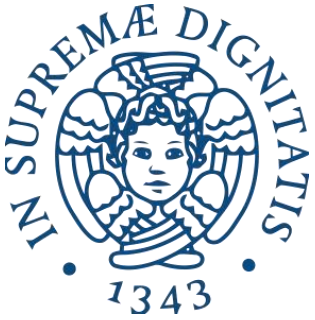


UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laurea in Ingegneria Informatica

Design e implementazione di un ambiente di simulazione e testing in Verilog per il processore sEP8

Relatori:
Prof. Giovanni Stea
Dott. Raffaele Zippo

Candidato:
Nicola Ramacciotti

ANNO ACCADEMICO 2022-2023

Indice

Capitolo 1

Introduzione

Il sEP8, noto anche come *8 bit SIMPLE EDUCATIONAL PROCESSOR*, è un processore a 8 bit perfettamente funzionante, introdotto e utilizzato nel corso di Reti Logiche di Ingegneria Informatica presso l'Università di Pisa con lo scopo di mostrare una versione semplificata, ma contenente i concetti fondamentali, di un processore, cioè l'unità di elaborazione centrale di un qualsiasi calcolatore.
Per un ingegnere informatico, la comprensione approfondita dei meccanismi di un processore è un aspetto importante. Il sEP8 è un valido strumento all'interno del corso per comprendere i principi fondamentali dell'architettura dei processori.
La descrizione di questo processore può essere un buon punto di partenza per tutti gli interessati a questo argomento per approfondire in maniera dettagliata la propria conoscenza sull'architettura dei processori, permettendo di interagire con concetti solitamente teorici in maniera pratica e visuale, portando quindi ad una migliore comprensione dei principali meccanismi alla base del funzionamento di un processore.
L'ambiente di simulazione è implementato attraverso Verilog, un linguaggio di descrizione hardware (HDL), largamente utilizzato nell'ambito del design di hardware digitale.
Questo linguaggio consente di progettare e simulare circuiti digitali in maniera dettagliata, fornendo un ambiente solido per la loro verifica, offrendo infatti la possibilità di scrittura di moduli dedicati al testing di vari sistemi digitali in un ambiente di simulazione senza la necessità di dover abbandonare l'ambiente offerto da un qualsiasi computer, rendendo il processo di verifica molto efficiente e limitando le prove dirette su hardware, che richiedono un consumo elevato di risorse e tempo.
Un processore è l'unità di elaborazione centrale alla base di un calcolatore. L'obbiettivo di questo lavoro è ben definito, si mira a sviluppare un ambiente di simulazione completo e funzionante per il processore didattico, implementato in Verilog, in modo tale che possa essere simulato e osservato nel dettaglio il suo funzionamento.
L'implementazione di questo ambiente offre la possibilità di eseguire programmi che presentano caratteristiche interessanti per vedere l'evoluzione interna del processore, attraverso un ambiente pratico e interattivo.

Per realizzare tutto questo ambiente sono partito dalla descrizione del processore disponibile nel libro [1] e successivamente sono andato ad ampliarla e ad aggiungere tutte quelle parti mancanti e necessarie per poter portare a termine l'obbiettivo del lavoro. Ho quindi completato il codice del processore ed inserito tutte le parti del calcolatore che il processore sfrutta per eseguire ogni sua elaborazione, in modo da poter osservare durante la simulazione il comportamento del processore. Ulteriori dettagli saranno forniti nei capitoli successivi.
I materiali relativi a questo lavoro, in particolare il codice Verilog completo, sono disponibili nel repository https://github.com/nicorama06/sEP8.git su GitHub, piattaforma popolare per ospitare i progetti. Questa piattaforma può essere utile anche in vista di lavori successivi sul progetto, espandendo il funzionamento del processore mostrato e introducendo sempre più elementi che sono tipici dei processori moderni.

Capitolo 2

L’architettura sEP8

Prima di entrare nel vivo del lavoro, concentriamoci sull’architettura di un calcolatore basato sul processore sEP8.
È un’architettura semplificata rispetto a quelle moderne, ma che comunque contiene i concetti fondamentali per capire il funzionamento di un calcolatore ed è una buona base per comprendere a pieno le potenzialità di tali dispositivi.
I calcolatori sono complesse macchine in grado di eseguire elaborazioni sui dati. Sono costituiti da vari moduli fisici, l’hardware, ciascuno dedicato all’esecuzione di funzioni specifiche, che permettono di leggere ed eseguire un programma, il software, ottenendo una variazione significativa nell’elaborazione dei dati al cambio del programma.
Per avere una visione d’insieme in Fig.1 è presente il tipico schema a blocchi di un calcolatore.

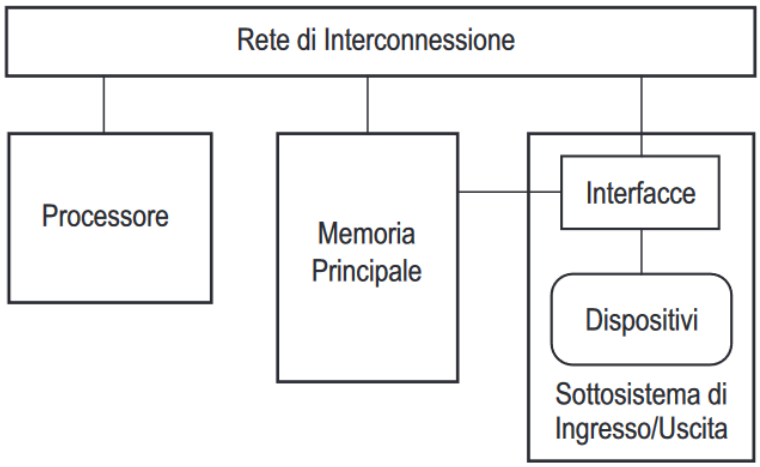


Fig.1 Schema a blocchi di un semplice calcolatore [1, p.169].

Esso comprende il *processore*, la *memoria principale* e un *sottosistema di ingresso e uscita*. Questi moduli sono collegati tra loro attraverso una *rete di interconnessione*, detta anche *bus*.
Esaminiamo adesso i moduli interni al calcolatore.

2.1 Sottosistema di ingresso/uscita

Il *sottosistema di ingresso/uscita* (detto anche *spazio di I/O*) è il responsabile delle interazioni del sistema con il mondo esterno attraverso scambio di informazioni codificate secondo una certa modalità.

All'interno del modulo troviamo i dispositivi e le interfacce. I primi sono i veri e propri responsabili della codifica dei dati da e verso il mondo esterno, mentre le interfacce sono delle componenti fondamentali che permettono al processore di interagire con i vari dispositivi, quali mouse, tastiera, ecc. con modalità standard.

In sintesi, il sottosistema ha un ruolo fondamentale all'interno del calcolatore, consentendo di avere una comunicazione con il mondo esterno.

Nell'architettura sEP8, il sottosistema consiste in una serie di 64k locazioni, dette anche porte, in cui ogni porta è indirizzabile tramite indirizzi su due byte ed è in grado di trasferire un byte.

2.2 La memoria principale

La *memoria principale* (detta anche *spazio di memoria*), durante l'elaborazione, contiene una copia del programma e i dati necessari per proseguire nell'elaborazione.

