Abbiamo visto gli handler per le IRQ e per le FIRQ, ci mancano quelli per il RESET e per la SWI.

Per quanto riguarda il RESET, il suo handler inizia con un branch nell’indirizzo 0 che porta all’indirizzo 192.

Reset\_h: 192: MOV FP, #16384 //il FP punta all’inizio del segmento per i dati statici

MOV SP, #32768 //Lo SP punta all’inizio dello Stack.

/\*quando il processore opera in modalità FIRQ ha un banco di registri speciali, dall’8 all’11, nei quali ci devono essere dei valori utili per il nostro handler, ossia i buffer di input e di output e gli indirizzi dei registri memory mapped dei dispositivi di IO. Tuttavia noi ci troviamo ancora in modalità Supervisore (11), bisogna pertanto passare alla modalità di funzionamento 1 (FIRQ) per inizializzare i registri \*/

MOV LP, #201326592 //Questo corrisponde ad aver acceso i bit F ed I nel registro di stato, ossia le maschere delle interruzioni

ADD LP, LP, #217 //Il motivo per cui non si è direttamente aggiunto 217 alla costante della MOV è perché il valore così ottenuto non sarebbe rappresentabile nel nostro Shifter Operand. Il numero è dispari perché l’1 indica lo stato FIRQ del PC, il 216 è l’indirizzo della prima istruzione.

STM, SP! FP, SP, LP

212: LDMDB SP!, FP, SP, PC^ //I registri sono salvati nello Stack, e allo stesso tempo si cambia il PC in modo che abbia modo di esecuzione 1 (FIRQ) anziché 3. La Load multipla degli altri registri serve perché il cambio di modalità AVVIENE COME PRIMA COSA, dunque adesso FP, SP sono stati copiati anche nei registri propri della FIRQ. Poiché l’istruzione dopo è comunque la 216, ciò che cambia nel PC sono solo il modo di esecuzione e le maschere delle interruzioni.

216: MOV FP, #tty\_in\_mmr //l’indirizzo del registro memory mapped del dispositivo di ingresso (quello di uscita è questo +4)

MOV R10, #buffers //descrittore dei buffers. L’inizializzazione è fatta una volta sola per le FIRQ, ma non per le IRQ.

/\*Bisogna però inizializzare il descrittore dei buffers e i buffers stessi\*/

MOV R4, #65536 //questo sarà l’indirizzo a partire dal quale allochiamo i nostri buffers

MOV R0, R4

MOV R3, #128 //i buffers saranno di 128 bytes.

MOV R2, #0

STR R0, [R10] //R0 è l’indirizzo del primo elemento del buffer, che metto all’inizio del vettore descrittivo #buffers

STR R3, [R10+4] //La Dimensione va nella cella di memoria successiva

STR R2, [R10+8] //indirizzo di inserzione

STR R2, [R10+12] //indirizzo di estrazione

MOV R1, R3, lsr #2 //in R1 c’è R3/4, poiché le celle di memoria contengono 4 bytes ciascuna

BL memset //questa funzione si aspetta 3 parametri, in R0, R1, R2: R0 = indirizzo di partenza, R1 = numero di parole da inizializzare (ecco il perché del /4) e R2 = il valore con cui inizializzare le celle di memoria indicate

/\*\*\*\*\*{

memset: 354: SUBS R1, R1, #1

STRGE R2, [R0, +R1, lsl #2]

MOVLT PC, LP

B memset

} \*\*\*\*\*\*/

ADD R0, R4, R3 //così in R0 c’è l’indirizzo di partenza del secondo buffer

STR R0, [R10, #+16] //complete il vettore descrittore dei buffers

STR R3, [R10, #+20]

STR R2, [R10, #+24]

STR R2, [R10, #+28]

MOV R1, R3, lsr #2

BL memset

MOV R0, #2147483648 //il valore che inserisce 1 al bit 31 e 0 a tutti gli altri (indica che possono essere lanciate le FIRQ)

MOV R1, #128 //portiamo a 1 il bit di “libertà”

ADD R2, R0, R1 //Le cose sono state fatte separatamente per colpa dello Shifter Operand

STR R2, [FP] //Il registro mappato in memoria viene inizializzato

ADD R2, R1, R0, lsr #1 //questo somma a 128 un registro che ha 1 solo al bit 30 (31 shiftato di 1)

STR R2, [FP, #+4] //Il registro mm per l’output può quindi solo mandare interruzioni non veloci

//a questo punto bisogna preparare il processore a gestire la funzione utente

MOV FP, #40960 //si assegna al programma una certa area di memoria

MOV SP, #49152

MOV LP, #main //si identifica la prima istruzione da eseguire

STM SP!, FP, SP, LP //si fa partire il programma in modalità utente

LDMDB SP!, FP, SP, PC^ //Questo fa sì che si passi al modo di esecuzione applicazione. Infatti, le ultime due cifre dell’LP saranno 00.

Per quanto riguarda le system call, necessarie per accedere ai buffer di sistema, sono implementate attraverso l’istruzione SWI.

4 bit sono dedicati alle condizioni di esecuzione e altri 4 al codice operativo della SWI, ne restano liberi 24. Questi 24 sono usati per inviare la motivazione della SWI, solo che durante l’esecuzione quel numero viene ignorato e non salvato: per cui la prima cosa da fare è andare a recuperare la sequenza di 24 bit dal segmento codice. Per fare ciò basta prendere l’indirizzo del link pointer -8 e vedere a quale istruzione puntava

Come funziona quindi l’handler della SWI?

Swi\_h: 336: STM SP!, R4, R5, R6, R7, R8, R9, R10, LP //la maggior parte delle volte non ci serviranno tutti ma dobbiamo considerare il caso peggiore

BIC LP, LP, #2113929216 //il BIC corrisponde al BIT CLEAR = AND NOT, ci serve per essere sicuri di mantenere soltanto l’indirizzo dell’istruzione (puliamo la parte di “registro di stato”): sappiamo infatti che gli ultimi due bit saranno a 00, ma non abbiamo nulla di garantito per i primi 6. La costante equivale a 111111000…000.

LDR R8, [LP, #-8] //questo ci salva in R8 il codice operativo della nostra SWI

BIC R8, R8, #2139095040 //questo ci lascia in R8 soltanto i 24 bit magici di motivazione

/\*A questo punto si definisce una convenzione sul significato dei 24 bit\*/

CMP R8, #0

BEQ exit //se in R8 c’è 0 si fa il branch a una cella contenente il codice exit

CMP R8, #1

MOVEQ R10, #buffers //a 1 associamo il significato di lettura da standard input

BEQ read\_stdin //si definisce convenzionalmente che read\_stdin vuole in R0 l’indirizzo del buffer applicativo, in R1 la lunghezza di tale buffer (= n° effettivo di caratteri letti)

CMP R8, #2

BEQ write\_stdout //indovina che fa.