una label. È una istruzione nota? Sì, allora la sostituisco con il suo codice operativo.
  - A → potrebbe essere una istruzione o una label. È una istruzione nota? No, allora cerco dove è definita e la sostituisco con l'indirizzo corrispondente.
- Notare che l'assemblatore può capire sia il decimale sia l'esadecimale, quindi inseriamo in fondo una *h* per indicare la base del nostro numero. Un numero deve sempre iniziare con una cifra da 0 a 9, quindi FFh verrebbe interpretato come una etichetta! Bisogna scrivere 0FFh.
- Un'altra caratteristica è quella di consentire l'inserimento di *commenti*. Per farlo, basta mettere su una riga un «;». Da quel punto fino alla fine della riga (cioè fino al primo a capo) tutti i caratteri verranno ignorati.

Da assembly a linguaggio macchina

- Con l'uso di etichette simboliche il nostro programma diventerebbe:

```
LD A
ADD B
ST C
Fine: JMP Fine
A: 17h
B: 30h
C: 00h
```



- Il simbolo «:» viene interpretato come la *definizione di una etichetta*. Una volta noto l'indirizzo in cui si troverà l'elemento di quella riga, la label può essere sostituita con quell'indirizzo.
- È chiaro che serviranno **due passate** sul testo per poter calcolare gli indirizzi: prima si vede quanto occupa ogni riga una volta tradotta in dati, poi si sostituiscono le label.

Da assembly a linguaggio macchina

- L'assemblatore normalmente ignora gli spazi e le tabulazioni, quindi possiamo utilizzare le tabulazioni per rendere più leggibile il programma:

```
            LD    A
            ADD   B
            ST    C
Fine:       JMP   Fine
A:          17h
B:          30h
C:          00h
```

- Inoltre l'assemblatore accetta le istruzioni sia in maiuscolo, sia in minuscolo, quindi si può scrivere JMP o jmp (o anche JmP, ma è meglio evitare).
- **Le etichette invece devono essere scritte esattamente come sono definite, rispettando maiuscole e minuscole.**

Da assembly a linguaggio macchina

- Calcoliamo quanto occupa ogni riga:

```
            LD    A            istruzione, 2 byte
            ADD   B            istruzione, 2 byte
            ST    C            istruzione, 2 byte
Fine:       JMP   Fine         istruzione, 2 byte
A:          17h                dato, 1 byte
B:          30h                dato, 1 byte
C:          00h                dato, 1 byte
```

Da assembly a linguaggio macchina

- Calcoliamo l'indirizzo a partire dal quale verrà inserita ogni «riga»:

```
            LD    A          00h
            ADD   B          02h
            ST    C          04h
Fine:       JMP   Fine       06h
A:          17h              08h
B:          30h              09h
C:          00h              0Ah
```

- Possiamo quindi *risolvere* le label:
  - `Fine = 06h`
  - `A = 08h`
  - `B = 09h`
  - `C = 0Ah`

Da assembly a linguaggio macchina

- Sostituendo le label con il loro indirizzo:

```
            LD    A              00h: LD    08h
            ADD   B              02h: ADD   09h
            ST    C              04h: ST    0Ah
Fine:       JMP   Fine     →     06h: JMP   06h
A:          17h                  08h: 17h
B:          30h                  09h: 30h
C:          00h                  0Ah: 00h
```

- Label risolte:
  - Fine = 06h
  - A = 08h
  - B = 09h
  - C = 0Ah

Da assembly a linguaggio macchina

- Sostituendo gli mnemonici con il loro opcode:

```
            LD    A              00h: 00h    08h
            ADD   B              02h: 40h    09h
            ST    C              04h: 10h    0Ah
Fine:       JMP   Fine    →      06h: 90h    06h
A:          17h                  08h: 17h
B:          30h                  09h: 30h
C:          00h                  0Ah: 00h
```

- Il nostro programma diventa quindi (tutti i valori sono in esadecimale, non metto la h):

```
00 08 40 09 10 0a 90 06 17 30 00
```

Da assembly a linguaggio macchina

- Notate che per fare riferimento ad una cella di memoria, devo specificare all'assemblatore un dato a cui fare riferimento, quindi per la variabile C ho dovuto definire un byte col valore 0, anche se non ha alcun senso, visto che l'unica cosa che faccio e scriverci dentro il risultato.
- Se scrivessi un programma così (senza senso!):

```
A: 17h
22h 4Fh
B: 30h 0C2h
```

- In memoria l'assemblatore produrrebbe:

```
17 22 4F 30 C2
```

- Ma non saprei (almeno per ora) come far riferimento ai valori 22h, 4fh e C2h. Ogni numero che scrivo diventa un byte in linguaggio macchina.
- Notate che per scrivere il numero esadecimale C2, ho dovuto scrivere 0C2h, in modo che cominci con una cifra e non con una lettera. Ricordate che comunque occupa sempre 8 bit e non 12!