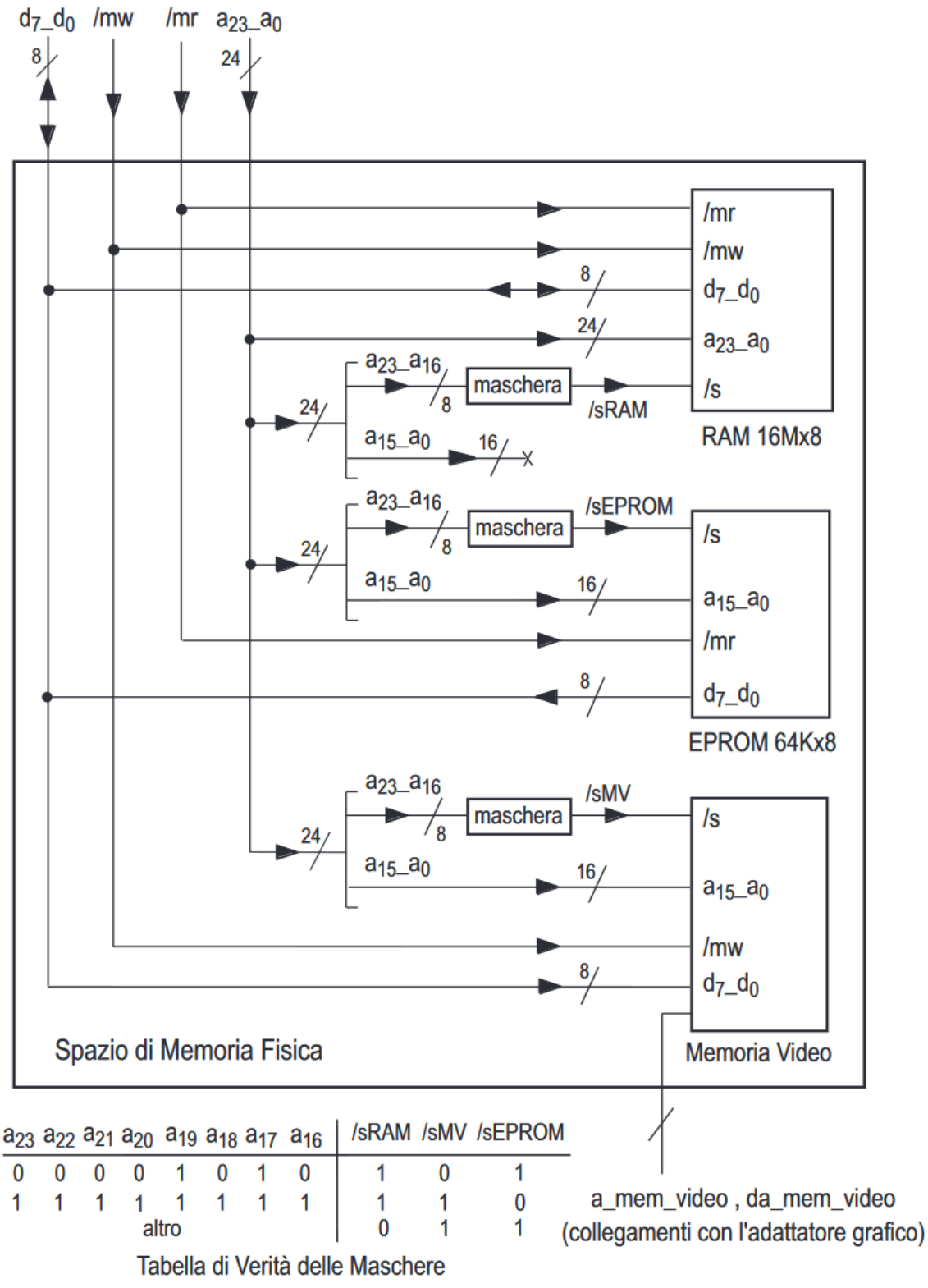
Essa consiste in una serie di 16M locazioni, in cui ogni locazione è indirizzabile tramite indirizzi su tre byte ed è in grado di contenere un byte. Lo spazio di memoria, illustrato in Fig. 2, è composto da moduli RAM e moduli ROM, inoltre l'architettura proposta nel libro prevede una sezione designata per la memoria video, la quale viene letta direttamente da componenti hardware predisposti all'output sul monitor. Questa funzionalità non è implementata in questo progetto.

Lo spazio di memoria è quindi suddiviso in molteplici zone:

1) volatile: porzione della memoria implementata con moduli RAM ed è mappata tra l'indirizzo 0x000000 e l'indirizzo 0x09FFFF;
2) video: porzione implementata con una memoria video ed è mappata tra l'indirizzo 0x0A0000 e l'indirizzo 0x0AFFFF;
3) volatile: altra porzione della memoria implementata con moduli RAM, mappata tra l'indirizzo 0x0B0000 e l'indirizzo 0xFEFFFF;
4) non volatile: porzione implementata con moduli ROM ed è mappata tra l'indirizzo 0xFF0000 e l'indirizzo 0xFFFFFF.

Come già detto in questo lavoro la memoria video non sarà presa in considerazione e lo spazio dedicata ad essa è lasciato alla memoria volatile. Quindi abbiamo che la memoria ROM ha una capacità totale di 64 Kbyte mentre il resto dello spazio di memoria contiene la memoria RAM.

Quando si ha un accesso alla memoria il modulo corretto è selezionato tramite una maschera sull'indirizzo che seleziona esclusivamente uno dei due moduli.



a_{23}	a_{22}	a_{21}	a_{20}	a_{19}	a_{18}	a_{17}	a_{16}	/sRAM	/sMV	/sEPROM
0	0	0	0	1	0	1	0	1	0	1
1	1	1	1	1	1	1	1	1	1	0
altro								0	1	1

Fig. 2 – Possibile implementazione della memoria [1, p.191].

2.3 Il processore

Durante il suo funzionamento a regime, il processore presenta un comportamento ciclico, schematizzato in Fig. 3. Nella *fase di chiamata*, anche detta di prelievo, il processore si procura la prossima istruzione da eseguire e gli operandi necessari, mentre nella *fase di esecuzione* esegue le operazioni specificate dall'istruzione letta nella fase precedente.
Se il processore non riconosce un'istruzione valida o se esegue un'istruzione *hlt* finisce in uno *stato di blocco*.

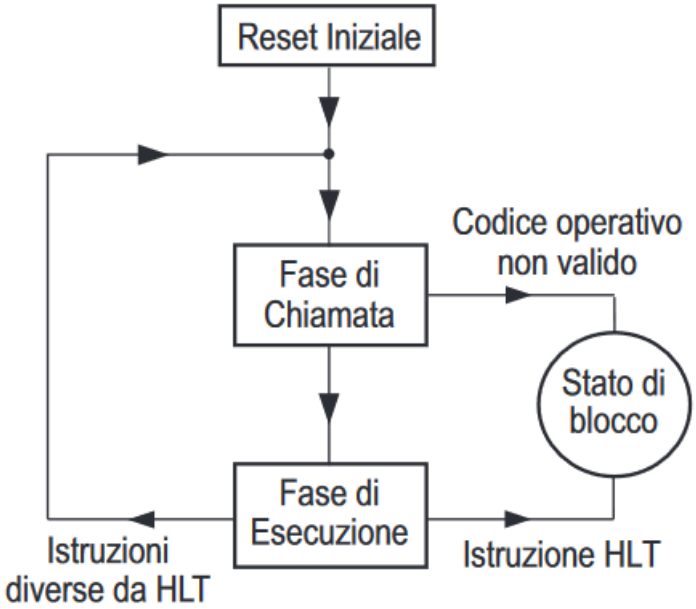


Fig. 3 – Schema del funzionamento del processore [1, p.197].

Le istruzioni che un processore esegue non sono altro che sequenze di bit e si dividono in due grandi categorie: quelle operative e quelle di controllo. Le istruzioni non solo specificano l'elaborazione che il processore deve realizzare, ma anche le modalità per individuare i dati da elaborare, per quanto riguarda le istruzioni operative, o la prossima istruzione da eseguire, per quanto riguarda le istruzioni di controllo. Tra quelle operative possiamo trovare ad esempio le operazioni aritmetiche, logiche e quelle di scrittura e lettura in memoria. Le operazioni di controllo, invece, sono le responsabili della possibile variazione al flusso di esecuzione sequenziale del processore.
Le istruzioni del processore si dividono in otto formati e sono riconoscibili dai tre bit più significativi del primo byte che viene prelevato durante la fase di prelievo di ogni istruzione. La suddivisione in formati è importante perché permette al processore di identificare i vari modi in cui vanno prelevati gli operandi delle istruzioni, elaborati durante la fase di esecuzione.
Il processore accede alla memoria nella fase di prelievo delle istruzioni e nella fase di esecuzione delle istruzioni operative, mentre accede allo spazio di I/O nella fase di esecuzione delle istruzioni di ingresso o di uscita.

Il processore, per poter funzionare secondo il modello descritto, necessità di una condizione iniziale in cui la memoria contenga un primo programma (bootstrap) e il processore sia inizializzato in modo tale che, durante la fase di *reset iniziale*, mostrata in Fig. 3, esso parta a prelevare le istruzioni da una zona di memoria principale ben precisa, implementata con tecnologie non volatili, che permettono la conservazione del programma bootstrap.
Il processore sEP8 ha al suo interno una serie di registri, mostrati in Fig. 4, di cui quelli visibili dal programmatore sono i *registri accumulatore* (AH e AL) su un byte, dove il processore memorizza gli operandi, i *registri puntatore* (DP, SP e IP) su tre byte, utilizzati per indirizzare locazioni di memoria, e il *registro dei flag* (F), che presenta informazioni utili al controllo del flusso di esecuzione del programma. Inoltre, è in grado di indirizzare una memoria di 16Mbyte e di elaborare dati a 8 bit. Esso lavora in base 2, interpretando i dati come numeri naturali o come interi in complemento a due.

2.4 Altre componenti utili

Il processore e le interfacce sono reti sequenziali sincronizzate, affinché il tutto funzioni correttamente abbiamo quindi bisogno di un dispositivo che riporti nelle condizioni iniziali le varie reti. Questo lavoro è fatto dal gruppo RC con trigger di Schmitt.
Il segnale di sincronizzazione necessario per il corretto funzionamento del sistema a regime è invece ottenuto da un modulo destinato alla generazione del clock.

