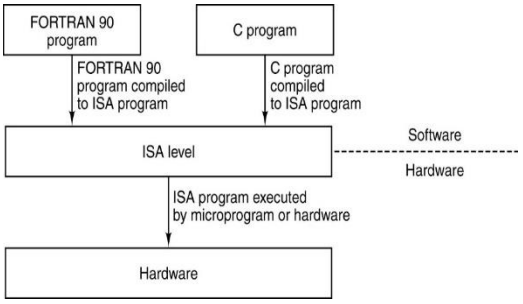
ISA

Il livello ISA ha una rilevanza particolare che lo rende importante per i progettisti di sistemi infatti costituisce l'interfaccia tra il software e l'hardware. Dato che il livello ISA è l'interfaccia tra hardware e software dovrebbe soddisfare sia i progettisti di hardware sia quelli di software.



Tale livello (ISA) si trova tra quello della microarchitettura e il sistema operativo.

L'approccio prediletto dalla quasi totalità dei progettisti di sistemi è quello di partire da vari linguaggi di alto livello per poi tradurli in una forma intermedia comune, il livello ISA, e quindi costruire l'hardware in grado di eseguire direttamente i programmi di alto livello ISA. Tale livello è l'interfaccia tra programmatore e macchina poiché è il linguaggio che entrambi possono comprendere.

Una caratteristica fondamentale del livello ISA è la **retrocompatibilità** cioè bisogna chiedersi "il nuovo progetto del livello ISA è compatibile con il predecessore?".

Che cosa rende un ISA un buon''ISA''?

Ci sono 2 fattori. In primo luogo un buon ISA dovrebbe definire un insieme d'istruzioni che può essere implementato efficientemente da tecnologie presenti e future, il che risulta in progetti duraturi e dunque economicamente vantaggiosi. Il secondo luogo, un buon ISA dovrebbe favorire una compilazione del codice ''pulita''.

Proprietà del livello ISA

Il codice di livello ISA è l'output di un compilatore. Al fine di produrre il codice di livello ISA, il progettista del compilatore deve conoscere il modello di memoria, quali registri ci sono, quali sono i tipi di dati e d'istruzioni disponibili, ecc. L'insieme di tutte queste informazioni definisce il livello ISA. Un'altra proprietà importante è che la maggior parte dei processori è dotata di almeno due modalità di esecuzioni. La **modalità kernel** serve a eseguire il sistema operativo e permette l'esecuzione di tutte le istruzioni. La **modalità utente** ha lo scopo di eseguire i programmi applicativi e non consente l'esecuzione di certe istruzioni come ad esempio quelle che manipolano direttamente la cache.

Modelli di memoria

Tutti i computer suddividono la memoria in celle indirizzate in modo consecutivo. Al momento la dimensione più comune delle celle è di 8 bit. Una cella di 8 bit si chiama byte (o ottetto). La ragione per prediligere i byte è che i caratteri nella tabella ASCII occupano 7 bit, cosi che un carattere ASCII più un bit di parità riempiono esattamente un byte.

In genere i byte vengono raggruppati in parole di 4 byte (32 bit) o di 8 byte (64 bit). Molte architetture esigono che le parole siano allineate lungo le loro estremità e così, per esempio, una parola di 4 byte può cominciare agli indirizzi 0, 4, 8 ma non agli indirizzi 1 ,2,3 ecc. Spesso

l'allineamento è richiesto perché in tal modo la memoria riesce a funzionare in modo più efficiente. La capacità di leggere parole che cominciano a indirizzi arbitrari richiede nel chip funzionalità logiche supplementari, il che lo rende più grande e più costoso. Ma d'altro canto queste funzionalità sono necessarie poiché interessano ai clienti.