Il valore assoluto di un numero

- Scrivere il programma che dato un numero in memoria (A) ne calcola il valore assoluto e lo mette ad un altro indirizzo (B).

Il valore assoluto di un numero

- Scrivere il programma che dato un numero in memoria (A) ne calcola il valore assoluto e lo mette ad un altro indirizzo (B).

```
                LD      A
                CMPv    0
                JLs     Negativo
                JMP     Store
Negativo:       NEG
Store:          ST      B
Fine:           JMP     Fine
A:              17h
B:              00h
```

Il valore assoluto di un numero

- Scrivere il programma che dato un numero in memoria (A) ne calcola il valore assoluto e lo mette ad un altro indirizzo (B).

```
                LD      A
                CMPv    0
                JLs     Negativo
                JMP     Store
Negativo:       NEG
Store:          ST      B
Fine:           JMP     Fine
A:              17h
B:              00h
```

- In linguaggio macchina diventa:

```
00 0e 61 00 95 08 90 0a
70 00 10 0f 90 0c 17 00
```

Il valore assoluto di un numero

- Scrivere il programma che dato un numero in memoria (A) ne calcola il valore assoluto e lo mette ad un altro indirizzo (B).

```
                LD      A
                CMPv    0
                JLs     Negativo
                JMP     Store
Negativo:       NEG
Store:          ST      B
Fine:           JMP     Fine
A:              17h
B:              00h
```

Da adesso in avanti utilizzeremo i colori per rendere più leggibile il codice. Questa modalità di mostrare il testo si chiama *syntax highlighting.* Ad esempio, i numeri vengono mostrati in rosso, le istruzioni in blu e le etichette in nero.

- In linguaggio macchina diventa:

```
00 0e 61 00 95 08 90 0a
70 00 10 0f 90 0c 17 00
```

Variabile

- Abbiamo utilizzato alcune locazioni di memoria per contenere dati.
- Questi dati vengono modificati durante l'esecuzione del programma. Diciamo quindi che un dato in memoria è variabile, ovvero può essere modificato (variato).
- Una *variabile* è quindi **un dato che viene mantenuto in qualche cella di memoria**.
- Per ora abbiamo sempre visto variabili che contenevano numeri interi a 8 bit. In particolare nel primo esempio non è importante decidere se a 8 bit con segno o senza segno. Nel secondo (il valore assoluto) erano variabili numeriche intere a 8 bit con segno.
- Una variabile può occupare anche più di una cella di memoria, come ad esempio un numero intero a 16 bit.

La moltiplicazione intera

- La moltiplicazione intera a 8 bit senza segno (ignorando che può sforare) si può realizzare sommando il moltiplicando tante volte quanto indicato dal moltiplicatore. Quindi $c \leftarrow a \times b$ si può realizzare così:

La moltiplicazione intera

- La moltiplicazione intera a 8 bit senza segno (ignorando che può sforare) si può realizzare sommando il moltiplicando tante volte quanto indicato dal moltiplicatore. Quindi $c \leftarrow a \times b$ si può realizzare così:

```
Inizio: LD      B
        CMPv    0       ; B = 0?
        JE      Fine
        LD      C
        ADD     A       ; C <- C + A
        ST      C
        LD      B
        SUBv    1       ; B <- B - 1
        ST      B
        JMP     Inizio
Fine:   JMP     Fine
A:      3       ; Moltiplicando
B:      5       ; Moltiplicatore
C:      0       ; Prodotto
```

La moltiplicazione intera

- Calcoliamo gli indirizzi:

```
00:(Inizio) LD      B
02:         CMPv    0
04:         JE      Fine
06:         LD      C
08:         ADD     A
0A:         ST      C
0C:         LD      B
0E:         SUBv    1
10:         ST      B
12:         JMP     Inizio
14:(Fine)   JMP     Fine
16:(A)      3
17:(B)      5
18:(C)      0
```

La moltiplicazione intera

- Sostituiamo le etichette:

```
00:(Inizio) LD      17
02:         CMPv    0
04:         JE      14
06:         LD      18
08:         ADD     16
0A:         ST      18
0C:         LD      17
0E:         SUBv    1
10:         ST      17
12:         JMP     0
14:(Fine)   JMP     14
16:(A)      3
17:(B)      5
18:(C)      0
```

La moltiplicazione intera

- Sostituiamo gli mnemonici:

```
00:     00    17
02:     61    0
04:     91    14
06:     00    18
08:     40    16
0A:     10    18
0C:     00    17
0E:     51    1
10:     10    17
12:     90    0
14:     90    14
16:     3
17:     5
18:     0
```

La moltiplicazione intera

- Il programma definitivo diventa:

```
00 17 61 00 91 14 00 18
40 16 10 18 00 17 51 01
10 17 90 00 90 14 03 05
00
```

Somma dei numeri da 0 a N

- Si calcoli la somma dei numeri da 0 a N (variabile in memoria), e si metta il risultato in una variabile S. L'operazione si può fare con 8 bit solo se $N \leq 22$.