2.5 Rete di interconnessione

Arrivati a questo punto, concentriamo l'attenzione sullo studio del bus del calcolatore basato sul processore sEP8, che comprende tutti i vari collegamenti tra i vari moduli.
La rete di collegamento presenta un insieme di fili, visibili in Fig. 4, necessari per il corretto funzionamento del sistema:

1) variabile a 24 bit *a23_a0*: variabile di uscita per il processore necessaria per l'accesso alle locazioni di memoria e dello spazio di I/O;
2) variabile a 8 bit *d7_d0*: variabile che può funzionare sia da ingresso che da uscita per il processore, necessaria per le letture e scritture, contiene rispettivamente o i dati in ingresso dallo spazio di memoria o dal modulo di I/O o i dati da mandare in uscita sulla memoria o sullo spazio di I/O;
3) variabili */mr*, */mw*, */ior*, */iow*: variabili di uscita per il processore, necessarie a pilotare rispettivamente la lettura in memoria, la scrittura in memoria, la lettura nello spazio di I/O e la scrittura nello spazio di I/O;
4) variabile *clock*: variabile in uscita dal generatore di clock, necessaria per portare il segnale di sincronizzazione al processore e alle interfacce dello spazio di I/O;
5) variabile /reset: variabile di uscita dal gruppo RC con trigger di Schmitt, necessaria per riportare nelle condizioni iniziali le varie reti sequenziali sincronizzate.

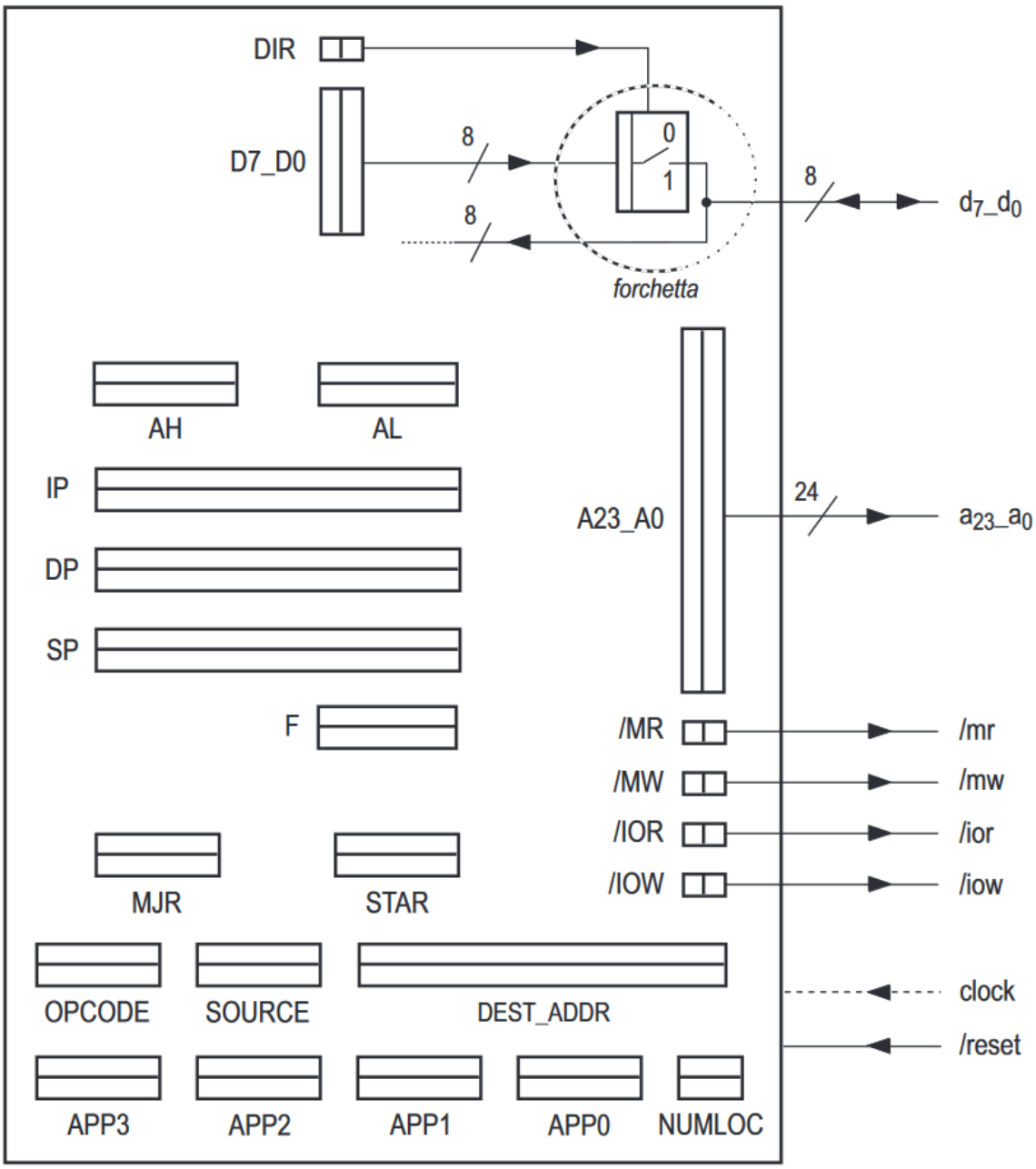


Fig. 4 – Schema con registri e collegamenti del processore sEP8 [1, p.198].

Nei prossimi capitoli vedremo come questi moduli interagiscono per eseguire elaborazioni complesse e produrre risultati affidabili durante la simulazione del comportamento del processore sEP8.

Capitolo 3

Ambiente di simulazione in Verilog

Come già detto, il Verilog è un linguaggio HDL molto potente che permette di descrivere hardware e anche di testarne il comportamento tramite l'utilizzo di apposite testbench.
In questo capitolo verranno mostrati tutti i moduli che sono stati necessari per poter descrivere e simulare il comportamento del processore sEP8.

3.1 Configurazioni iniziali

Il processore lavora con una codifica in linguaggio macchina, basata su sequenze binarie. Per facilitare il lavoro, quindi, è stato sviluppato un file dedicato al contenimento di parametri utili per altri moduli, permettendo così di evitare la scrittura diretta di zeri e uni, fornendo un formato più chiaro e leggibile per un utente. Nella Fig. 5 è possibile vedere una parte del codice che implementa questo aspetto.

```
1.              // FORMATO F0
2. parameter    opcode_hlt    = 8'b00000000,
3.              opcode_nop    = 8'b00000001,
4.              // ...
5.              // FORMATO F1
6.              opcode_inAL             =  8'b00100000,
7.              opcode_outAL            =  8'b00100001,
8.              // ...
9.              // FORMATO F2
10.             opcode_mov_DP_AL =  8'b01000000,
11.             opcode_cmp_DP_AL =  8'b01000001,
12.             // ...
13.             // FORMATO F3
14.             opcode_mov_AL_DP =  8'b01100000,
15.             opcode_mov_AH_DP =  8'b01100001,
16.             // FORMATO F4
17.             opcode_mov_operando_AL =  8'b10000000,
18.             opcode_cmp_operando_AL =  8'b10000001,
19.             // ...
20.             // FORMATO F5
21.             opcode_mov_indirizzo_AL =  8'b10100000,
22.             opcode_cmp_indirizzo_AL =  8'b10100001,
23.             // ...
24.             // FORMATO F6
25.             opcode_mov_AL_indirizzo =  8'b11000000,
```

```
26.             opcode_mov_AH_indirizzo =  8'b11000001,
27.             // FORMATO F7
28.             opcode_jmp  =  8'b11100000,
29.             opcode_je   =  8'b11100001,
30.             // ...
```

Fig. 5 – Parte del codice che contiene gli opcode sotto forma di parametri.

3.2 Il processore sEP8

Successivamente è stata completata la descrizione in Verilog, presentata durante il corso, aggiungendo l'elenco dei parametri necessari al modulo e le varie reti combinatorie interne al processore indispensabili per eseguire correttamente le istruzioni. Queste reti combinatorie sono state realizzate in Verilog tramite l'uso di *function*.

3.2.1 Valid fetch

Prende in ingresso un byte e restituisce uno se viene verificato che rappresenta l'opcode di un'istruzione nota, in caso contrario restituisce zero.

