Architettura degli elaboratori – A. A. 2019-20 – Corso A
Secondo appello – 6 febbraio 2020

Riportare su tutti i fogli consegnati in alto a destra Nome, Cognome, Matricola e Corso di appartenenza (A/B).
I risultati saranno resi pubblici via WEB appena disponibili assieme al calendario degli orali.

Domanda 1

Si fornisca il codice assembler ARM risultante dalla compilazione dello pseudocodice (assumiamo che k<M<N):

```
i=0;
found=false;
while (!found && i<N) {
   x=0;
   for(j=0; j<M; j++) {
      if big[i+j]==small[j] {  x++; }
   } // end for
   if (x<k) {  i++; } else { found = true; }
} // end while
```

Il codice deve essere derivato utilizzando le regole di compilazione discusse a lezione. Del codice risultante, relativamente all'esecuzione su un processore ARM pipeline del solo ciclo for:

- si individuino i data hazard che introducono stalli
- si individuino i control hazard che introducono stalli
- si valuti il numero di cicli di clock necessari ad eseguire una iterazione del ciclo for, assumendo che la condizione (guardia) dell'if sia vera nella metà dei casi e falsa nell'altra metà.

Domanda 2

Si consideri una rete sequenziale che è dotata di un'interfaccia con un registro in ingresso X da 32 bit e un registro in uscita Z da 1 bit, oltra al segnale di clock. La rete sequenziale riceve, in due cicli di clock successivi, un primo valore I che utilizza come indice per accedere ad una memoria interna K (reg [31:0] K [0:1023] in Verilog) ed un valore V che viene confrontato con il valore della memoria interna alla posizione K[I]. A questo punto:

- Se i valori coincidono, viene mandato in uscita un 1 e si attende un terzo valore che viene confrontato con il valore contenuto in una seconda memoria interna A (reg [31:0] A [0:1023] in Verilog), sempre alla posizione I. L'esito del confronto viene utilizzato come uscita (0 => valori uguali, 1 => valori diversi).
- Se i valori non coincidono, si invia uno 0 in uscita e ci si appresta a ripetere l'intera operazione.

Si fornisca l'automa di Moore che modella la rete sequenziale e la sua codifica come modulo Verilog.

Bozza di soluzione

Domanda 1

Assumiamo di avere come preambolo:

```
        .data
small:  .word 1,2,3,4
big:    .word 1,2,3,4,5,6,7,8
        @ n = #8
        @ m = #4
        @ k = #2
        .text
        .global main
main:   mov r0, #0      @ i
        mov r1, #0      @ found
```

Compilato (regole standard di compilazione):

```
while:  cmp r1, #1
        beq fine        @ se !found è falso, esco
        cmp r0, #8
        bge fine        @ se = n, esco
        mov r2, #0      @ x
        mov r3, #0      @ j
for:    cmp r3, #4
        bge if2
if1:    add r4, r0, r3                  @ calcola i+j
        ldr r5, =big
        ldr r6, [r5, r4, lsl #2]        @ big[i+j]
        ldr r7, =small
```

```
        ldr r8, [r7, r3, lsl #2]        @ small[j]
        cmp r6, r8
        addeq r2, r2, #1        @ ramo then: solo se r6 == r8 incremento x (r2)
        add r3, r3, #1          @ j ++
        b for                   @ nuova iterazione
if2:    cmp r2, #2
        addlt r0, r0, #1
        movge r1, #1
        b while
fine:   pop {r4-r8}
```

Notiamo che l'inizializzazione dei registri base dei vettori big e small può essere portata fuori dal compilato del ciclo, che così diventa:

```
while:  cmp r1, #1
        beq fine        @ se !found è falso, esco
        cmp r0, #8
        bge fine        @ se = n, esco
        mov r2, #0      @ x
        mov r3, #0      @ j
for:    cmp r3, #4
        bge if2
if1:    add r4, r0, r3                  @ calcola i+j
        ldr r6, [r5, r4, lsl #2]        @ big[i+j]
        ldr r8, [r7, r3, lsl #2]        @ small[j]
        cmp r6, r8
        addeq r2, r2, #1        @ ramo then: solo se r6 == r8 incremento x (r2)
        add r3, r3, #1          @ j ++
```

```
        b for                   @ nuova iterazione
if2:    cmp r2, #2
        addlt r0, r0, #1
        movge r1, #1
        b while
fine:   pop {r4-r8}
```

La parte che dobbiamo valutare è il ciclo for:

```
for:    cmp r3, #4
        bge if2
if1:    add r4, r0, r3                  @ calcola i+j
        ldr r6, [r5, r4, lsl #2]        @ big[i+j]
        ldr r8, [r7, r3, lsl #2]        @ small[j]
        cmp r6, r8
        addeq r2, r2, #1        @ ramo then: solo se r6 == r8 incremento x (r2)
        add r3, r3, #1          @ j ++
        b for                   @ nuova iterazione
if2:
```

Abbiamo una sola RAW che origina bolle (la LDR R8, … -> CMP R6, R8, introduce una bolla da 1 ciclo). Abbiamo due control hazard (BGE iniziale e B FOR finale) che introducono ciascuna una bolla da due cicli). La ADDEQ viene eseguita comunque, con o senza scrittura nei registri. Dunque in ogni caso l'iterazione richiede l'esecuzione di 9 istruzioni che richiedono 14 cicli di clock per completare a regime.