Somma dei numeri da 0 a N

- Si calcoli la somma dei numeri da 0 a N (variabile in memoria), e si metta il risultato in una variabile S. L'operazione si può fare con 8 bit solo se $N \leq 22$.

```
Inizio: LD      N
        CMPv    0
        JE      Fine
        ADD     S
        ST      S
        LD      N
        SUBv    1
        ST      N
        JMP     Inizio
Fine:   JMP     Fine
N:      5
S:      0
```

Somma dei numeri da 0 a N

- Si calcoli la somma dei numeri da 0 a N (variabile in memoria), e si metta il risultato in una variabile S. L'operazione si può fare con 8 bit solo se $N \leq 22$.

```
Inizio: LD      N
        CMPv    0
        JE      Fine
        ADD     S
        ST      S
        LD      N
        SUBv    1
        ST      N
        JMP     Inizio
Fine:   JMP     Fine
N:      5
S:      0
```

- Il programma in linguaggio macchina è:

```
00 14 61 00 91 12 40 15 10 15 00 14 51 01 10 14
90 00 90 12 05 00
```

Altri esercizi

- Calcolare il massimo tra A e B (interi a 8 bit con segno) e mettere il risultato in C.
- Calcolare il massimo tra A e B (interi a 8 bit senza segno) e mettere il risultato in C.

- Controllare se A è pari. Se sì B deve valere 1, altrimenti 0.

- Controllare se A è positivo. Se sì B deve valere 1, altrimenti 0.

- Scambiare due byte A e B.

- Calcolare il M.C.D. tra A e B utilizzando l'algoritmo di Euclide (http://utenti.quipo.it/base5/numeri/euclidalgor.htm). Il risultato si troverà in B.

L'ACCESSO INDIRETTO

Lavorare con più dati

- Per ora abbiamo visto programmi che lavorano con alcuni dati, ognuno con una propria «identità». Ovvero operazioni che coinvolgono pochi elementi.
- La potenza del calcolatore però è proprio quella di poter elaborare grandi quantità di dati.
- Supponiamo di voler sommare 10 numeri. Questi numeri appartengono ad esempio all'insieme $X = \{x_1, x_2, \dots, x_{10}\}$. In formule matematiche, scriveremmo che:

$$S = \sum_{i=1}^{10} x_i$$

- generalizzando quindi l'espressione. Infatti se X contiene n elementi, basta sostituire al 10 il valore n.
- Per quello che abbiamo visto finora, non appare molto semplice fare questo passaggio. Vediamo una prima soluzione. I numeri da sommare sono 12,16,27,2,41,79,5,28,3,30. La somma fa 243 (F3h), quindi limitiamoci a fare la somma con risultato in un solo byte.

Somma di 10 numeri

```
          JMP     Inizio                  (segue)
S:        0
Inizio:   LDV     0                x3:    27
          ADD     x1               x4:    2
          ADD     x2               x5:    41
          ADD     x3               x6:    79
          ADD     x4               x7:    5
          ADD     x5               x8:    28
          ADD     x6               x9:    3
          ADD     x7               x10:   30
          ADD     x8
          ADD     x9
          ADD     x10
          ST      S
Fine:     JMP     Fine

x1:       12
x2:       16
```

Somma di N numeri

- Modificare il numero di dati è veramente scomodo, perché dobbiamo definire una nuova variabile per ogni dato e aggiungere un'istruzione di somma per ogni nuovo dato.
- Insomma un programma diverso per ogni diverso numero di dati!
- È chiaro che non può essere questo il modo di scrivere programmi.
- Possiamo notare però che le istruzioni che eseguono la somma sono tutte molto simili. Infatti «add x1» in linguaggio macchina diventa 3a 1d, mentre «add x2» diventa 3a 1e. L'unica cosa che cambia è l'indirizzo in memoria che contiene il dato *i*-esimo. Se il dato *i* è all'indirizzo *x*, il dato *i+1*, si troverà all'indirizzo *x+1*. Quindi se potessimo cambiare il campo param dell'istruzione, sommandogli 1, otterremmo il risultato desiderato.
- Ma noi possiamo cambiare le istruzioni?
- Le istruzioni che cosa sono? Sono numeri in una memoria. Visto che possiamo leggere e scrivere quella memoria, possiamo anche cambiare il valore di una cella. E quindi modificare le istruzioni.
- Vediamo come potremmo fare.