Una parte del codice è visibile in Fig. 6.

```
 1. function valid_fetch;
 2.         input[7:0] opcode;
 3.         casex (opcode)
 4.             // FORMATO F0
 5.             opcode_hlt: valid_fetch = 1'b1;
 6.             // ...
 7.             // FORMATO F1
 8.             opcode_inAL: valid_fetch = 1'b1;
 9.             // ...
10.             // FORMATO F2
11.             opcode_mov_DP_AL: valid_fetch = 1'b1;
12.             // ...
13.             // FORMATO F3
14.             opcode_mov_AL_DP: valid_fetch = 1'b1;
15.             // ...
16.             // FORMATO F4
17.             opcode_mov_operando_AL: valid_fetch = 1'b1;
18.             // ...
19.             // FORMATO F5
20.             opcode_mov_indirizzo_AL: valid_fetch = 1'b1;
21.             // ...
22.             // FORMATO F6
23.             opcode_mov_AL_indirizzo: valid_fetch = 1'b1;
24.             // ...
25.             // FORMATO F7
26.             opcode_jmp  : valid_fetch = 1'b1;
27.             // ...
28.             // ISTRUZIONE NON NOTA
29.             default: valid_fetch = 1'b0;
30.         endcase
31. endfunction
```

Fig. 6 – Parte del codice che implementa la rete combinatoria.

3.2.2 First execution state

Prende in ingresso un byte e restituisce il valore del primo stato di esecuzione, relativo ad una istruzione, in cui si deve portare il processore.
Una parte del codice è visibile in Fig. 7.

```
function[6:0] first_execution_state;
        input[7:0] opcode;
        casex (opcode)
            // FORMATO F0
            opcode_hlt : first_execution_state = hlt;
            // ...
            // FORMATO F1
            opcode_inAL : first_execution_state = in;
            // ...
            // FORMATO F2
            opcode_mov_DP_AL : first_execution_state = ldAL;
            // ...
            // FORMATO F3
            opcode_mov_AL_DP : first_execution_state = storeAL;
            // ...
            // FORMATO F4
            opcode_mov_operando_AL : first_execution_state = ldAL;
            // ...
            // FORMATO F5
            opcode_mov_indirizzo_AL : first_execution_state = ldAL;
            // ...
            // FORMATO F6
            opcode_mov_AL_indirizzo : first_execution_state = storeAL;
            // ...
            // FORMATO F7
            opcode_jmp  : first_execution_state = jmp;
            // ...
        endcase
endfunction
```

Fig. 7 – Parte del codice che implementa la rete combinatoria.

3.2.3 JMP condition

Prende in ingresso due byte. Il primo è interpretato come opcode, il secondo come registro dei flag. Viene quindi controllato il contenuto dei flag a seconda dell'opcode e viene restituito uno in caso di controllo andato a buon fine, zero altrimenti. Questa serve per implementare la funzionalità dei salti condizionati.
Una parte del codice è visibile in Fig. 8.

```
function jmp_condition;
        input[7:0] opcode;
        input[7:0] flag;
        casex (opcode)
            opcode_jmp : jmp_condition = 1'b1;
            opcode_je  : jmp_condition = ( flag[1]==1 ) ? 1'b1: 1'b0 ;
            // ...
            opcode_jns : jmp_condition = ( flag[2]==0 ) ? 1'b1: 1'b0 ;
```

```
 9.         endcase
10. endfunction
```

Fig. 8 – Parte del codice che implementa la rete combinatoria.

3.2.4 ALU result

Prende in ingresso tre byte. Il primo è interpretato come opcode, gli altri sono interpretati come operandi per le varie operazioni. Infatti questa rete combinatoria ha il ruolo di simulare una ALU interna al processore. Restituisce il risultato dell'operazione eseguita dalla ALU.

Una parte del codice è visibile in Fig. 9.

```
 1. function[7:0] alu_result;
 2.         input[7:0] opcode, operando1, operando2;
 3.         casex (opcode)
 4.             opcode_cmp_DP_AL: alu_result = operando2;
 5.             opcode_add_DP_AL: alu_result = operando1 + operando2;
 6.             opcode_sub_DP_AL: alu_result = operando2 - operando1;
 7.             opcode_and_DP_AL: alu_result = operando1 & operando2;
 8.             opcode_or_DP_AL: alu_result = operando1 | operando2;
 9.             // ...
10.             opcode_notAL: alu_result = ~operando2;
11.             opcode_shlAL: alu_result = operando2<<1;
12.             opcode_shrAL: alu_result = operando2>>1;
13.         endcase
14. endfunction
```

Fig. 9 – Parte del codice che implementa la rete combinatoria.

3.2.5 ALU flag

Prende gli stessi ingressi della ALU RESULT, ma restituisce un byte che indica il nuovo contenuto del registro dei flag coerentemente con l'opcode e gli operandi inseriti.

Una parte del codice è visibile in Fig. 10.

```
 1. function[3:0] alu_flag; // OF SF ZF CF
 2.         input[7:0] opcode, operando1, operando2;
 3.         reg [7:0] and_bit_bit;
 4.         // ...
 5.         begin
 6.                 and_bit_bit = operando1 & operando2;
 7.                 // ...
 8.                 casex (opcode)
 9.                           opcode_and_DP_AL: alu_flag = {
10.                                     1'b0,
11.                                     and_bit_bit[7]==1?1'b1:1'b0,
12.                                     and_bit_bit==0?1'b1:1'b0,
13.                                     1'b0
14.                           };
15.                           opcode_shlAL: alu_flag = {
16.                                     1'b0,
17.                                     1'b0,
18.                                     1'b0,
19.                                     operando2[7]
```

```
                        };
                        // ...
                endcase
        end
endfunction
```

Fig. 10 – Parte del codice che implementa la rete combinatoria.

3.3 Generatore di clock

È il modulo necessario a generare il segnale di clock.

Una possibile implementazione è mostrata in Fig. 11.

```
module clock_generator(
    clock
);
    parameter HALF_PERIOD = 5;
    output clock;
    reg CLOCK;
    assign clock = CLOCK;
    initial CLOCK <= 0;
    always #HALF_PERIOD CLOCK <= ~CLOCK;
endmodule
```

Fig. 11 – Codice che implementa il generatore del segnale di clock.

3.4 Spazio di memoria

Lo spazio di memoria contiene al suo interno due moduli ed è organizzato in maniera tale da direzionare al modulo corretto a seconda della richiesta attraverso circuiteria interna.

Al suo interno contiene un modulo RAM e un modulo ROM. Come già detto, in questo lavoro non è stata introdotta la memoria video.

In Fig. 12 si può trovare una sua possibile realizzazione.

```
module MEMORIA (
    a23_a0,
    mr_,mw_,
    d7_d0
);
    input[23:0] a23_a0;
    input mr_,mw_;
    inout[7:0] d7_d0;
    function[1:0] seleziona;
        input [7:0] a23_a16;
        casex(a23_a16)
            8'B11111111: seleziona = 2'B10;
            default:     seleziona = 2'B01;
        endcase
    endfunction
    wire selettoreRAM_,selettoreROM_;
    assign {selettoreRAM_,selettoreROM_} = seleziona(a23_a0[23:16]);
    ROM rom(
        .a23_a0(a23_a0),
        .s_(selettoreROM_),
        .mr_(mr_),
        .d7_d0(d7_d0)
```

```
23.     );
24.     RAM ram(
25.         .a23_a0(a23_a0),
26.         .s_(selettoreRAM_),
27.         .mr_(mr_),
28.         .mw_(mw_),
29.         .d7_d0(d7_d0)
30.     );
31. endmodule
```

Fig. 12 – Implementazione dello spazio di memoria.

Vediamo adesso come sono realizzati i moduli interni.

3.4.1 Modulo RAM

Il modulo RAM contiene una memoria volatile, destinata a contenere i programmi e i dati che il processore elaborerà. È stata realizzata tramite l'utilizzo di appositi costrutti del Verilog. In Fig. 13 si può vedere una possibile realizzazione.

```
 1. module RAM (
 2.     a23_a0,
 3.     s_, mw_, mr_,
 4.     d7_d0
 5. );
 6.     parameter dimensione_ram = (1<<24),
 7.               ritardo_lettura = 2,
 8.               ritardo_scrittura = 2;
 9.     input[23:0] a23_a0;
10.     input s_,mw_,mr_;
11.     inout[7:0] d7_d0;
12.     reg [7:0] CELLE[0:dimensione_ram-1];
13.     assign #ritardo_lettura d7_d0 = {s_,mr_,mw_}==3'B001 ? CELLE[a23_a0] : 8'HZZ;
14.     always @(d7_d0 , s_, mr_, mw_ ) #ritardo_scrittura
15.         CELLE[a23_a0]<= {s_,mr_,mw_}==3'B010 ? d7_d0 : CELLE[a23_a0];
16. endmodule
```

Fig. 13 – implementazione del modulo RAM.

3.4.2 Modulo ROM

Il modulo ROM contiene al suo interno la memoria non volatile, cruciale nella fase iniziale di avvio del calcolatore, in quanto contiene il programma di bootstrap. In questo caso il programma di avvio è il programma che verrà eseguito dal processore durante la simulazione.

Per facilitare la scrittura del modulo ROM è stato adattato un assemblatore in Python, reperibile nel seguente repository https://github.com/federicorossifr/sep8emulator.git, in modo tale che, partendo da un file Assembly semplificato, si riesca ad ottenere un modulo ROM, che può essere usato nel contesto della simulazione.

