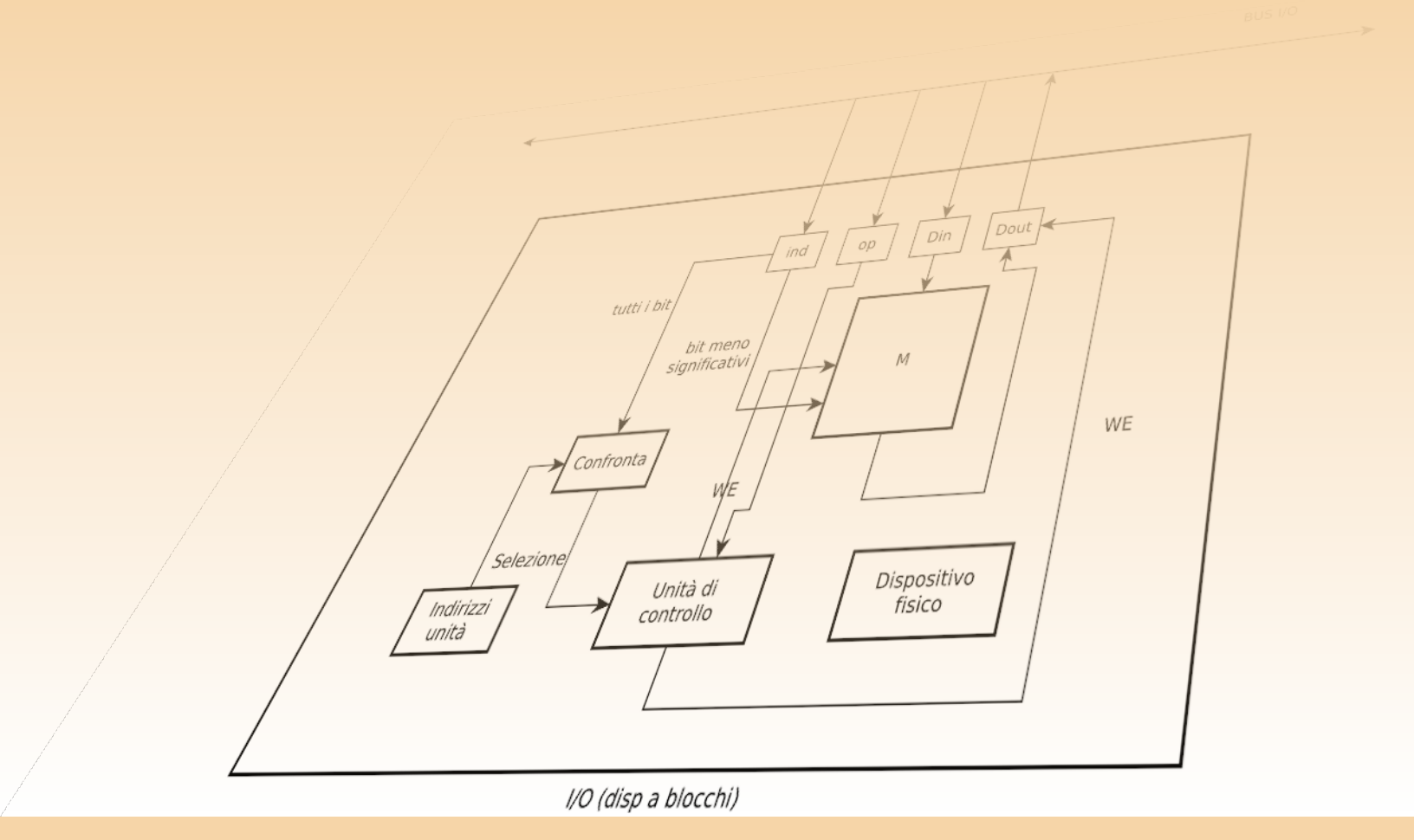
Integrazioni AE

Marco Danelutto

A.A. 2020–21



Indice

Parte I

Verilog

Capitolo 1

Verilog

Questo capitolo intende dare una breve e sommaria descrizione del sottoinsieme di Verilog necessario per la realizzazione dei progetti di Architettura degli Elaboratori. Non vuole essere una trattazione completa del linguaggio e, in particolare, non tratta tutti gli aspetti legati all'utilizzo del linguaggio come strumento per la sintesi su FPGA.