Somma di N numeri

```
        jmp     Inizio
S:      0
Inizio:

Ciclo:
        ld      S
Somma:  add
Cosa:   x
        st      S

        ld      Cosa
        addv    1
        st      Cosa

        ld      N
        subv    1
        st      N

        jz      Fine
        jmp     Ciclo

Fine:   jmp     Fine

N:      10
x:      12 16 27 2 41 79 5 28 3 30
```

Notate che ho scritto lo mnemonico e il parametro su linee diverse per poter assegnare un etichetta anche al parametro

Qui modifico l'istruzione incrementando di 1 il parametro (l'indirizzo della cella di memoria da sommare)

Decremento il numero di elementi da sommare

Se la subv ha ritornato zero ho finito, altrimenti continuo.

Basta un'etichetta al primo byte, gli altri saranno agli indirizzi successivi.

I programmi automodificanti

- Quello che abbiamo fatto è creare un programma che modifica le proprie istruzioni. Si chiama programma *automodificante*.
- Purtroppo, appena le cose diventano un po' più complicate questo modo di scrivere i programmi diventa incomprensibile, perché non sappiamo più quale istruzione verrà eseguita, dato che la modifichiamo ogni volta.
- Per evitare questa situazione, bisogna che il processore ci metta a disposizione un supporto hardware, cioè delle istruzioni, che consentano di fare quello che abbiamo appena visto, ovvero accedere ad un indirizzo di memoria *variabile*.
- Perché questo sia variabile (o «una variabile»), deve essere memorizzato da qualche parte. Nel nostro caso in memoria.
- L'accesso quindi deve avvenire in modo *indiretto*, cioè non specificando nell'istruzione l'indirizzo da cui leggere, ma quello della variabile che contiene questo indirizzo.
- Una variabile che contiene un indirizzo viene chiamata *puntatore*.

Accesso indiretto alla memoria

- Istruzioni per lettura e scrittura in memoria in modo indiretto, ovvero con un *puntatore* ad un dato a 8 bit.

LDi (02h): ACC ← mem[mem[*param*]]

STi (12h): mem[mem[*param*]] ← ACC

- Queste due istruzioni sono piuttosto complesse: vanno in memoria all'indirizzo *param*, prelevano un numero a 8 bit e lo usano come indirizzo per accedere ad un'altra cella di memoria (per leggerla o per scriverci dentro).
- La «i» dopo LD e ST indica quindi *indiretto*.
- Vediamo come diventa il nostro programma utilizzando queste istruzioni.

Accesso indiretto alla memoria

```
        jmp     Inizio
S:      0
Inizio: ldv     x
        st      P

Ciclo:  ldi     P
        add     S
        st      S

        ld      P
        addv    1
        st      P

        ld      N
        subv    1
        st      N

        jz      Fine
        jmp     Ciclo

Fine:   jmp     Fine

N:      10
x:      12 16 27 2 41 79 5 28 3 30
P:      0
```

Imposto il puntatore all'indirizzo x

Leggo la cella *puntata* da P e ci sommo S. Notate che ho scambiato l'ordine delle operazioni rispetto a prima.

Incremento il puntatore.

Il puntatore è una variabile in memoria come le altre. Quello che cambia è come lo uso.

Accesso indiretto alla memoria

- Apposite istruzioni con accesso indiretto alla memoria possono essere create anche per le operazioni aritmetico logiche diadiche.
- Per simmetria con le versioni immediate ed ad accesso diretto, si avranno quindi le seguenti istruzioni:

ANDi (22h):	ACC ← ACC **and** mem[mem[*param*]]	(aggiorna i FLAG)
ORi (32h):	ACC ← ACC **or** mem[mem[*param*]]	(aggiorna i FLAG)
ADDi (42h):	ACC ← ACC **+** mem[mem[*param*]]	(aggiorna i FLAG)
SUBi (52h):	ACC ← ACC **-** mem[mem[*param*]]	(aggiorna i FLAG)
CMPi (62h):	ACC **-** mem[mem[*param*]]	(aggiorna i FLAG)

Altri esercizi

- Calcolare il massimo di un insieme di numeri X (interi a 8 bit con segno) e mettere il risultato in Max.
- Calcolare il massimo di un insieme di numeri X (interi a 8 bit senza segno) e mettere il risultato in Max.

- Calcolare il minimo di un insieme di numeri X (interi a 8 bit con segno) e mettere il risultato in Min.
- Calcolare il minimo di un insieme di numeri X (interi a 8 bit senza segno) e mettere il risultato in Min.

- Controllare se tutti gli elementi di un insieme di numeri X sono pari. Se sì B deve valere 1, altrimenti 0.

- Controllare se tutti gli elementi di un insieme di numeri X sono positivi. Se sì B deve valere 1, altrimenti 0.