Vediamo adesso la sua realizzazione in Fig. 14.

```
 1. module ROM (
 2.     a23_a0,
 3.     s_,
```

```
 4.     mr_,
 5.     d7_d0
 6. );
 7.     parameter ritardo_lettura = 2;
 8.     input [23:0] a23_a0;
 9.     input s_,mr_;
10.     output [7:0] d7_d0;
11.     `include "parametri_opcode.v"
12.     function [7:0] valore;
13.         input [23:0] a23_a0;
14.         casex (a23_a0)
15.             24'HFF0000: valore = opcode_nop; // NOP
16.             // ...
17.             24'HFF000E: valore = opcode_hlt; // HLT
18.         endcase
19.     endfunction
20.     assign #ritardo_lettura d7_d0 = {s_,mr_}==2'b00 ? valore(a23_a0) : 8'HZZ;
21. endmodule
```

Fig. 14 – Implementazione del modulo ROM.

3.5 Spazio di I/O

Lo spazio di I/O implementato in questo lavoro contiene solo un modulo che rappresenta un'interfaccia di uscita, che ha lo scopo di stampare a video il carattere che viene passato con l'istruzione di uscita relativa all'indirizzo dove è presente questa interfaccia. Di questa interfaccia ne sono state realizzate due versioni: una che prevede la sincronizzazione tra processore e dispositivo per poter immettere nuovi dati sull'interfaccia e una senza sincronizzazione. Qui viene mostrata la versione con sincronizzazione. Al fine di avere la sincronizzazione si sfruttano le potenzialità del Verilog che permette di usare codice destinato alla simulazione dentro lo spazio di memoria per simulare il comportamento del dispositivo connesso all'interfaccia.

In Fig. 15 è mostrata l'implementazione.

```
 1. module IO (
 2.     clock,
 3.     reset_,
 4.     ior_,
 5.     iow_,
 6.     d7_d0,
 7.     a15_a0
 8. );
 9.     input clock, reset_;
10.     input ior_, iow_;
11.     inout [7:0] d7_d0;
12.     input [15:0] a15_a0;
13.     reg fo;
14.     wire [7:0] carattere;
15.     wire out_s_;
16.     assign out_s_ = a15_a0[15:1] == 15'B000000000000000 ? 1'B0 : 1'B1;
17.     interfaccia_stampa_carattere stampa_carattere(
18.         .ior_(ior_),
```

```
        .iow_(iow_),
        .s_(out_s_),
        .a0(a15_a0[0]),
        .d7_d0(d7_d0),
        .fo(fo),
        .carattere(carattere)
    );
    initial
        begin
            fo = 0;
            while(1) begin
                #200 // simulo il comportamento del dispositivo
                fo = 1;
                @(carattere) // devo aspettare che venga fatta una scrittura
                fo = 0;
            end
        end
endmodule
```

Fig. 15 – Implementazione dello spazio di I/O.

3.5.1 Interfaccia di uscita stampa carattere

Questa interfaccia ha il compito di stampare come output della simulazione caratteri.

Vediamo quindi l'implementazione di questa interfaccia di uscita analizzando il codice in Fig. 16.

```
module interfaccia_stampa_carattere (
    iow_, ior_,
    s_, a0, d7_d0,
    fo,
    carattere
);
    input iow_, ior_;
    input s_;
    input a0;
    input fo;
    inout [7:0] d7_d0;
    output [7:0] carattere;
    reg DIR;
    reg [7:0] D7_D0;
    assign d7_d0 = DIR == 1 ? D7_D0 : 8'HZZ;
    reg [7:0] CARATTERE;
    assign carattere = CARATTERE;
    wire eb;
    assign eb = {s_, iow_, a0} == 3'B001 ? 1'B1 : 1'B0;
    wire es;
    assign es = {s_, ior_, a0} == 3'B000 ? 1'B1 : 1'B0;
    always @(*) begin
        if( s_==1 || ior_ ==1 )
            DIR <= 0;
    end
    always @(*)
        if ( es==1'B1 ) // operazione di lettura all'indirizzo corretto
                begin
```

```
29.                     D7_D0 <= {2'B0, fo, 5'B0};
30.                     DIR <= 1;
31.                 end
32.     always @(*)
33.         if( eb==1'b1 && fo==1 ) // operazione di scrittura all'indirizzo corretto
34.                 begin
35.                     $write("%c",d7_d0);
36.                     CARATTERE <= d7_d0;
37.                 end
38. endmodule
```

Fig. 16 – Implementazione di una interfaccia di uscita.

Si può notare che anche questa interfaccia ha al suo interno del codice utile per la simulazione. Infatti, quando il processore esegue un'istruzione di uscita viene stampato come output della simulazione il carattere in codifica ASCII, passato come operando dell'istruzione.

3.6 Testbench

La testbench contiene al suo interno, tutti i vari moduli introdotti in precedenza ed è utile per far partire la simulazione e controllare il momento in cui essa termina, ovvero nel caso in cui venga incontrata un'istruzione non valida o un'istruzione *hlt*. Inoltre, la testbench è la responsabile della creazione di un file di log (*waveform.vcd*) con il quale è possibile vedere l'evoluzione del processore nel tempo.

Il codice è quello mostrato in Fig. 17.

```
 1. module testbench();
 2.     // bus
 3.     wire clock;
 4.     wire[7:0] d7_d0;
 5.     wire[23:0] a23_a0;
 6.     wire [15:0] a15_a0;
 7.     wire mr_,mw_,ior_,iow_;
 8.     wire tb_halt;
 9.     wire tb_nvi;
10.     reg reset_;
11.     assign a15_a0 = a23_a0[15:0];
12.     // moduli
13.     clock_generator clk(
14.         .clock(clock)
15.     );
16.     MEMORIA memoria(
17.         .a23_a0(a23_a0),
18.         .mr_(mr_),
19.         .mw_(mw_),
20.         .d7_d0(d7_d0)
21.     );
22.     Processore sEP8(
23.         .d7_d0(d7_d0),
24.         .a23_a0(a23_a0),
25.         .mr_(mr_),
26.         .mw_(mw_),
```

```
27.         .ior_(ior_),
28.         .iow_(iow_),
29.         .clock(clock),
30.         .reset_(reset_),
31.         .tb_halt(tb_halt),
32.         .tb_nvi(tb_nvi)
33.     );
34.     IO spazio_IO (
35.         .clock(clock),
36.         .reset_(reset_),
37.         .ior_(ior_),
38.         .iow_(iow_),
39.         .a15_a0(a15_a0),
40.         .d7_d0(d7_d0)
41.     );
42.     initial
43.         begin
44.             $dumpfile("waveform.vcd");
45.             $dumpvars;
46.             reset_ = 0;
47.             #(clk.HALF_PERIOD)
48.             reset_ = 1;
49.             fork
50.                 begin
51.                     @(posedge tb_halt) begin
52.                         $display("\nSimulazione terminata: il processore ha eseguito
un'istruzione HLT");
53.                         $finish;
54.                     end
55.                 end
56.                 begin
57.                     @(posedge tb_nvi) begin
58.                         $display("\nSimulazione terminata: il processore ha prelevato
un'istruzione non valida");
59.                         $finish;
60.                     end
61.                 end
62.             join
63.         end
64. endmodule
```

Fig. 17 – Implementazione della testbench per la simulazione.

Abbiamo quindi introdotto tutte le componenti necessarie per simulare il comportamento del processore. Nel prossimo capitolo verranno analizzati frammenti interessanti di codice eseguito dal processore e la relativa waveform, che mostra l'evoluzione del processore durante l'esecuzione del codice.

Capitolo 4

Esempi di uso del simulatore

In questo capitolo verrà mostrata l'evoluzione del processore durante l'esecuzione di parti di codice interessante. Vediamo quindi degli esempi partendo dal codice in assembly, passando poi dalla sua rappresentazione in binario e, infine, guardando l'evoluzione del processore.

4.1 Condizioni iniziali

Prima di vedere gli esempi però, concentriamoci sulla fase di reset iniziale del processore, che sarà uguale tutte le volte che verrà fatta una simulazione del comportamento del processore durante l'esecuzione di un qualsiasi programma, perciò la riporto qua una sola volta.

La condizione iniziale di reset è impostata tramite testbench. Qui alcuni registri vengono inizializzati con il loro valore di default, per poter portare il sistema in uno stato consistente. Abbiamo infatti, come si può vedere in Fig. 18, che il registro F viene inizializzato con tutti zeri, il registro IP con l'indirizzo della prima locazione di memoria ROM, il registro STAR con il primo stato interno in cui si deve trovare il processore e da cui inizierà il suo ciclo di prelievo ed esecuzione di istruzioni. I registri IOR_, IOW_, MR_ e MW_ sono impostati in modo da non selezionare né lo spazio di memoria né lo spazio di I/O. Infine il registro DIR è imposto a zero in modo tale che i fili dei dati non siano in conduzione, per non avere problemi con gli altri moduli che potrebbero pilotarli. Gli altri registri non è necessario che nella fase di reset abbiano un valore predefinito.