Gran parte dei processori a livello ISA dispone di un solo spazio lineare degli indirizzi, che si estende dall'indirizzo 0 fino a un certo massimo, generalmente 2^32 o 2^64 byte. Esistono tuttavia macchine dispongono di spazi degli indirizzi separati per le istruzioni e per i dati, di modo che il fetch di un'istruzione all'indirizzo 8 proviene da un diverso spazio degli rispetto al fetch di un dato all'indirizzo 8. La separazione dello spazio delle istruzioni da quello dei dati rende inoltre più difficili gli attacchi dei malware, perché il software "maligno" non può accedere al programma.

Un'altra caratteristica importante del livello ISA è la "semantica" delle operazioni di accesso alla memoria.

La sicurezza della esecuzione corretta di una LOAD che segue una STORE: problemi nel caso di istruzioni non eseguite in sequenza; tre possibili approcci (via via più restrittivi):

*Lasciare al programmatore (al compilatore) la responsabilità di assicurare l'effettiva memorizzazione dei dati tramite una SYNC.

*Realizzare in hardware la verifica automatica della presenza di RAW (Read After Write) o di WAR (Write After Read) nell'esecuzione di LOAD/STORE.

*Sequenzializzare tutte le LOAD/STORE.

Registri

Tutti i calcolatori dispongono di registri su cui lavora la ALU. Il loro compito è il contenimento dei risultati temporanei, il controllo dell'esecuzione del programma o altro. I registri del livello ISA possono essere divisi in due categorie: **registri specializzati** e **registri d'uso generale**. I primi comprendono il program counter, il puntatore alla cima dello stack e altri registri con funzioni specifiche. I registri d'uso generale sono destinati invece a contenere le variabili locali più importanti e i risultati parziali del calcolo. La loro funzione principale è di consentire un accesso rapido a dati usati in modo rincorrente.

Un registro particolare presente in tutte le architetture è il **registro di flag** o **PSW** (Program Stauts Word) che contiene i codici di condizione che sono asseriti dalla ALU:

N – a 1 quando il risultato è negativo

Z – a 1 quando il risultato è zero

V – a 1 quando il risultato causa overflow

C – a 1 quando il risultato produce un riporto

I codici di condizione sono importanti perché sono utilizzati dalle istruzioni di confronto e di salto condizionato.

Tipi di dato

Una data architettura può gestire direttamente diversi tipi di dati (nel senso che hanno istruzioni specifiche per trattare tali tipi di dati). Tipicamente tutte le ISA gestiscono gli interi (normalmente rappresentati in complemento a 2), ma possono esserci istruzioni specifiche dedicate alle operazioni su tipi non interi Floating Point, su tipi booleani (cioè sui singoli bit di un byte o di una parola), su (stringhe di) caratteri (es. copia di una stringa di caratteri da un punto ad un altro in memoria). Infine si devono poter esprimere indirizzi, per cui è previsto l'uso di puntatori (vedere più avanti modalità di indirizzamento). Possiamo avere anche registri dedicati a tipi di dati diversi, tipicamente vi sono insiemi di registri distinti per i dati INTERI per i dati FLOATING POINT.

Booleani

A cosa servono i tipi booleani?

In teoria un singolo bit e sufficiente a rappresentare i valori possibili di un dato booleano; in pratica vengono spesso utilizzati un intero byte o addirittura un'intera parola. La ragione di questa scelta poco efficiente è l'impossibilità (e la non-convenienza) di indirizzare il singolo bit; per convenzione, in questi casi, 0 significa FALSO, mentre qualunque altro valore viene interpretato come VERO.

Puntatori

I puntatori sono di fatto indirizzi di memoria. La possibilità di usare puntatori nelle istruzioni permette di fare riferimento a locazioni di memoria diverse con la stessa istruzione (utile per esempio per scorrere un vettore in un ciclo).