Domanda 2

Gli ingressi della rete sono il CLOCK e il valore del registro X. Dunque, gli archi dell'automa saranno etichettati da valori di questi due segnali. Come di norma, i valori degli ingressi vengono letti al posedge del segnale CLOCK. Quindi qualsiasi arco che porta da uno stato all'altro sarà etichettato da "posedge CLOCK" e da un valore per l'ingresso X ("X= …").

Gli stati dell'automa devono ricordare dove ci troviamo nella computazione:

- In attesa di un indice I da utilizzare per l'accesso alle memorie
- In attesa di un valore V da confrontare con il valore contenuto in K[I]

- In attesa di un valore da confrontare con il valore contenuto in A[I]

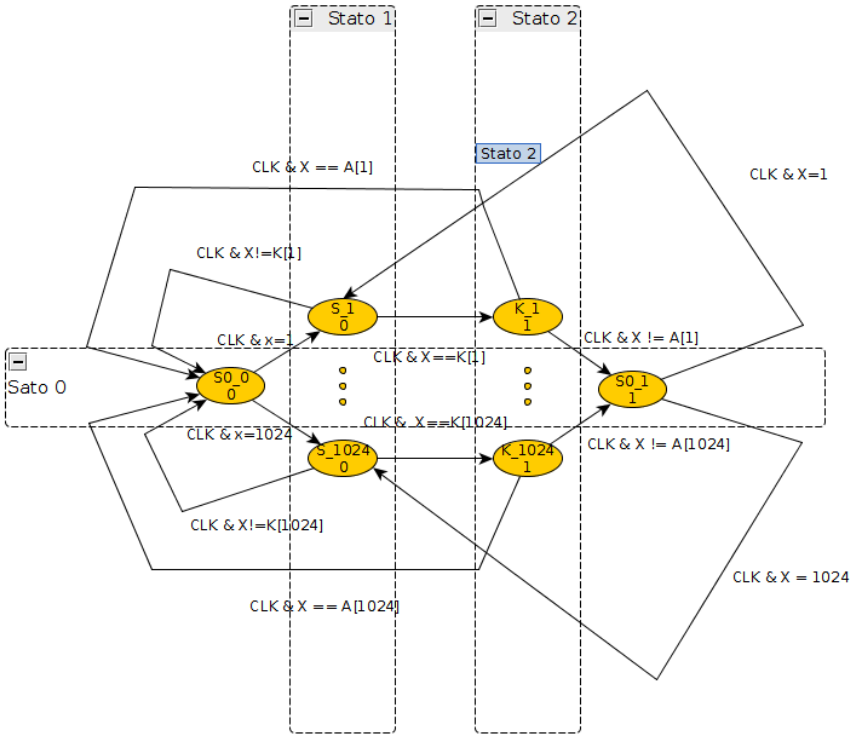
Da notare che una volta ricevuto il valore I, occorre ricordarselo per poterlo utilizzare nel secondo e terzo stato. Questo significa che lo stato iniziale è unico ma il secondo e il terzo stato sono in realtà una serie di stati distinti, uno per ognuno dei valori di I possibili (che sono 1K, in effetti).

Infine, siccome vogliamo un automa di Moore, le uscite devono essere associate agli stati. Quando valori diversi di un ingresso associati allo stesso stato determinino uscite diverse, occorre prevedere l'utilizzo di tanti stati quante sono le uscite diverse.

Con queste considerazioni, il nostro automa di Moore potrebbe avere:

- Uno stato iniziale (S0). Da qui si muove quando il clock è alto e leggendo allo stesso tempo dall'ingresso il valore dell'indice da utilizzare per l'accesso alla memoria K. Ogni ingresso diverso determina uno stato diverso di arrivo, di modo che lo stato "ricordi" il valore letto.
- Stati di lettura chiave (S1). Sono gli stati cui si arriva dallo stato inziale. Ne abbiamo uno per ognuno degli indirizzi da 10 bit che leggiamo nello stato inziale. Da questi stati usciamo quando il clock è alto e leggiamo il valore V. In questo caso ci muoviamo in un altro stato (S2), se e solo se il valore letto è uguale a K[IND], altrimenti ci muoviamo di nuovo nello stato inziale.
- Stati lettura valore (S2). Qui si legge il valore che viene confrontato con il valore della memoria A. Ce ne sono tanti quanti gli indirizzi letti al primo passo. Da questi stati si esce confrontando il valore letto con quello nella posizione IND della memoria A. Nel caso vada data una riposta 0 (i valori sono uguali), torniamo allo stato iniziale. Altrimenti (i valori sono diversi) andiamo in uno stato duale rispetto allo stato iniziale: ha le stesse transizioni in uscita, ma l'uscita associata è 1 invece che 0 come nello stato iniziale.