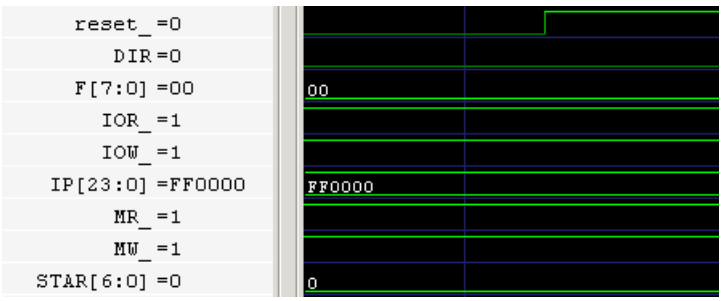


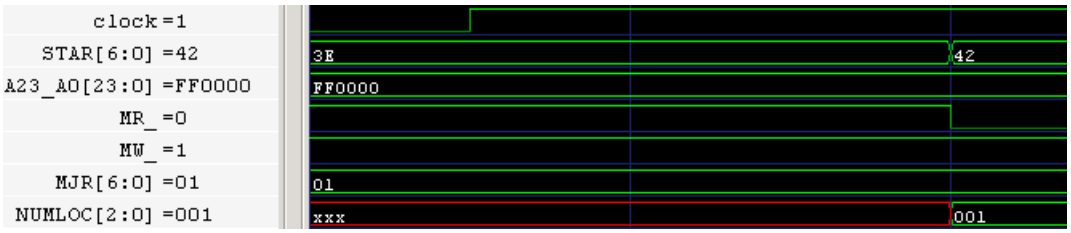
Fig. 18 – Condizioni dei registri del processore durante la fase di reset del processore

4.2 Ciclo di lettura in memoria

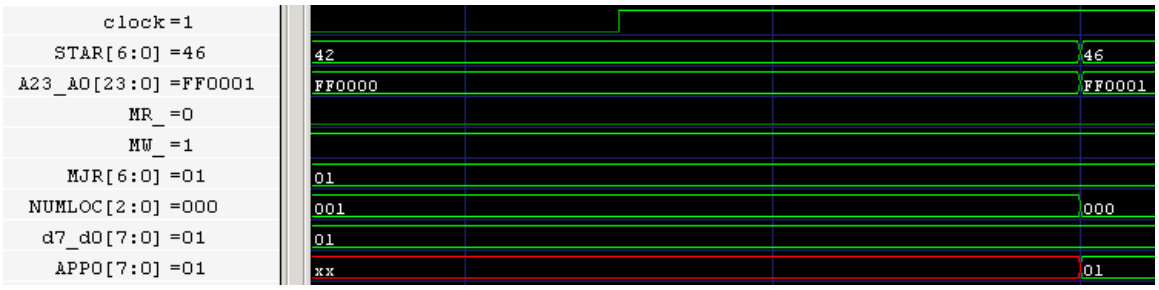
È importante inoltre vedere come si comporta il processore per una lettura nella memoria, in quanto anche questo aspetto è ripetitivo durante il funzionamento del processore e quindi verrà trattato solo una volta. Innanzitutto, vediamo gli stati interni del processore responsabili della lettura in memoria come descritti in [1, p.200].

```
 1. readB: begin MR_<=0; NUMLOC<=1; STAR<=read0; end
 2. readW: begin MR_<=0; NUMLOC<=2; STAR<=read0; end
 3. readM: begin MR_<=0; NUMLOC<=3; STAR<=read0; end
 4. readL: begin MR_<=0; NUMLOC<=4; STAR<=read0; en
 5. read0: begin APP0<=d7_d0; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1;
 6.              STAR<=(NUMLOC==1)?read4:read1; end
 7. read1: begin APP1<=d7_d0; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1;
 8.              STAR<=(NUMLOC==1)?read4:read2; end
 9. read2: begin APP2<=d7_d0; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1;
10.              STAR<=(NUMLOC==1)?read4:read3; end
11. read3: begin APP3<=d7_d0; A23_A0<=A23_A0+1; STAR<=read4; end
12. read4: begin MR_<=1; STAR<=MJR; TB_STAGE<=(MJR==fetch0)?0:TB_STAGE; end
```

Le prime quattro righe differiscono nel numero di byte da leggere che rispettivamente sono uno, due, tre e quattro byte. Le successive righe indicano il comportamento del processore a seconda del numero di byte da leggere. Per funzionare prima di entrare in questi stati dobbiamo avere nel registro A23_A0, che pilota la variabile di uscita a23_a0, il valore della prima locazione che vogliamo leggere. Si ha quindi nel primo stato che il registro MR_, che pilota la variabile mr_, viene messo a zero per iniziare la fase di lettura. Si ha quindi un passaggio tra i vari stati *read0*, *read1*, *read2*, *read3* a seconda del numero di byte da leggere e infine si arriva allo stato *read4*, dove si indica portando MR_ ad uno che la lettura è finita e andando nello stato successivo in cui si deve essere dopo la lettura, avendo conservato lo stato nel registro MJR. Il comportamento per una lettura di un byte è schematizzato nelle successive figure.



All'arrivo del clock, il contenuto dei registri si aggiorna avendo STAR che passa dallo stato 0x3E, cioè *readB*, allo stato 0x42, cioè read0. Inoltre, si ha che il contenuto del registro NUMLOC, che contiene il numero di locazioni da leggere, viene impostato ad uno. Infine, viene anche aggiornato il registro MR_ portandolo a zero e segnalando che il processore vuole accedere in memoria all'indirizzo segnalato dal contenuto di A23_A0.



Dallo stato *read0*, all'arrivo del clock, viene assegnato al registro di appoggio APP0 il valore di d7_d0. Vengono inoltre incrementato il contenuto di A23_A0, decrementato il contenuto di NUMLOC e modificato il valore di STAR in modo tale da passare dallo stato *read0* allo stato 0x46, cioè *read4*. Si passa subito allo stato finale della lettura in quanto l'obbiettivo in questo caso era leggere solo un byte, altrimenti NUMLOC avrebbe contenuto un altro valore e avremmo continuato a leggere dalla memoria.

All'arrivo del successivo clock in *read4*, si aggiorna STAR con il contenuto del registro MJR, che contiene lo stato in cui il processore si deve portare dopo essere passati dal microsottoprogramma. Infine, si pone il contenuto di MR_ a uno per segnalare la fine della lettura in memoria.

Questo è quindi il comportamento del processore durante la lettura in memoria.

4.3 Esempio di esecuzione di un programma

Partiamo con un esempio semplice, un programma, mostrato in Fig. 19, che stampa a schermo le lettere dell'alfabeto in minuscolo. Il programma parte alla riga 5 mettendo in ingresso nel registro AL il valore 'A', codificato in ASCII, esegue poi alla riga 7 un'istruzione di OR, andando ad ottenere in AL il valore 'a', infatti la differenza tra le lettere in maiuscolo e in minuscolo in codifica ASCII è il valore del bit in posizione 5, che, se settato come fa la OR, porta la lettera maiuscola in minuscolo. Dopodiché procede a mettere in uscita sull'interfaccia di uscita mappata all'indirizzo zero nello spazio di I/O con un'istruzione OUT alla riga 10. Successivamente incrementa il contenuto di AL di uno alla riga 13 e lo confronta alla riga 15 con 0x7A, la lettera 'z' in codifica ASCII, se il contenuto di AL è minore o uguale di quel valore con l'istruzione JBE alla riga 17 ritorna alla istruzione OUT alla riga 10, altrimenti prosegue il flusso normale di esecuzione e va alla riga 21 dove esegue l'istruzione HLT e termina. Notare che in questo esempio è usata l'interfaccia senza sincronizzazione, quindi non si fa un ciclo di attesa per controllare che il dato trasferito in uscita sia stato correttamente elaborato dal dispositivo connesso all'interfaccia, prima di trasferire un nuovo dato.