Istruzioni

Infine, l'aspetto più importante del livello ISA è rappresentato dall'insieme di istruzioni fornite:

LOAD e STORE // trasferimento dati da memoria a registro e viceversa

MOVE // copia di dati da un registro ad un altro

ARITHMETIC // operazioni aritmetico-logiche

BOOLEAN // operazioni su variabili booleane (codificate su un byte o parola: 0 = FALSO, non 0 =VERO), oppure raggruppate in parole che giocano il ruolo di "mappe di bit".

CONFRONTO E SALTO // per controllare il flusso di esecuzione delle istruzioni: il confronto imposta i bit nella PSW, il salto viene deciso in base a tali bit.

Formato istruzioni

La codifica binaria delle istruzioni è formata da un **codice operativo** a cui si possono aggiungere uno o più campi per indicare da dove prelevare i dati e dove memorizzare i risultati (**i cosi' detti *operandi***). Il problema generale della indicazione del luogo da cui prelevare o in cui memorizzare i risultati è chiamato **indirizzamento** (cioè dove è come prelevare i dati). Per quanto riguarda il numero di indirizzi specificati in una istruzione le possibili alternative sono: zero indirizzi, un indirizzo, due indirizzi, tre indirizzi.

Un problema importante nella scelta del formato delle istruzioni è la loro lunghezza (quanti bit sono necessari per specificarle). C'è bisogno quindi di un compromesso tra espressività (quindi la loro lunghezza) e velocità di prelievo delle istruzioni.

L'efficienza di una architettura a livello ISA dipende fortemente dal tipo di tecnologia con cui è realizzata e di cui si deve tenere conto nelle **scelte per il formato delle istruzioni.**

Dimensione dell'istruzione (a lunghezza fissa vs. variabile).

*Istruzioni corte sono preferibili a causa del minor spazio di memoria richiesto e della capacità di trasferimento CPU-Memoria che esse richiedono.

*Istruzioni corte riducono la possibilità che la CPU rimanga in attesa del fetch di nuove istruzioni; ma, istruzioni corte sono (in generale) più difficili da decodificare.

Dimensione del campo destinato al codice operativo.

Deve essere possibile rappresentare sufficienti codici operativi diversi (ed eventualmente poterne aggiungere nel tempo).

Dimensione dei campi di indirizzamento (e numero di campi).

Dipende dalla dimensione della memoria (e quindi dall'unità di indirizzamento della memoria), ma anche dal tipo di indirizzamento.

Scelta del formato

Compromesso tra lunghezza dei codici operativi ed ampiezza dei campi di indirizzamento.

Codici operativi corti (e regolari) danno luogo a 1) una decodifica veloce, 2) grande spazio di indirizzamento, 3) Conviene assegnare codici operativi brevi (veloci da decodificare) a istruzioni frequenti o a istruzioni che richiedono grande spazio di indirizzamento e assegnare codici operativi lunghi a istruzioni eseguite raramente o a istruzioni che non richiedono grande spazio di indirizzamento. Istruzioni di lunghezza fissa sono più efficienti da gestire di quelle a lunghezza variabile.

La complessità del formato di un'istruzione dipende fortemente, oltre che dalla scelta della rappresentazione del codice operativo, anche dalla **modalità di indirizzamento** cioè la modalità con cui si specifica dove si trovano gli operandi (chiamiamo operandi sia gli operandi veri e propri sia la destinazione del risultato). Per ridurre la lunghezza dell'istruzione è bene, per quanto possibile, fare riferimento ai registri:

1)portando i dati di uso più frequente in registri di lavoro

2)utilizzando dei registri per "muoversi" all'interno della memoria (registri che contengono indirizzi di memoria, cioè *puntatori*).