Il sottoinsieme del linguaggio viene descritto in maniera informale e facendo abbondante uso di esempi. Per una introduzione più completa del linguaggio si rimanda alla letteratura riportata in bibliografia.

> Il linguaggio Verilog, come tutti gli altri linguaggi "RTL" (Register Transfer Languages), è in linguaggio per la descrizione dell'hardware. In quanto tale permette di definire componenti che calcolano funzioni, con o senza stato, che possono successivamente essere utilizzati come moduli di altri componenti. I linguaggi RTL possono essere utilizzati per la *simulazione* o per la *sintesi* di circuiti. Nel primo caso, il programma che descrive un certo circuito (componente) viene utilizzato per simularne il comportamento e per controllare dunque che calcoli ciò per cui era stato progettato. Nel secondo caso, il programma viene utilizzato per generare le specifiche da utilizzare per realizzare un circuito che implementi fisicamente quello descritto dal programma. Le specifiche posson consistere nello schema di realizzazione di un integrato VLSI o nel file di configurazione di una FPGA.

Nelle sezioni che seguono, introduciamo prima i tipi di dati trattati in Verilog, poi i moduli ed i comandi ed infine discutiamo brevemente l'utilizzo del linguaggio Verilog per la realizzazione di esercizi e progetti quali quelli assegnati nell'ambito del corso di Architettura degli Elaboratori.

1.1 Dati

In Verilog si possono utilizzare diversi tipi di dati: costanti (literal), wire, registri, vettori e interi (variabili generiche). Nel seguito descriviamo sommariamente tutte queste diverse categorie.

1.1.1 Costanti (literal)

I numeri si possono rappresentare specificando la base ed il numero di cifre, utilizzando la notazione

<n>'<b>xxxx

dove <n> in decimale rappresenta il numero di bit, <b> è un singolo carattere che rappresenta la base (`d` per decimale, `b` per binario, `o` per ottale e `h` per esadecimale). Quindi per rappresentare 9 in binario su 4 bit si utilizzerà la costante `4'b1001`, per indicare 127 in esadecimale si utilizzerà `8'hff`, per indicare 16 in ottale si utilizzerà `6'o20`.

1.1.2 Wire

I wire sono "fili", ovvero servono per realizzare i collegamenti utilizzati per connettere componenti implementati come `module` Verilog. Un wire si può dichiarare mediante la parola chiave `wire`:

```
wire x;
wire y,z;
wire [0:7]a;
wire [3:0]b;
```

Le prime due dichiarazioni introducono tre fili di nome `x` `y` e `z`, ognuno dei quali realizza un collegamento da 1 bit.

Le ultime due dichiarazioni introducono due gruppi di fili:

- uno da 7 bit (`a`) e
- uno da 4 bit (`b`)

Gli indici fra parentesi quadre permettono di identificare quanti sono i fili (da 0 a 7, quindi 8 e da 3 a 0, quindi 4) e come si identificano i singoli bit:

- il bit 0 è il più significativo e il bit 7 il meno significativo nel primo caso ($[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]$)
- il bit 3 è il più significativo e il bit 0 è il meno significativo nel secondo caso ($[b_3, b_2, b_1, b_0]$).

Si possono riferire singoli wire di un gruppo utilizzando le parentesi quadre:

- `a[0] è il bit più significativo del wire da 8 bit di nome \verba1`
- `b[1:0]` sono i due bit meno significativi del wire `b` (b_1, b_0)

1.1.3 Registri

I registri si possono dichiarare utilizzando la parola chiave `reg` e convenzioni come quelle utilizzate per i wire per definirne la dimensione:

```
reg uno, due;
reg [0:7]unbyte;
reg [31:0]unaword;
```

`uno` e `due` sono due registri da un bit. `unbyte` è un registro da 8 bit (`unbyte[0]` è il bit più significativo). `unaword` è un registro da 32 bit (`unaword[31]` è il bit più significativo).