L'automa potrebbe essere rappresentato come segue:



La codifica in Verilog utilizza una variabile per mantenere il "tipo" dello stato e una per mantenere il valore I che distingue i diversi "stati 2" e "stati 3". Si noti come negli stati della parte sn l'uscita sia sempre 0 e in quelli della parte ds l''uscita sia sempre 1. Ci avvarremo di questa proprietà per semplificare il calcolo dell'uscita, rappresentando lo stato S0_0 con 00, gli stati S_I con 01 + IND = X, gli stati K_I con 11 + IND = lo stesso valore di prima e infine S0_1 con 10.
La logica per il cambiamento di stato è codificata mediante un **always @(posedge clock)**. La logica per il calcolo delle uscite è codificata mediante un **assign**.

```
module m(output z, input [31:0] x, input CLK);


   // stato

   reg [1:0] stato;    // tipo dello stato

   reg [9:0] ind;      // valore dell'indirizzo associato allo stato

   // memoria interna

   reg [31:0] K [0:1023];

   reg [31:0] A [0:1023];
```

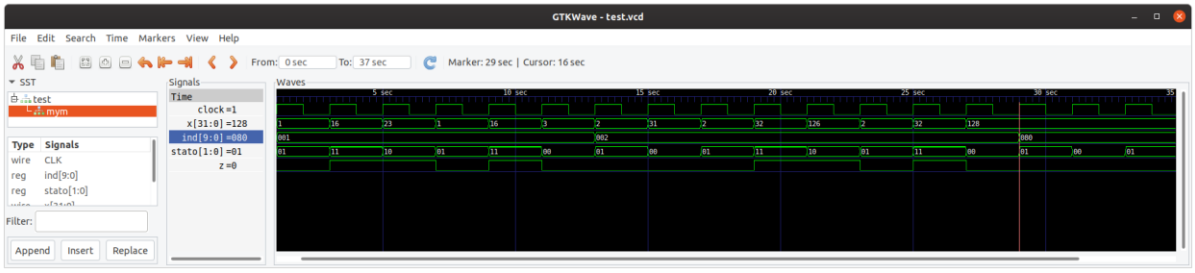
```
initial
  begin
    stato <= 2'b00;  //  stato iniziale
    ind = 0;        // giusto per il test alcuni valori delle memorie
    K[1] <= 16;       K[2] <= 32;
    A[1] <= 3;       A[2] <= 128;
  end


// logica sigma
always @(posedge CLK)
  begin
    case (stato)
      2'b00:  // primo stato, leggi indirizzo
        begin
           ind <= x & 32'd1023;
           stato <= 2'b01;
        end
      2'b01: // secondo tipo (ind contiene ormai il valore I)
        begin
           if(x == K[ind])
            stato  <= 2'b11;
           else
            stato  <= 2'b00;
        end
      2'b11: // terzo tipo (ind contiene ancora il valore I)
        begin
```

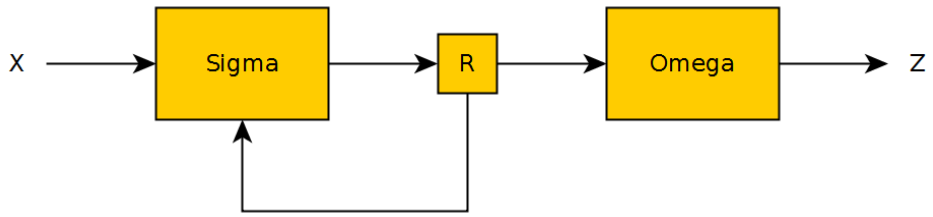
```
            if(x == A[ind])
             stato <= 2'b00;
            else
             stato <= 2'b10;
          end
        2'b10:
          begin  // stato inziale duale (come il primo ma uscita a 1)
            ind <= x & 32'd1023;
            stato <= 2'b01;
          end
      endcase
    end
  // logica omega
  assign
    z = (stato[1] == 1'b0 ? 1'b0 : 1'b1);


endmodule
```

Utilizzando un programma di test in cui vengono presentati, in cicli di clock successivi, gli input 1, 16, 23, 1, 16, 3, 2, 31, 2, 32, 126, 2, 32, 128 possiamo vedere come effettivamente l'automa calcoli quanto descritto nel testo:



Se avessimo voluto vedere in modo strutturale la rete sequenziale avremmo dovuto definire le diverse componenti dello schema classico:



In questo caso:

- R dovrebbe contenere I e lo stato di avanzamento (iniziale, lettura chiave, lettura valore). 10 bit per I ([9:0] I) e 2 bit per lo stato di avanzamento ([1:0] ST_AV)
- Sigma dovrebbe calcolare:
    a. {X, 01} nello stato iniziale
    b. {I, 11} nello stato 01 se X == K[I] oppure {I,00} se X != K[I]
    c. {I, 10} nello stato 11 se X >= A[I] oppure {I, 00} se X<A[i]
    d. {X, 01} nello stato 10
- Omega dovrebbe calcolare:
    a. 0 se siamo nello stato iniziale (00) o nello stato 01
    b. 1 se siamo nello stato 11 o 10