In alternativa è possibile ridurre la lunghezza dell'istruzione anche facendo uso di operandi impliciti:

1)il medesimo operando utilizzato in input e in output (2 indirizzi)

2)uso di registri specializzati (per esempio l'accumulatore (1 indirizzo)

3)uso della stack (zero indirizzi)

Codice operativo espansibile

Si consideri un'istruzione lunga (n+k) bit, composta da un opcode di k bit e da un solo indirizzo di n bit. Un' istruzione fatta così consente 2^k operazioni diverse e permette d'indirizzare 2^n celle di memoria. In alternativa gli stessi (n+k) bit potrebbero essere spezzati in (k-1) bit di opcode e (n+1) bit d'indirizzo, dimezzando il numero d'istruzioni, ma al contempo raddoppiando la dimensione della memoria raggiungibile.

Introduciamo ora il concetto di **codice operativo espansibile**, che descriviamo mediante un esempio. Si consideri una macchina in cui le istruzioni sono lunghe 16 bit e gli indirizzi 4 bit. Questa situazione potrebbe essere ragionevole per una macchina che ha 16 registri (dunque indirizzabile con 4 bit), sufficiente per lo svolgimento di tutte le operazioni aritmetiche. Un'architettura possibile sarebbe quella di comporre ogni istruzione con 4 bit di opcode e tre indirizzi, per un totale di 16 bit.

Indirizzamento

Molte istruzioni contengono operandi e si pone il problema di come specificarne la posizione. L'indirizzamento è l'argomento che tratta queste problematiche che ora affrontiamo.

Indirizzamento immediato

L'operando è contenuto nell'istruzione **(BIPUSH 10)** ***IJVM***

Indirizzamento diretto

L'operando è all'indirizzo di memoria specificato nell'istruzione. Usato per accedere a variabili globali.

LOAD R1, 0x102930 **semantica:** R1 ← m[0x102930]

Indirizzamento a registro (modalità a registro)

L'indirizzamento a registro è concettualmente analogo all'indirizzamento diretto, ma specifica un registro invece di una locazione di memoria.

Istruzione **ADD**

Indirizzamento a registro indiretto

In questa modalità l'operando in esame proviene o è destinato alla memoria, ma il suo indirizzo non è incorporato nell'istruzione. Un grande vantaggio dell'indirizzamento a registro indiretto è che può referenziare la memoria senza dover necessariamente incorporare un intero indirizzo di memoria all'interno dell'istruzione.

Istruzione **MOVE**

Indirizzamento indicizzato

L'indirizzamento alla memoria che si ottiene specificando un registro (in via esplicita o implicita), più uno spiazzamento costante, si dice **indirizzamento indicizzato.**

Indirizzamento indicizzato esteso

Alcune macchine dispongono della cosiddetta modalità d'indirizzamento indicizzato esteso, in cui l'indirizzo di memoria è calcolato sommando tra loro il contenuto di due registri più un offset (opzionale). Un registro funge da base e l'altro da indice.

Indirizzamento a stack

Abbiamo visto che è consigliabile rendere le istruzioni macchina quanto più corte possibili. Il limite alla riduzione della lunghezza degli indirizzi equivale a non averne per nulla. Le istruzioni senza indirizzi, come l'istruzione IADD è possibile solo in associazione con uno stack.

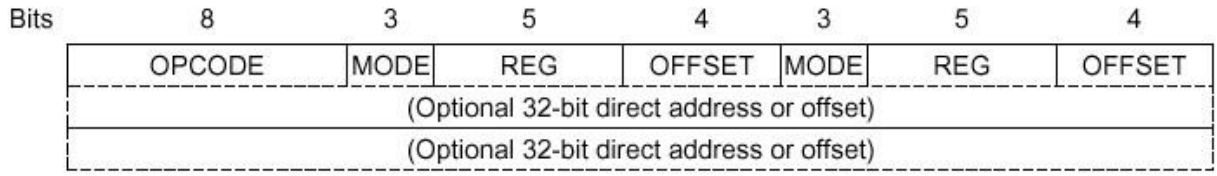
In matematica è tradizione, indicare l'operatore in mezzo agli operandi (x+y), invece che dopo gli operandi (xy+). La forma con l'operatore in mezzo è chiamata **postfissa**, mentre la forma con l'operatore dopo gli operandi si chiama **postfissa** o **notazione polacca inversa**. Le espressioni in notazione polacca hanno un vantaggio sono facilmente eseguibili da una architettura a stack.