1.1.4 Array

Si possono definire array utilizzando sempre le parentesi quadre, poste *dopo* l'identificatore nella dichiarazione:

```
reg v[16];
reg [7:0]t[256];
```

`v` è un vettore di registri da 1 bit da 16 posizioni, `t` è un vettore di 256 registri da 8 bit ciascuno. Per assegnare un vlaore alla posizione $i-$esima del vettore `t` (l-value) o per leggerne il valore (r-value) utilizziamo la sintassi con l'indice fra parentesi quadre, come in C:

```
t[i] = ...;          ... = ... t[i] ... ;
```

1.1.5 Integers

Le variabili generiche vengono introdotte con la parola chiave `integer`. Sono implicitamente di tipo `reg` e sono interpretate come interi con segno (positivi e negativi)[1].

[1] i registri sono invece considerati sempre unsigned

1.2 Operatori

In Verilog si possono utilizzare molti degli operatori solitamente disponibili nei normali linguaggi di programmazione:

- Operatori aritmetici: `+ - * / %` (modulo)
- Operatori relazionali: `< > <= >= == !=`
- Operatori bit a bit: `~ & | ^` (bitwise not, and, or e xor, rispettivamente)
- Operatori logici: `! && ||` (not, or e and)
- Operatori di shift: `<< >>` (shift a sinistra e a destra)
- Operatore di concatenazione `{ , }` (concatena nell'ordine. `{a,b,c}` resituisce la concatenazione dei bit di a b e c, nell'ordine)
- Operatore di replicazione: `{n{item}}` (ripete n volte item come in una concatenazione. `{2{a}}` equivale a `{a,a}`.)
- Operatore condizionale: `( ? : )` (`(x<=y ? 1'b1 : 1'b0)` restituisce il bit 1 se e è minore o uguale a y, altrimenti restituisce il bit 0)

La precedenza fra gli operatori è definita come nella tabella seguente (precedenza decrescente dall'alto al basso):

Operatore	**Nome**
`[ ]`	selettore di bit o di parti
`( )`	parentesi
`! ~`	NOT logico e bit a bit
`& | ~& ~| ^ ~^`	operatori "reduce" (and, or, nand, nor, xor, nxor)
`+ -`	segno unario
`{ }`	concatenazione (`{2'B01,2'B10}=4'B0110`)
`{{ }}`	replicazione (`{2{2'B01}}=4'B0101`)
`* / %`	moltiplicazione, divisione, modulo
`+ -`	addizione, sottrazione
`<< >>`	shift destro e sinistro (`X<<2` moltiplica X per 4)
`< <= > >=`	confronti, registri e wire interpretati come numeriinteri positivi
`== !=`	uguaglianza/disuguaglianza logica
`&`	and bit a bit di una parola
`^ ~^`	xor nxor bit a bit
`|`	or bit a bit
`&&`	and logico (0 è false, il resto è true)
`||`	or logico
`?:`	condizionale (`X==Y ? 1'B1 : 1'B0`)

1.3 Componenti (moduli)

I moduli in Verilog rappresentano l'astrazione di una componente. Un modulo può rappresentare una funzione (rete logica) o una funzione con stato (rete sequenziale). I moduli possono essere ricorsivamente definiti come composizione di altri moduli.

In Verilog si possono definire componenti attraverso il costrutto `module`. Un modulo è molto simile ad una dichiarazione di procedura:

- ha un nome,

- una lista di parametri formali (di ingresso o di uscita[2])

- un corpo, che definisce come i parametri di uscita vengono calcolati a partire dai parametri in ingresso.

Tuttavia, i moduli non vengono "chiamati" bensì *instanziati* utilizzando opportuni parametri attuali.

La sintassi per definire un modulo è la seguente:

```
module <nome>(<lista parametri formali>);
  <corpo>
endmodule
```

I parametri formali possono essere dichiarati come `input` o `output`, sia nella testa della dichiarazione che fra la testa e il corpo (come nel vecchio C).