INPUT E OUTPUT

Input e output

- Come si interfaccia un processore con l'esterno?
- Anche in questo caso, la magia non c'entra e il meccanismo è abbastanza semplice.
- Alcuni indirizzi di memoria, non sono collegati alla memoria, ma ad altri dispositivi.
- In particolare, nelle scelte fatte durante la realizzazione di ADE8 e della sua interfaccia esterna, si è deciso che 16 indirizzi sono collegati ad altri dispositivi. Gli indirizzi sono quelli da D0h fino ad DFh. In pratica se i 4 bit più significativi dell'indirizzo valgono 1101, allora la memoria li ignora e toccherà a qualche altro dispositivo rispondere.
- Gli altri quattro bit invece vengono detti *porta* di input/output.
- Quindi per inviare il byte contenuto nell'accumulatore alla porta 5, basta fare: ST 0D5h. Per leggere dalla porta 8: LD 0D8h. (Ricordate lo zero davanti ai numeri esadecimali per far sì che l'assemblatore li riconosca).

- Ma che cosa c'è attaccato a queste porte?

Dispositivi

- La trattazione del funzionamento dei dispositivi di input/output è decisamente al di fuori degli argomenti del corso, ma per fare un esempio semplice sul simulatore sono stati collegati due dispositivi di output e uno di input:
  - alla porta 0 c'è un display a 7 segmenti (i classici numeri da sveglia digitale) con due cifre, che visualizza in esadecimale il valore scritto.
  - alla porta 1 c'è un display a caratteri, che visualizza i caratteri corrispondenti ai codici ASCII (a 7 bit) scritti. Dopo ogni scrittura, la posizione di visualizzazione si sposta avanti e può quindi essere utilizzato per visualizzare brevi messaggi, scrivendo sulla porta una sequenza di byte corrispondenti ai caratteri ASCII che si vogliono ottenere.
  - alla porta 2 sono collegati 8 interruttori che possono essere impostati a 0 o a 1. Tutti assieme formano un valore a 8 bit che può essere letto da ADE8.
  - alla porta 3 c'è un dispositivo per controllare lo stato dell'input.
- L'input è piuttosto complesso e non ne parleremo nel dettaglio.
- Proviamo allora a visualizzare «Hello world!» sul display a caratteri.
- I valori ASCII necessari (in esadecimale) sono: 48 65 6C 6C 6F 20 77 6F 72 6C 64 21.

Il primo output di ADE8

```
        ldv     48h
        st      0d1h
        ldv     65h
        st      0d1h
        ldv     6ch
        st      0d1h
        ldv     6ch
        st      0d1h
        ldv     6fh
        st      0d1h
        ldv     20h
        st      0d1h
        ldv     77h
        st      0d1h
        ldv     6fh
        st      0d1h
        ldv     72h
        st      0d1h
        ldv     6ch
        st      0d1h
        ldv     64h
        st      0d1h
        ldv     21h
        st      0d1h

Fine:   jmp     Fine
```

Imposto l'accumulatore al codice ASCII corrispondente a «H»

Scrivo il valore all'indirizzo D1, ovvero lo invio in output alla porta 1.

Estendiamo le capacità dell'assemblatore

- Abbiamo già visto che ricordare a memoria gli opcode delle istruzioni è scomodo e possiamo utilizzare un programma che si occupa di sostituire agli mnemonici il corrispondente opcode.
- Analogamente è scomodo ricordare i codici ASCII. Lasciamo che ci pensi l'assemblatore a fare la sostituzione. Purtroppo, se scriviamo H, l'assemblatore la interpreta come una etichetta. Serve quindi un simbolo speciale per dirgli che il carattere che scriviamo è un byte e vogliamo che lo sostituisca con il corrispondente codice ASCII.
- La convenzione è molto semplice: si include il carattere che deve essere convertito tra apici singoli. Quindi scrivere:

```
ldv     48h
st      0d1h
```

- è equivalente a

```
ldv     'H'
st      0d1h
```

- Però in questo modo è molto più chiaro il significato! Vediamo quindi il risultato.

Seconda versione

```
        ldv     'H'
        st      0d1h
        ldv     'e'
        st      0d1h
        ldv     'l'
        st      0d1h
        ldv     'l'
        st      0d1h
        ldv     'o'
        st      0d1h
        ldv     ' '
        st      0d1h
        ldv     'w'
        st      0d1h
        ldv     'o'
        st      0d1h
        ldv     'r'
        st      0d1h
        ldv     'l'
        st      0d1h
        ldv     'd'
        st      0d1h
        ldv     '!'
        st      0d1h

Fine:   jmp     Fine
```

- Andiamo meglio, ma certo non è il massimo dell'intelligenza. Ogni volta che cambiamo la sequenza, dobbiamo modificare in giro per il programma le singole lettere.
- Inoltre si ripete continuamente l'istruzione di invio alla porta 1.
- Proviamo a sfruttare l'accesso indiretto, come fatto per le sequenze numeriche. Anche perché anche questa è una sequenza numerica.