(8+2x5)/(1+3x2-4)

8 2 5 x + 1 3 2 x + 4 - /

Step	Remaining string	Instruction	Stack
1	8 2 5 × + 1 3 2 × + 4 – /	BIPUSH 8	8
2	2 5 × + 1 3 2 × + 4 – /	BIPUSH 2	8, 2
3	5 × + 1 3 2 × + 4 – /	BIPUSH 5	8, 2, 5
4	× + 1 3 2 × + 4 – /	IMUL	8, 10
5	+ 1 3 2 × + 4 – /	IADD	18
6	1 3 2 × + 4 – /	BIPUSH 1	18, 1
7	3 2 × + 4 – /	BIPUSH 3	18, 1, 3
8	2 × + 4 – /	BIPUSH 2	18, 1, 3, 2
9	× + 4 – /	IMUL	18, 1, 6
10	+ 4 – /	IADD	18, 7
11	4 – /	BIPUSH 4	18, 7, 4
12	– /	ISUB	18, 3
13	/	IDIV	6

Se tutte le modalità di indirizzamento si possono applicare a tutti gli operandi indipendentemente dal codice operativo utilizzato si dice che c'è **ORTOGONALITA' tra opcode** (cioè tipo di operazione da compiere) e modalità di indirizzamento (cioè come si specifica dove si trovano gli operandi). Questa caratteristica facilita molto il compito del compilatore.



Si consideri ora il progetto di una macchina a 2 indirizzi, che può specificare parole di memoria per entrambi gli operandi. Una tale macchina è in grado di sommare una parola di memoria a un registro, oppure sommare tra di loro 2 parole di memoria. In questo schema i bit per il codice operativo sono 8, e abbiamo a disposizione 12 bit per specificare la sorgente e altri 12 per la destinazione. Ciascuna modalità ha 3 bit per la modalità d'indirizzamento, 5 per i registri e 4 per l'offset. Con 3 bit possiamo gestire tutte le modalità, immediata, diretta, a registro, a registro indiretto, indicizzata, a stack e avremmo spazio per altre 2 modalità.

Modalità d'indirizzamento per istruzioni di salto

Anche le istruzioni di salto necessitano di modalità di indirizzamento per specificare l'indirizzo di destinazione. Le modalità affrontate finora funzionano anche per i salti. Per esempio l'indirizzamento diretto è di certo un'opzione, secondo cui l'indirizzo di destinazione viene semplicemente riportato per intero all'interno dell'istruzione.

Input e Output

Le operazioni di input e output rappresentano una variazione del flusso. Queste variazioni del flusso di controllo devono essere previste dal programma (salti, chiamate a procedura) e possono essere dovute ad eventi eccezionali prodotti dal programma (trap) o dovute ad eventi esterni al programma (interrupt). Ogni dispositivo è collegato al bus di sistema tramite un'interfaccia dove risiedono alcuni registri (di controllo, per permettere la sincronizzazione CPU-Dispositivo, buffer per contenere i dati in ingresso/uscita). Questa interfaccia è chiamata **controller**.

La CPU colloquia con il controller scrivendo/leggendo i suoi registri in uno dei seguenti modi:

1.utilizzando apposite istruzioni (IN / OUT) che avranno come parametro un "indirizzo" che identifica un particolare controller.