Supponiamo di avere dichiarato i moduli:

```
module Uno(output z, input x, input y);
  ...
endmodule

module Due(z, x, y);
  input x,y;
  output z;

  ...
endmodule
```

Il primo modulo usa la dichiarazione del tipo dei parametri direttamente nelle parentesi tonde. Il secondo dichiara i nomi dei parametri nelle parentesi e poi il tipo (in ingresso o in uscita) è dichiarato dopo l'intestazione. I parametri in uscita possono essere di tipo `wire` (questo è il titpo di default, non occorre specificarlo) oppure register. In questo secondo caso occorre specificare la parola chiave `reg` nella dichiarazione, sia essa fra le parentesi o successiva alla intestazione del modulo.

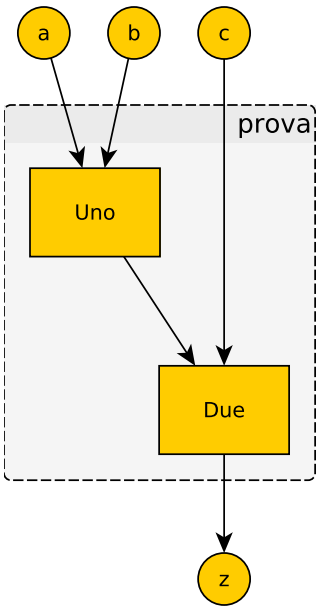
Possiamo utilizzare i nostri due moduli così definiti all'interno di in un altro modulo, instanziandoli. Per esempio:

```
module prova(zz, a,b,c);
  output zz;
  input a,b,c;

  wire da_uno_a_due_x;

  Uno istanza_di_uno(da_uno_a_due_x,a,b);
  Due istanza_di_due(zz, da_uno_a_due_x, c);

endmodule
```

In questo caso, vengono create due istanze (uno per ciascun tipo di modulo) e si usa un filo per collegare l'uscita dell'istanza del modulo di tipo Uno al primo ingresso del modulo di tipo Due. Gli altri ingressi (i due del primo modulo e l'altro ingresso del secondo) arrivano dagli ingressi del nostro modulo di tipo prova.

Sostanzialmente realizziamo uno schema tipo:

[2] si possono definire anche parametri in ingresso e uscita ma non servono per gli scopi del nostro corso



Una discussione più ampia sulla dichiarazione e sull'utilizzo dei moduli è riportata in Sez. ??.

1.3.1 "Parametri" dei moduli

I moduli possono avere "parametri" definiti per default e che possono essere variati in fase di istanziazione. I parametri corrispondono alle `#define` del linguaggio C, con alcune piccole ma sostanziali differenze. I parametri si dichiarano subito dopo la testa del modulo come

```
parameter <nome> = <valore>;
```

I parametri possono essere utilizzati nel modulo, sia per definire i parametri formali dichiarati nella testa del modulo che nel corpo. Ad esempio, possiamo definire un modulo con parametri di input e output di dimensione parametrica che modella un commutatore (due ingressi da N bit, un ingresso di controllo da 1 bit, una uscita da N bit) specificando una cosa tipo:

```
module commutatore_nbit(z, x, y, alpha);

  parameter N = 32;

  output [N-1:0]z;
  input [N-1:0]x;
  input [N-1:0]y;
  input alpha;

  assign
    z = ((~alpha) ? x : y);

endmodule
```

I parametri possono essere ridefiniti in fase di istanziazione del modulo, semplicemente anteponendo il nuovo valore del parametro preceduto da un # e fra parentesi tonde al nome dell'istanza del modulo, in fase di instaziazione. Dunque potremmo istanziare il modulo `comm`

```
  commutatore_nbit #(16) mio_commutatore(...);
```

Questa riga di codice crea un'istanza di un commutatore a due ingressi da 16 bit, anche se la dichiarazione del modulo `commutatore_nbit` definisce il parametro $N = 32$.

In caso si usino più parametri, i loro valori possono essere specificati in fase di istanziazione, nell'ordine in cui sono stati dichiarati nel modulo, fra le parentesi tonde precedute dalla gratella.

