Oggi vediamo il funzionamento del simulatore del processore Amber creato dal prof. La possibilità in più rispetto a prima è che c’è una finestra in cui si può inserire le istruzioni a livello assembler. C’è un menu a tendina che fornisce le istruzioni possibili. Quando è costruita l’istruzione coi suoi parametri basta premere sull’encode e si ottiene la versione compilata di quell’istruzione. Ci sono due modalità di funzionamento del simulatore: quella di simulazione di esecuzione e quella di programmazione. Per attivare questa seconda modalità di scrittura si va sul menu a sinistra e si cambia 1 tick exec con “allow write”. Sono state aggiunte le label, nomi simbolici associabili a indirizzi della ram, utilizzati per i branch. Per creare la Label si inserisce l’etichetta e il numero della cella (multiplo di 16 e che verrà ridotto di 8 in automatico) e poi cliccare su SET. Fatto ciò queste etichette possono essere usate per il codice assembler. Per una questione di semplicità le etichette sono rappresentate sempre con un $ all’inizio (quindi l’etichetta ciao va inserita come $ciao).

Un’altra differenza rispetto alla versione precedente è che ci sono dei registri memory mapped per i dispositivi di input-output.

Ci sono già due programmi eseguibili, Euclide0.ram (che funziona analogamente a quello di prima). I commenti adesso sono iniziati dal carattere % anziché #.

Un altro esempio di programma inserito è il Divisione.ram (il programma visto la volta scorsa).

Vediamo quindi degli esempi legati alla gestione dei dispositivi di ingresso e uscita: ci possono servire dei programmi di conversione (in ingresso e in uscita arrivano caratteri ascii ma in memoria sono salvati numeri in binario).

Cosa dobbiamo fare per codificare un numero dalla rappresentazione in ascii al binario? Supponiamo che l’utente scriva sulla tastiera la sequenza di caratteri 123. In tal caso questi caratteri vengono letti uno alla volta, bisognerà fare delle somme e delle moltiplicazioni:

Si parte col valore iniziale v = 0, trovo il carattere 1 e lo sommo a v, v = 1. Trovo poi il carattere 2, a questo punto devo moltiplicare per 10 e poi sommare 2, v = 10\*v+2, lo stesso si ripete col 3: v = 10\*v + 3.

Le somme non ci danno problemi, abbiamo l’istruzione macchina che usa la ALU apposta per quello, infatti basta chiamare ADD R0, R0, R1 (R1=registro dove è stato salvato il contenuto del registro mappato in memoria della tastiera).

Il codice del programma sarà quindi una cosa di questo tipo:

MOV R0, #0

Leggi in R1 //nota: non è stata ancora implementata la lettura

SUB R1, R1, #’0’ (metti in R1, R1-codice ascii di ‘0’) //si potrebbero mettere dei controlli per verificare che otteniamo un numero tra 0 e 9, ma non lo faremo

//10 lo possiamo scomporre come 8+2 o come (4+1)\*2, quindi possiamo fare la moltiplicazione così:

ADD R0, R0, R0 lsl #2 //Somma R0 a R0\*4, otteniamo così la moltiplicazione per 5

MOV R0, R0 lsl #1 //moltiplichiamo R0 per 2, ora in R0 c’è il valore numerico di R1 \* 10

Operazione opposta, abbiamo un numero in binario e vogliamo convertirlo in codifica Ascii.

Per esempio, se R0 contiene il valore 456 vogliamo stampare i caratteri 4, 5 e 6.

Per realizzare ciò bisogna prendere il numero v = 456 e dividerlo per 10, stampando il resto (in C sono due istruzioni separate, da una parte ci sarà v % 10 e dall’altra v / 10). Sfortunatamente questo modo di procedere ci stampa i caratteri al contrario. Per poter invertire l’ordine ci sono più alternative diverse: la più semplice di esse è di utilizzare una struttura di tipo stack, in cui man mano che trovo una cifra la inserisco all’interno dello stack, in cui il sistema di uscita è LIFO. Fatto questo posso passare alla stampa.

Potremmo usare una stampa ricorsiva, in C sarebbe tipo:

stampa(int v) {

r = v % 10;

v /= 10;

if(v > 0) stampa(v);

scrivo r + ‘\0’;

}

In questo caso però si usa lo stack del sistema anziché uno stack applicativo. Quali sono i vantaggi e gli svantaggi di questa versione ricorsiva?

La versione non ricorsiva sarebbe

stampa(int v) {

do {

r = v % 10;

push(r);

v /= 10;

} while(v > 0);

While((r = pop()) != -1) {

scrivo r + ‘\0’;

}

}

Nella chiamata ricorsiva avremo dei branch and link, mentre nell’iterativa avremo dei branch (ma per il processore costa lo stesso). Ci può essere però un overhead nella versione ricorsiva nel fatto che bisogna identificare dei registri da utilizzare, che ad ogni chiamata vanno salvati nello stack e ripristinati alla fine. D’altro canto nella funzione iterativa la push e la pop sono l’equivalente di funzioni Load e Store, quindi bisogna vedere quale dei sistemi è più veloce (specie se si considera che il salvataggio dei registri nella versione ricorsiva è fatto con la LoadM e la StoreM).

Proviamo quindi a tradurre in assembler queste funzioni.

Versione ricorsiva.

V -> R0

R -> R3

Stampa(V)

STM [SP], R1, R2, R3, LP

MOV R1, #10

BL divide //al termine della chiamata avremo il resto nel registro 0 e il quoziente nel registro 2.

//ci servono altri registri per il confronto del risultato

CMP R2, #0

MOV R3, R0 //variabile d’appoggio

MOVGT R0, R2 //scambio

BLGT stampa

ADD R0, R3, #’0’

BL putchar

LDM [SP], R1, R2, R3, PC

Versione iterativa.

V -> R0

R -> R3

Stampa(V)

STM [SP], R1, R2, R3, LP

MOV R1, #10

STR #-1, SP!

loop

BL divide

STR R0, SP!

CMP R2, #0

MOVGT R0, R2

BGT loop

pop

LDR R0, SP!

CMP R0, #0

LDMLT [SP], R1, R2, R3, PC

ADD R0, R0, #’0’ (= codice ascii di ‘0’)

BL putchar

B pop

La versione ricorsiva necessita di meno istruzioni, tuttavia ad ogni chiamata ricorsiva viene effettuata la store di 4 registri, contro la store di un singolo registro per ciclo fatto nella versione iterativa (quindi quella istruzione prenderà 4 cicli di clock contro 1), inoltre c’è la ridondanza di alcune operazioni (come l’inserimento di 10 in R1).

La versione ricorsiva è quindi più breve, ma è in generale più costosa da implementare rispetto alla versione iterativa (a livello di spazio occupato e di cicli di clock).

Una considerazione di carattere pratico: Quando ha senso farsi problemi per cicli di clock in più o in meno? Quando il pezzo di codice in questione viene ripetuto per Molte volte (se quel pezzo è ripetuto migliaia di volte diventa notabile).

Dunque il passaggio dalla versione ricorsiva a quella iterativa si ha quando il codice deve essere eseguito parecchie volte e frequentemente, non ha senso quando il codice è fatto per essere usato una sola volta.

Un altro caso in cui ha senso ridurre il numero di cicli di clock è quando si programma un handler di interruzioni o trap, perché ogni ciclo di clock perso è un ciclo ad interruzioni disattivate, il che può essere critico in queste situazioni.