In Fig. 19 sono anche mostrati gli indirizzi della ROM dove si trova il programma e il contenuto di ogni cella della memoria ROM.

```
 1.    Programma Assembly   | Indirizzo Memoria ROM | Contenuto Memoria ROM
 2.                        |                       |
 3.     main:              |                       |
 4.         NOP            |       0xFF0000        |    0x01
 5.         MOV $0x41,%AL  |       0xFF0001        |    0x80
 6.                        |       0xFF0002        |    0x41
 7.         OR $0x20,%AL   |       0xFF0003        |    0x85
 8.                        |       0xFF0004        |    0x20
 9.     loop :             |                       |
10.         OUT %AL,0x0000 |       0xFF0005        |    0x21
11.                        |       0xFF0006        |    0x00
12.                        |       0xFF0007        |    0x00
13.         ADD $0x01,%AL  |       0xFF0008        |    0x82
14.                        |       0xFF0009        |    0x01
15.         CMP $0x7A,%AL  |       0xFF000A        |    0x81
16.                        |       0xFF000B        |    0x7A
17.         JBE loop       |       0xFF000C        |    0xE6
18.                        |       0xFF000D        |    0x05
19.                        |       0xFF000E        |    0x00
20.                        |       0xFF000F        |    0xFF
21.         HLT            |       0xFF0010        |    0x00
22.                        |                       |
```

Fig. 19 – Programma Assembly, Indirizzo e contenuto della memoria ROM.

Notare che le istruzioni, in base al formato, corrispondono ad un numero di byte diverso.

L'output della simulazione è mostrato in Fig. 20.

```
1. VCD info: dumpfile waveform.vcd opened for output.
2. abcdefghijklmnopqrstuvwxyz
3. Simulazione terminata: il processore ha eseguito un'istruzione HLT
```

Fig. 20 – Output della simulazione.

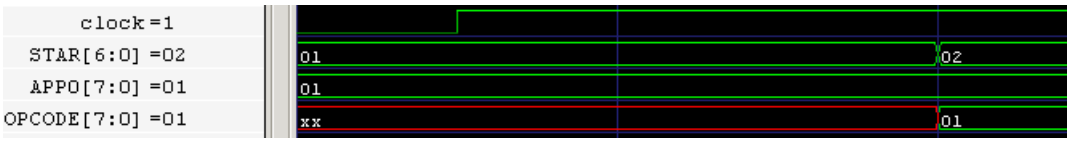
Il primo messaggio indica che è stato creato il file di log, per monitorare l'evoluzione del processore. La seconda riga mostra tutti i caratteri dell'alfabeto in minuscolo che l'interfaccia di uscita ha stampato, dopo che il processore ha eseguito le istruzioni di uscita. Infine, l'ultima riga, mostra il fatto che il processore ha eseguito l'ultima istruzione del programma Assembly, cioè la *hlt*, che interrompe la simulazione.

Andiamo ad analizzare quindi l'evoluzione del processore durante l'esecuzione di questo programma, tramite il file di log generato.

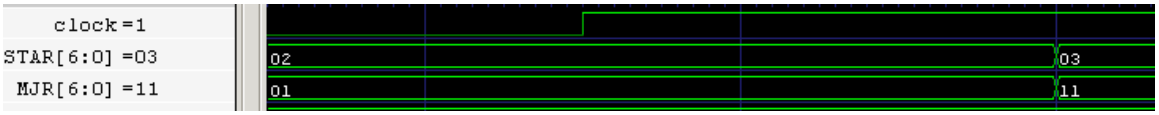
Tralasciamo le condizioni iniziali ed entriamo nel vivo del comportamento del processore.

Il processore si trova nello stato indicato dal registro STAR, che, alla fine della fase di reset contiene il valore zero, corrispondente al primo stato della fase di prelievo delle istruzioni. Infatti, da qui esegue una lettura nello spazio di memoria di un byte per ottenere una nuova istruzione.

Alla fine della lettura in memoria si ha quindi che STAR si trova nello stato 0x01, secondo stato della fase di prelievo e APP0 contiene il primo byte della nuova istruzione 0x01, che corrisponde proprio all'istruzione NOP, del programma che il processore sta eseguendo.

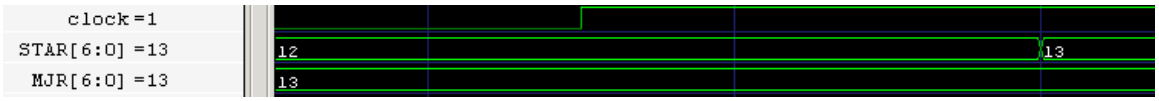


All'arrivo del clock STAR va nello stato 0x02, che corrisponde al terzo stato della fase di Prelievo e si aggiorna il registro OPCODE, con il contenuto di APP0.



All'arrivo del clock quindi si controlla con la rete combinatoria *valid_fetch* che il contenuto di OPCODE sia corretto e quindi si aggiorna STAR con lo stato successivo della fase di prelievo, altrimenti si sarebbe finiti in uno stato di blocco. Inoltre, si determina anche in che stato bisogna portarci in base al formato dell'istruzione prelevata. In questo caso essendo una NOP del formato F0 non bisogna prelevare altri operandi, quindi il processore, all'arrivo del successivo clock, si porta alla fine della fase di prelievo, nello stato 0x11.

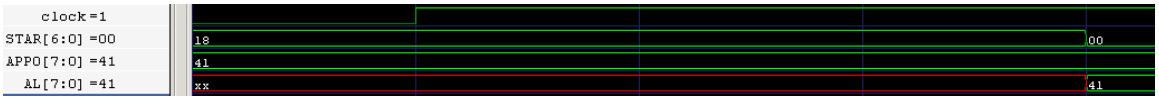
Qui, al clock, il registro MJR, viene aggiornato in base alla rete combinatoria *first execution state*, da cui si trova il primo stato di esecuzione dell'istruzione e STAR si porta allo stato successivo in cui prende, al successivo clock, il contenuto di MJR. Arrivando ad avere alla fine della fase di prelievo il valore 0x13, ovvero lo stato di esecuzione di NOP.



Questa istruzione non compie elaborazioni particolari e all'arrivo del clock successivo STAR ritorna nello stato iniziale di prelievo 0x00, da cui riparte un ciclo di lettura.

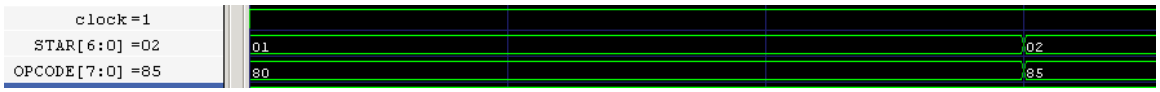
Alla fine del ciclo di lettura e dopo un paio di cicli di clock il processore si trova nella seguente situazione

In cui OPCODE contiene 0x80, cioè l'istruzione successiva del programma (MOV $0x41,%AL), e MJR lo stato successivo della fase di prelievo, che, quando verrà assegnato a STAR al clock successivo, porterà ad una nuova lettura in memoria dell'operando, specificato dal formato dell'istruzione prelevata.



Arrivando infine ad assegnare ad AL l'operando specificato nell'istruzione, ovvero 0x41, la lettera 'A' in codifica ASCII.

Da qui si ritorna alla fase successiva di prelievo in cui viene prelevata la successiva istruzione 0x85, che corrisponde proprio alla nuova istruzione (OR $0x20,%AL).



Nei successivi cicli di clock, viene inoltre, come specificato dal formato che è lo stesso dell'istruzione precedente, letto un altro byte.

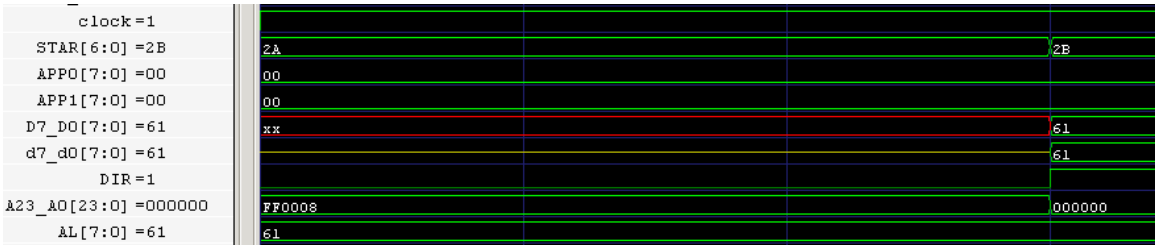
Il byte prelevato 0x20 corrisponde all'operando sorgente della nuova istruzione e questo viene posto come contenuto del registro SOURCE che servirà per tenere il valore dell'operando sorgente durante la fase di esecuzione.

In particolare, questa istruzione prevede l'OR tra l'operando contenuto in SOURCE e AL. Quindi, durante la fase di esecuzione si ottiene in AL il seguente risultato.

Che è proprio l'operazione di OR tra 0x20 e 0x41, ottenendo quindi in AL la lettera 'a'.

A questo punto entriamo nel cuore del programma con il ciclo in cui vengono stampate tutte le lettere dell'alfabeto in minuscolo.