Terza versione

```
Ciclo:
        ldi     P
        st      0d1h

        ld      P
        addv    1
        st      P

        ld      N
        subv    1
        st      N

        jz      Fine
        jmp     Ciclo

Fine:   jmp     Fine

N:      12
P:      msg
msg:    'H' 'e' 'l' 'l' 'o' ' ' 'w' 'o' 'r' 'l' 'd' '!'
```

Carico il codice ASCII puntato da P

Incremento P

Decremento N

Notate che P è inizializzato a msg, ovvero all'indirizzo a cui si trova il primo carattere da inviare in output.

Ulteriori miglioramenti

- Restano due scomodità:
  - scrivere una parola intera con tutti quegli apici è molto fastidioso!
  - ci tocca contare quanti caratteri ha la frase, includendo spazio e punto esclamativo.
- Il primo problema lo risolviamo facendo di nuovo lavorare l'assemblatore. Se tra apici ci sono più caratteri, lui li sostituirà tutti con il corrispondente codice ASCII. Quindi:

```
'H' 'e' 'l' 'l' 'o' ' ' 'w' 'o' 'r' 'l' 'd' '!'
```

- sarà equivalente a

```
'Hello world!'
```

- decisamente più comprensibile.
- Il secondo problema è più complesso, ma può essere risolto osservando che il valore 0 come codice ASCII non corrisponde a nessun simbolo utile. Quindi possiamo utilizzarlo come valore che, se incontrato, indica la fine della sequenza di caratteri da visualizzare.
- Ecco come si potrebbe scrivere un programma che utilizza questo «trucco».

Quarta versione

```
Ciclo:
        ldi     P
        cmpv    0
        je      Fine

        st      0d1h

        ld      P
        addv    1
        st      P

        jmp     Ciclo

Fine:   jmp     Fine

P:      msg
msg:    'Hello world!' 0
```

Carico il numero puntato da P

Il valore letto è 0? Se sì, vai alla fine.

Ecco la sequenza di numeri da inviare in output, seguita dal numero 0 che ne indica la fine.

I SOTTOPROGRAMMI E LO STACK

Ancora sul programma realizzato

- Osservando il programma che abbiamo realizzato, è facile notare che questo permette una sola esecuzione.
- Infatti il puntatore P viene inizializzato a msg solo al caricamento del programma e successivamente viene modificato.
- Un programma previdente, evita di assumere qualsiasi cosa sul valore delle proprie variabili e specifica chiaramente che cosa vuole in input.
- In questo caso potremmo definire una serie di istruzioni che si occupa di visualizzare un messaggio sul display e che richiede in ingresso che sull'accumulatore venga fornito l'indirizzo del messaggio da visualizzare.
- In questo modo potremo *riutilizzare* il blocco di istruzioni anche in altri programmi.
- Vediamo come potremmo organizzare il programma con questo scopo.

Quinta versione

```
        ldv     msg
        jmp     Display

Fine:   jmp     Fine
msg:    'Hello world!' 0

; Blocco di istruzioni per visualizzare il
; messaggio contenuto in ACC
Display:
        st      P
Ciclo:
        ldi     P
        cmpv    0
        je      DisplayFine

        st      0d1h

        ld      P
        addv    1
        st      P

        jmp     Ciclo

DisplayFine:
        jmp     Fine
P:      0
```

Metto in ACC l'indirizzo del messaggio da visualizzare.

Salto al blocco di istruzioni che svolge il compito.

Metto in P il valore di ACC (l'indirizzo del primo codice da inviare).

Se il messaggio è terminato salto alla fine del blocco.

Terminato il blocco torno alla fine del programma.

Riutilizzare il codice

- Per quanto elegante, il blocco di istruzioni che abbiamo scritto, non è realmente riutilizzabile.
- Infatti, supponiamo di voler scrivere due messaggi: «Hello » e «world!» in momenti diversi. Non possiamo saltare due volte a Display, infatti, dopo la prima volta, salterebbe a Fine, terminando il programma.
- Servirebbe un meccanismo che permetta di ritornare all'istruzione successiva al JMP. Ovvero servirebbe poter modificare l'istruzione che effettua il ritorno dal blocco di istruzioni Display.
- Ma noi possiamo modificare le istruzioni e l'abbiamo già fatto! Basta cambiare l'indirizzo dell'ultimo salto.
- Un blocco di istruzioni che possa ritornare alla posizione chiamante (o meglio all'istruzione successiva) viene detto *funzione*. Quindi d'ora in avanti non parleremo più di «blocco di istruzioni», ma di «funzione».
- Vediamo come possiamo realizzare una funzione con quello che sappiamo.

