Settimana scorsa abbiamo visto un esempio di processore, il PDP-11 e abbiamo iniziato a vedere come vengono codificate le istruzioni. Ossia, c’è una struttura base, costituita dal codice operativo, e i due operandi che occupano i bit rimanenti. Si ricorda a tal proposito che nella Cpu sono contenuti 8 registri, 6 di uso generale e 2 che sono SP e PC.

Il codice operativo è a espansione, oltre a 4 bit ne può occupare 7 (se i primi 4 sono X111) o 10 (se i primi 4 bit sono 0000). L’istruzione più significativa è l’istruzione MOVE (che interpreta i due operandi come sorgente e destinazione).

Abbiamo visto due metodi di indirizzamento particolari: il modo autoincremento e il modo indicizzato.

Per convenzione, l’indirizzo 0 sta “in cima” alla RAM e gli indirizzi aumentando scendendo.

Durante la fase di fetch viene inserita nell’IR l’istruzione presente nella cella puntata dal PC, poi il valore di PC incrementa di 1, poi avviene la decodifica dell’istruzione e infine essa viene eseguita.

Il modo più intelligente di sfruttare la struttura PDP-11 è stato quello di suddividere la RAM in una sezione iniziale dedicata al codice, poi una seconda parte dedicata ai dati statici/globali e in fondo una ripartizione dei dati, chiamata Stack (nello Stack gli indirizzi sono i più alti).

Lo Stack serve per poter realizzare una serie di funzionalità molto importanti: l’estensione procedurale, il meccanismo delle interruzioni (interrupt) e il meccanismo delle trap. Di base lo Stack è una struttura dati di tipo dinamico, su cui si possono aggiungere o togliere informazioni durante l’esecuzione.

I registri del processore che puntano allo Stack (R6) vengono inizializzati in modo autodecremento.

R6 viene quindi prima decrementato (R6 <- R6 -2 per esempio) e dopo si accede alla memoria RAM usando il registro R6. ES:

MOVE R7, R6 <- R6-2, RAM[R6] -> Questa operazione è detta PUSH del contenuto del PC(R7) nello Stack.

MOVE R6, R6 <- R6-2, RAM[R6]

Per inizializzare il registro 6 in modo che sia il valore più grande possibile si sfrutta una condizione di Overflow (lo si fa partire dal valore 0, così appena si toglie il primo byte/word (1/2) esso punta alla cella più grande possibile).

Una volta che si è salvato il contenuto del registro nello Stack si può modificare il suo contenuto come ci pare consci del fatto che il valore precedente è recuperabile dallo Stack (sempre con delle istruzioni di tipo MOVE, ma scambiando gli operandi). Volendo si può, per fare ciò, utilizzare la funzione inversa (autoincremento) per l’estrazione degli elementi dallo Stack -> questa operazione viene chiamata POP.

MOVE RAM[R6], R6 <- R6+2, R6

MOVE RAM[R6], R6 <- R6+2, R7

(Nota: la porzione dello Stack che contiene dati allocati dal nostro programma sono le celle di indirizzo maggiore o uguale del nostro StackPointer).

Tutto ciò serve per poter interrompere un programma quando questo avvia l’esecuzione di un’altra procedura/funzione e per poterlo riprendere al termine di tale esecuzione.

Questa operazione può essere fatta quante volte si vuole a patto di avere a disposizione sufficiente memoria. Quando si chiamano funzioni recursive questo sistema ci permette di continuare a impilare strati nello Stack fintanto che però non raggiungiamo l’area di memoria dedicata ai dati + codice. Questa struttura è comunque decisamente flessibile, poiché la quantità di memoria è l’unico vincolo che impone.

Esempio:

main() {

a = 0; //indirizzo 15

b = a + 2; //16

// Queste sono poche istruzioni ad alto livello traducibili in un certo numero di istruzioni in codice macchina

c = sqrt(2); //17

d = 5; //18

// Per la chiamata di funzione viene interrotta l’esecuzione del programma e si parte ad eseguire la prima istruzione della procedura sqrt(). Al termine dell’esecuzione di sqrt il PC deve essere ripristinato per poter eseguire l’istruzione d = 5; Ecco perché prima si esegue l’operazione di PUSH per salvare il valore del PC nello Stack prima di eseguire la funzione.

}

Quando si fa il fetch dell’istruzione 17, il PC viene incrementato a 18. Quindi si esegue il PUSH:

MOVE R7, R6 <- R6-2, RAM[R6]

Dopodiché si modifica il valore del PC per poter eseguire sqrt:

MOVE 152, ,R7

Quando termina l’esecuzione della funzione si esegue il POP per ripristinare il PC:

MOVE RAM[R6],R6<-R6+2, R7

Il motivo per cui è possibile realizzare macchine virtuali è perché si possono avere processori “virtuali” tramite il salvataggio di registri non fisici nello Stack.

L’esecuzione di un programma è una cosa autocontenuta per quanto riguarda lo Stack: non c’è interazione con l’utente. Per questo si introduce il meccanismo delle interruzioni (per analogia funzionano come il campanello di casa -> se lo sentiamo suonare interrompiamo l’attività in corso e andiamo a controllare chi è alla porta). Quindi per poter rispondere a una interruzione è necessario congelare l’esecuzione del programma/della procedura -> il processore interrompe l’esecuzione, salva i registri nello Stack e passa ad eseguire il gestore delle interruzioni: esso è ciò che interroga i dispositivi per controllare chi e per quale motivo ha causato una interruzione (input da tastiera ecc.).

I trap sono invece le segnalazioni di errore: Quando lo SP arriva a un punto che supera la zona di memoria massima dedicata allo Stack avviene una segnalazione di errore. Analogamente alle interruzioni viene congelato il programma e mandata una specie di interruzione ma il chi l’ha causata e il per cosa non sono normali: infatti è chiamata dal gestore di errori (un dispositivo hardware che si accorge di ciò che è successo e lo segnala). La Trap è quindi un modo per cercare di correggere la situazione: la soluzione più tipica è l’interruzione del programma in corso con segnalazione della causa all’utente.

In particolare, nell’architettura PDP-11 il registro 6 viene utilizzato come Stack Pointer per le Trap, mentre all’utente per gli usi applicativi sono disponibili come Stack pointer tutti gli altri registri (al di fuori del R7 obv). Si può quindi considerare di suddividere lo Stack in sezioni, ognuna delle quali gestita da un registro-Stack Pointer diverso.