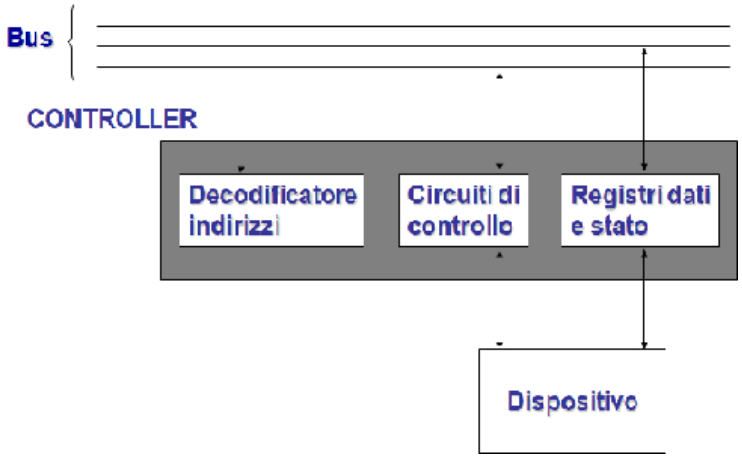
2.utilizzando normali istruzioni LOAD/STORE a indirizzi associati ai registri del controller: questa tecnica si chiama Memory Mapped I/O.

Nessun raggruppamento delle istruzioni manifesta la stessa variabilità tra macchine diverse come le istruzioni di I/O. Attualmente i calcolatori usano tre schemi diversi di I/O:

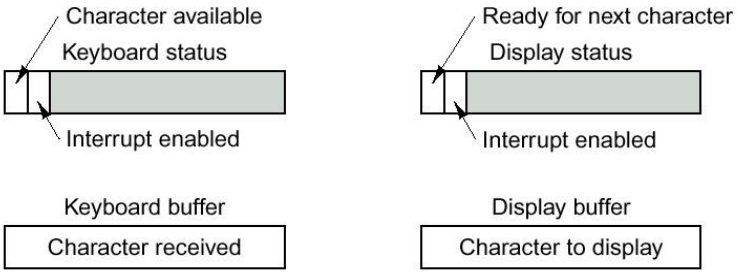
1. I/O programmato con attesa attiva
2. I/O interrupt driven (cioè innescato dagli interrupt)
3. I/O con DMA.

Il metodo di I/O più semplice possibile è quello **programmato** impiegato di solito nei microprocessori di fascia bassa come, per esempio, i sistemi integrati oppure i sistemi che devono rispondere agli stimoli esterni (sistemi in tempo reale).

Memory mapped I/O: il controller contiene un decodificatore di indirizzi che riconosce gli indirizzi assegnati ad esso (quando viene rilevata una richiesta di lettura/scrittura riferita a tali indirizzi restituisce o recepisce i dati nei suoi registri).



Esempio di I/O programmato con approccio busy waiting cioè con attesa attiva.



Consideriamo un semplice terminale con 4 registri di 1 byte. Due registri sono utilizzati per lo stato e i dati in input, mentre gli altri due sono usati per lo stato e i dati in output. Ogni registro ha un indirizzo unico. Se l'I/O è mappato in memoria, i 4 registri fanno parte dello spazio degli indirizzi della memoria del computer e possono essere letti e scritti mediante le istruzioni ordinarie. In caso contrario ci sono istruzioni speciali di I/O, che chiamiamo IN e OUT, per la lettura e la scrittura dei registri. In entrambi i casi l'I/O avviene tramite il trasferimento dati del loro stato tra la CPU e questi registri.

Il registro dello stato della tastiera utilizza 2 soli bit degli 8 disponibili: il bit più significativo è posto a 1 via hardware ogniqualvolta arriva un carattere; questo evento solleva un interrupt se e solo se il sesto bit era stato asserito precedentemente via software. Nel caso dell'I/O programmato la CPU aspetta dati in ingresso effettuando un ciclo serrato e ripetuto di letture sul registro di stato della tastiera, aspettando che il bit 7 assuma il valore 1. Non appena ciò si verifica, il software legge il carattere del registro buffer della tastiera. La lettura del registro dati della tastiera causa l'azzeramento del **bit carattere disponibile.**
Per visualizzare un carattere sullo schermo il software per prima cosa legge il registro di stato dello schermo per assicurarsi che il bit PRONTO valga 1 e in caso negativo si ripete finché il bit non viene asserito a indicare che il dispositivo è pronto ad accettare un carattere. Non appena il terminale è pronto, il software scrive un carattere nel registro buffer dello schermo, il che provoca la sua trasmissione allo schermo e anche l'azzeramento del bit PRONTO nel registro di stato dello schermo. Dopo la visualizzazione del carattere, il controllore imposta automaticamente a 1 il bit PRONTO non appena il dispositivo è preparato a trattare il carattere successivo.