Sesta versione

```
        ldv     Return1
        st      DisplayReturnAddress
        ldv     msg1
        jmp     Display
Return1:
        ldv     Return2
        st      DisplayReturnAddress
        ldv     msg2
        jmp     Display
Return2:
Fine:   jmp     Fine

msg1:   'Hello ' 0
msg2:   'world!' 0

; Funzione per visualizzare il messaggio contenuto in ACC
Display:
        st      P
Ciclo:  ldi     P
        cmpv    0
        je      DisplayFine
        st      0d1h
        ld      P
        addv    1
        st      P
        jmp     Ciclo
DisplayFine:
        jmp
DisplayReturnAddress:
        0
P:      0
```

Scrivo sul byte DisplayReturnAddress l'indirizzo a cui tornare.

Qui faccio lo stesso, ma questa volta torno in un posto diverso.

DisplayReturnAddress è il campo *param* del salto finale.

Le funzioni

- Quello che abbiamo fatto è stato fondamentalmente fare un altro programma automodificante per cambiare il «posto» a cui si torna.
- Come già detto questa soluzione crea delle difficoltà e delle complessità che sarebbe meglio evitare. Serve un supporto hardware per questa funzionalità.
- Inoltre c'è un problema non indifferente che si può osservare quando una funzione deve chiamare se stessa (funzione ricorsiva). Se una funzione dovesse avere bisogno di «auto chiamarsi», cancellerebbe il primo indirizzo di ritorno.
- Può sembrare una follia, ma ci sono diversi casi in cui questo può essere utile. Pertanto sarebbe meglio consentire una gestione più flessibile della posizione dell'indirizzo di ritorno.
- Questo viene fornito in ADE8 (e in moltissime altre architetture) dal supporto hardware allo *stack*.
- Il meccanismo dello stack è molto comune in informatica e ne descriviamo brevemente il funzionamento, lasciando però la trattazione completa a corsi successivi.

Lo stack

- La gestione dello stack, o meglio la *gestione della memoria a stack*, viene gestita da un registro detto *stack pointer* (SP).
- SP contiene sempre l'indirizzo dell'ultimo dato salvato.
- Il meccanismo utilizzato in ADE8 è copiato da quello dei processori Intel.
- Quando voglio salvare un dato sullo stack, decremento SP e scrivo all'indirizzo contenuto in SP il dato (operazione di *push*).
- Quando voglio estrarre un dato dallo stack, leggo il dato all'indirizzo contenuto in SP e incremento SP (operazione di *pop*).

- Perché questo c'entra qualcosa con il nostro problema delle funzioni?
- Per chiamare una funzione possiamo fare una push dell'indirizzo successivo al salto, mentre per ritornare possiamo semplicemente fare il pop dell'indirizzo contenuto nello stack e mettere quell'indirizzo nel PC.

Istruzioni per lavorare con lo stack

- Istruzioni per l'utilizzo dello stack:

PUSH (80h): SP ← SP - 1
mem[SP] ← ACC
POP (81h): ACC ← mem[SP]
SP ← SP + 1

- La prima istruzione mette il valore dell'accumulatore sullo stack, mentre la seconda estrae l'ultimo valore dallo stack e lo mette nell'accumulatore.
- Con queste istruzioni possiamo salvare dati temporanei sullo stack e recuperarli successivamente, senza bisogno di definire posizioni fissate per variabili.
- Notate che nella sintassi di ADE8, queste due istruzioni non hanno parametri, dato che lavorano solo con l'accumulatore. Inoltre SP è inizializzato a 0, quindi la prima push a che indirizzo scrive? 00 - 1 = ... FF! Ovvero lo stack viene messo alla «fine» della memoria.

Istruzioni per le funzioni

- Vediamo quindi quale supporto ci offre ADE8 per gestire le funzioni tramite lo stack:

CALL (82h): SP ← SP - 1
mem[SP] ← PC
PC ← *param*

RET (83h): PC ← mem[SP]
SP ← SP + 1

- La prima istruzione esegue in pratica la push del PC (che è stato già incrementato per puntare all'istruzione successiva) e poi salta all'indirizzo richiesto.
- La seconda istruzione «ritorna» all'ultima posizione da cui è stata chiamata, estraendo il valore presente sullo stack e mettendolo nel PC.
- RET non ha parametri.

Settima versione

```
        ldv     msg1
        call    Display

        ldv     msg2
        call    Display

Fine:   jmp     Fine

msg1:   'Hello ' 0
msg2:   'world!' 0

; Funzione per visualizzare il messaggio contenuto in ACC
Display:
        st      P
Ciclo:  ldi     P
        cmpv    0
        je      DisplayFine
        st      0d1h
        ld      P
        addv    1
        st      P
        jmp     Ciclo
DisplayFine:
        ret
P:      0
```

Chiamo la funzione Display: non devo più preoccuparmi della gestione del ritorno.

Per ritornare a chi mi ha chiamato, lascio fare all'istruzione ret, che troverà l'indirizzo sullo stack.