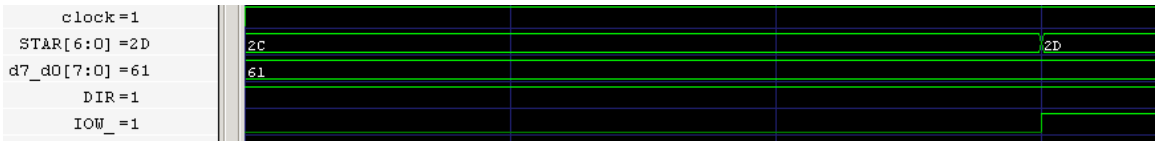
Viene prelevata la successiva istruzione (OUT %AL,0x0000). Questa istruzione è di un formato particolare, in cui il prelievo dei dati viene fatto direttamente durante la fase di esecuzione. In questa fase viene appunto fatta una lettura di due byte, essendo l'indirizzo della porta di I/O su due byte e si ottiene la seguente situazione



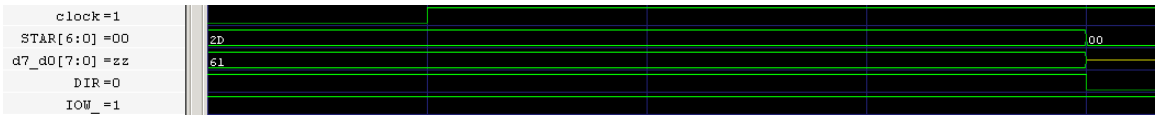
Qui si può notare che i registri di appoggio APP0 e APP1 contengono i due byte letti, che sono 0x00 entrambi, in quanto sono i byte specificati nell'istruzione, ed indicano l'indirizzo nello spazio di I/O su cui bisogna trasferire il dato.

Viene assegnato quindi l'indirizzo al registro A23_A0 (notare che sono significativi solo i primi sedici bit) e viene anche aggiornato il contenuto di D7_D0 con il contenuto di AL, ossia 'a', viene quindi posto DIR a uno, in modo da attivare come uscita i fili di dati del processore.

Nei successivi clock viene posto IOW_ prima a zero e poi ad uno.

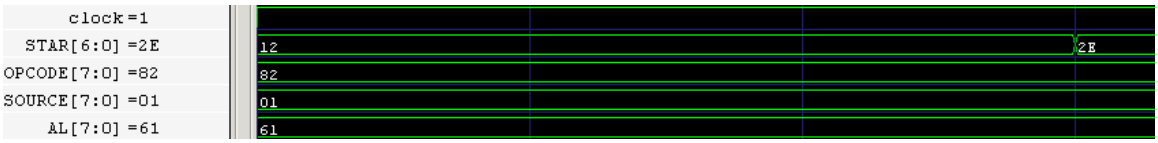


Infine, viene rimesso a zero il contenuto di DIR, togliendo in uscita il valore di D7_D0 e si ritorna nello stato di prelievo per una successiva istruzione



In questi stati il processore ha mandato in uscita il valore 0x61 e l'interfaccia ha stampato quel valore in codifica ASCII come output della simulazione, ottenendo proprio 'a', la prima lettera della seconda riga dell'output della simulazione mostrato in Fig. 20.

Viene quindi prelevata una nuova istruzione, la ADD e il corrispondente operando.

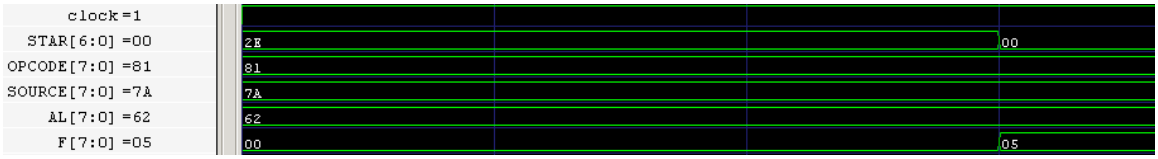


Alla fine della fase di prelievo della istruzione siamo quindi nella seguente situazione. Abbiamo in OPCODE l'istruzione 0x82, che corrisponde alla ADD e in SOURCE l'operando passato nell'istruzione.

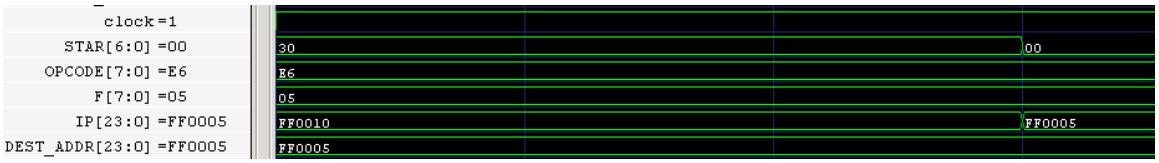
Si arriva quindi alla fase di esecuzione in cui viene aggiornato il contenuto di AL del valore voluto e si procede al prelievo della nuova istruzione (CMP $0x7A,%AL).

Alla fine della fase di prelievo, si ottiene in OPCODE il valore 0x81 e in SOURCE l'operando specificato, cioè 0x7A, il carattere 'z' in ASCII.

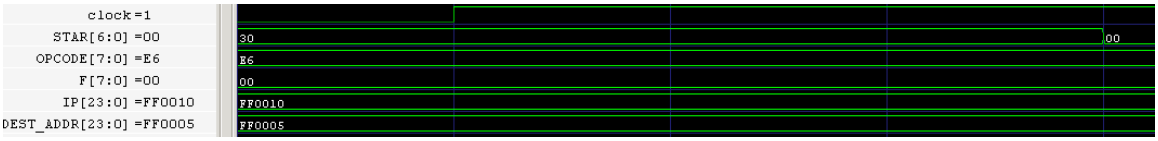
Nella fase di esecuzione dell'istruzione CMP, si ottiene l'aggiornamento dei flag.



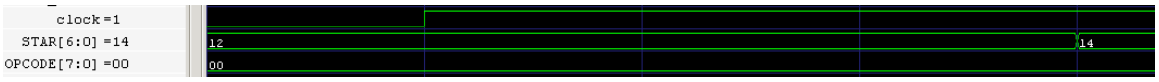
Avendo quindi settati il *carry flag* ed il *sign flag*. Questi saranno utilizzati dall'istruzione successiva (JBE loop). Alla fine della fase di fetch di quest'ultima abbiamo in OPCODE 0xE6, che indica l'istruzione JBE, e in DEST_ADDR l'indirizzo a cui saltare, in questo caso 0xFF0005, che corrisponde all'istruzione OUT vista in precedenza.



Durante la fase di esecuzione c'è la rete combinatoria *jump condition* che, tramite l'OPCODE e il registro F, controlla se la condizione dell'istruzione per saltare è rispettata. In questo avendo il *carry flag* settato, la condizione risulta verificata e quindi si salta all'istruzione OUT aggiornando il contenuto di IP.
Si torna quindi all'istruzione OUT e vengono rieseguite le stesse istruzioni, mettendo come output della simulazione tutti i caratteri dell'alfabeto in minuscolo. Dopodiché il contenuto di AL fa si che la condizione della JBE non sia rispettata, essendo 0x7B maggiore di 0x7A, e quindi si prosegue con l'istruzione successiva al ciclo (HLT).



Si può notare infatti che all'arrivo del clock il contenuto di IP non viene aggiornato in quanto il registro F contiene 0x00, che non verifica le condizioni di JBE e quindi non si modifica il flusso di esecuzione.
Si procede quindi con la prossima istruzione, prelevando il byte successivo 0x00.



Si ha quindi il prelievo dell'ultima istruzione (HLT), che all'esecuzione interrompe la simulazione e viene stampato a video l'ultimo messaggio dell'output della simulazione, mostrato in Fig. 20.

Capitolo 5

Conclusioni

Con questo lavoro ho raggiunto, dunque, la realizzazione di un ambiente di simulazione completamente funzionante per il processore sEP8. Ho aggiunto tutti gli elementi mancanti rispetto alla descrizione fornita nel libro e ho incluso tutti i moduli cruciali per il corretto funzionamento del sistema. A questo punto è possibile eseguire con successo un programma che prevede un insieme qualsiasi delle istruzioni previste dal processore e grazie all'ambiente si ha la possibilità di osservare l'andamento interno del processore tramite lo studio della waveform.
In conclusione, questo progetto rappresenta quindi un solido punto di partenza per esplorare il mondo dei processori, con la possibilità di ulteriori sviluppi introducendo concetti più avanzati tipici dei processori moderni. È possibile, infatti, introdurre il supporto all'utilizzo della tastiera per l'acquisizione di input dall'utente, il supporto alle interruzioni, il supporto alla memoria cache e ad altri meccanismi avanzati. È inoltre possibile ampliare il set di istruzioni previste dal processore.

Bibliografia

[1] Paolo Corsini. *Dalle Porte AND OR NOT al Sistema Calcolatore. Un viaggio nel mondo delle reti logiche in compagnia del linguaggio Verilog*. Pisa: Edizioni ETS. 2021.