I/O programmato con attesa attiva nell'emulatore di MIC-1

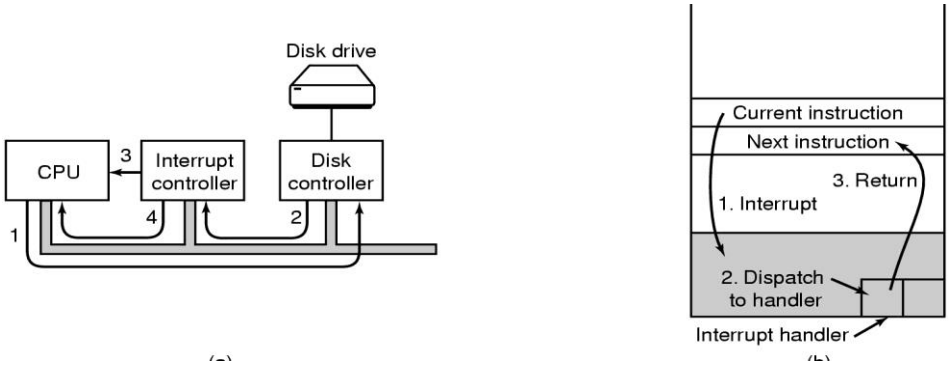
Anche nell'emulatore di MIC-1 usiamo questo approccio per l'I/O. Nel linguaggio abbiamo due istruzioni particolari IN e OUT per effettuare l'input (da tastiera) e l'output (su display). Nel nostro caso la IN legge sia il bit di stato sia il carattere: se il bit di stato è 0 la IN restituisce sul top-of-stack e dobbiamo ripetere la lettura; se la IN restituisce un valore diverso da zero sullo stack allora questo è il codice del tasto appena premuto.

```
input:  IN
        DUP
        IFEQ nessuncarattere
        ISTORE c
        …
nessuncarattere:
        POP
        GOTO input
```

La OUT è più semplice e in sostanza si assume che il dispositivo (display) sia sempre pronto a ricevere un carattere per visualizzarlo sulla console (si omette il test sullo stato assumendo che in ogni caso la OUT conclude prima che possa essere eseguita una seconda istruzione dello stesso tipo.

Dato che CPU e dispositivi di I/O hanno velocità molto diverse, è opportuno non tenere la CPU bloccata in un ciclo di controllo dei bit di stato in attesa che termini l'operazione richiesta. Se si potesse evitare il busy waiting, mentre i dispositivi di I/O lavorano, la CPU potrebbe essere impiegata in compiti più utili (per esempio eseguire un altro programma). Infatti lo svantaggio principale dell'I/O programmato è che la CPU passa gran parte del suo tempo in un ciclo serrato in cui attende che il dispositivo risulti pronto.

SEQUENZA DI GESTIONE DELL'INTERRUPT



1. Il driver dice al controller cosa fare
2. Il controller al termine dell'operazione invia un interrupt
3. L'interrupt controller se libero inoltra il segnale alla CPU
4. L'interrupt controller indica alla CPU l'ID di chi ha generato l'interrupt

Quando riceve INT la CPU termina l'istruzione corrente, salva lo stato e passa il controllo all'interrupt handler terminata la gestione dell'interrupt riprende l'esecuzione.