Più parametri

- Come la mettiamo con questo mini processore, se abbiamo bisogno di creare una funzione con più di un parametro?
- Per ora abbiamo utilizzato ACC, ma come facciamo con un secondo parametro? E con un terzo?
- Anche qui ci viene in aiuto lo stack. Infatti sullo stack possiamo appoggiare tutti i dati che vogliamo, estraendoli poi con l'istruzione POP.
- In una funzione però questo potrebbe rivelarsi scomodo, dato che dovremmo estrarre l'indirizzo di ritorno, appoggiarlo su una variabile e poi rimetterlo nello stack.
- La soluzione viene nuovamente fornita da un supporto hardware, che ci consente un nuovo tipo di indirizzamento indiretto: relativo rispetto allo stack pointer.

Accesso indiretto allo stack

- Vediamo quindi quale supporto ci offre ADE8 per l'accesso indiretto allo stack:

LDs (03h): ACC ← mem[SP + *param*]

STs (13h): mem[SP + *param*] ← ACC

- La prima istruzione legge il contenuto a partire dalla posizione dello stack più il valore passato come parametro.
- La seconda istruzione al contrario scrive in quella posizione.

- Se quindi prima di chiamare una funzione eseguiamo una PUSH, nella funzione possiamo leggere il dato con LDs 1, perché a SP+0 c'è l'indirizzo di ritorno (salvato dalla CALL), mentre a SP+1 troviamo il dato che noi abbiamo inserito.
- Bisogna ricordarsi poi di estrarre i dati dallo stack al ritorno.

Raddoppiare un numero

- Realizziamo, per capire il meccanismo, una funzione che restituisce nell'accumulatore il doppio di un numero. Per ottenere questo risultato basta sommare il numero a se stesso.
- Nella prima versione passiamo il parametro sull'accumulatore:

```
        ld      num
        call    Doppio
        st      ris
Fine:   jmp     Fine

num:    17
ris:    0

; Funzione che raddoppia l'ACC
Doppio:
        st      tmp
        add     tmp
        ret
tmp:    0
```

Metto in ACC il valore da raddoppiare e chiamo la funzione.

Salvo il valore in una variabile temporanea. Sommo ad ACC quel valore.

Raddoppiare un numero

- Nella seconda versione passiamo il parametro sullo stack, ritornando sempre il risultato nell'accumulatore:

```
        ld      num
        push
        call    Doppio
        st      ris
        pop
Fine:   jmp     Fine

num:    17
ris:    0

; Funzione che raddoppia il parametro passato sullo stack
Doppio:
        lds     1
        st      tmp
        add     tmp
        ret
tmp:    0
```

Carico il valore e lo metto sullo stack.

Devo ricordarmi di togliere il valore dallo stack, ma prima salvo il risultato che era nell'accumulatore.

Estraggo il valore dallo stack. Poi procedo come prima.

Lavorare con lo stack

- Osservando con attenzione la funzione Doppio, notiamo che leggiamo dallo stack il parametro, lo salviamo in memoria (tmp) e poi sommiamo all'accumulatore il dato salvato.
- Ma lo stack **è in memoria**! Perché è necessario copiare nuovamente il dato?
- Il problema è che l'istruzione ADD richiede l'indirizzo assoluto del dato da utilizzare e, ovviamente, non l'abbiamo.
- Per rendere completo quindi questo processore, si rende necessario introdurre una versione delle varie istruzioni in grado di accedere ai dati in modo indiretto tramite lo stack pointer.
- In pratica quello che abbiamo fatto per la LD e la ST deve essere replicato anche con le altre istruzioni.

Istruzioni che lavorano con lo stack

- Istruzioni aritmetiche e logiche con dati sullo stack

ANDs (23h):	ACC ← ACC **and** mem[SP + *param*]	(aggiorna i FLAG)
ORs (33h):	ACC ← ACC **or** mem[SP + *param*]	(aggiorna i FLAG)
ADDs (43h):	ACC ← ACC **+** mem[SP + *param*]	(aggiorna i FLAG)
SUBs (53h):	ACC ← ACC **-** mem[SP + *param*]	(aggiorna i FLAG)
CMPs (63h):	ACC **-** mem[SP + *param*]	(aggiorna i FLAG)

- Queste istruzioni funzionano esattamente come le corrispondenti senza la «s», ma il dato utilizzato viene estratto dallo stack e il parametro indica di quanto spostarsi rispetto allo stack pointer.

Raddoppiare un numero

- In questa versione utilizziamo direttamente il dato sullo stack, in modo da non aver più bisogno di copie temporanee.

```
        ld      num
        push
        call    Doppio
        st      ris
        pop
Fine:   jmp     Fine

num:    17
ris:    0

; Funzione che raddoppia il parametro passato sullo stack
Doppio:
        lds     1
        adds    1
        ret
```

Estraggo il valore dallo stack e lo metto sull'accumulatore. Poi sommo all'accumulatore lo stesso valore per raddoppiarlo: niente variabili temporanee.