1.3.2 Moduli per il test di componenti

Mediante i moduli possiamo definire componenti e "moduli di prova" (*testbench*) ovvero moduli che servono per testare componenti o assemblaggi di componenti.

Un modulo di prova è un modulo *senza parametri formali*. All'interno del modulo di prova possiamo utilizzare alcune istruzioni (vedi sez. 7.2) che serviranno a guidare la simulazione dei componenti e a registrarne gli effetti.

La tipica struttura di un modulo di prova è la seguente:

```
module <nome>();

 // dichiarazioni di wire per ognuno degli output del componente testato
 ...
 // dichiarazioni di register per ognuno degli input del componente testato
 ...

 // istanziazione del componente
 ...
 // programma di prova : assegna valori agli input
 // (valori diversi in  tempi diversi)
 ..
endmodule
```

1.4 Blocchi e comandi

All'interno di un modulo si possono usare sostanzialmente diversi tipi di comandi, nonchè blocchi di comandi delimitati da un `begin end`. In un modulo si possono anche utilizzare, al livello più esterno, blocchi di comandi delimitati da `begin end` e qualificati dalle keyword:

- **initial**
  i comandi del blocco vengono eseguiti solo alla partenza della simulazione. Ad esempio, qualora nella dichiarazione di un modulo compaia il blocco di comandi:

  ```
  intial
    begin
      r0 = 0;
      r1 = 1;
    end;
  ```

  i comandi fra il `begin` e l'`end` vengono eseguiti all'atto dell'istanziazione del modulo e inizializzano una volta per tutte il registro `r0` a 0 e il registro `r1` a 1.

- **always**
  i comandi del blocco vengono eseguiti continuamente, come se il blocco fosse il corpo di un `(while(true))`. Ad esempio, un blocco tipo:

  ```
  always
    begin
      #1 clock = ~clock;
    end
  ```

  in un modulo dove abbiamo anche specificato

  ```
  reg clock;

  initial
    begin
      clock = 0;
    end
  ```

  farà sì che il valore del registro `clock` oscilli fra 0 e 1, mantenendo lo stato 0 (1) per una unità di tempo. Qualora volessimo mantenere alto il livello del clock per una unità di tempo e basso per 17 unità di tempo (per esempio) potremmo utilizzare un blocco always.

```
always
  begin
    #17 clock = 1;
    #1 clock = 0;
  end
```

Alla parola chiave `always` si possono associare dei modificatori. Dopo la always si può introdurre una `@` seguita da una lista di variabili fra parentesi tonde (detta *sensitivity list*), col significato: esegui il blocco always ogni volta che una delle variabili della lista cambia valore:

```
always @ (x or y)
  begin
    ...
  end
```

esegue il blocco ogni volta che cambia il valore di x o quello di y. Dalla versione 2001 del verilog al posto dell'`or` si può utilizzare la virgola:

```
always @(x, y)
```

ha la stessa semantica della notazione precedente con gli `or`. Possiamo anche utilizzare la `@` per introdurre una cosa tipo

```
  always @ (negedge x)
```

oppure

```
  always @ (posedge x)
```

che significano, rispettivamente, esegui il blocco che segue ogni qualvolta x passa da 1 a 0 o da 0 a 1.

Comandi

Dentro ad un blocco si possono utilizzare diversi tipi di comandi:

- assegnamento
  esistono diversi tipi di assegnamento: bloccante e non bloccante. L'assegnamento bloccante (simbolo =) termina prima che venga eseguito il prossimo statement (magari di assegnamento). Dunque

```
x = 1;
y = x;
```

assegna ad x il valore 1 e **successivamente** assegna ad y il valore di x, quindi 1. Nell'assegnamento non bloccante (simbolo <=) gli assegnamenti avvengono tutti allo stesso istante, ovvero la lettura delle variabili delle parti destre e il calcolo delle espressioni da assegnare avvengono contemporaneamente. Dunque

```
x <= y;
y <= x;
```

realizza uno scambio fra i valori di x e y. Esiste anche un terzo tipo di assegnamento, l'assegnamento *continuo* (simbolo `assign <parte-sn> = <expr-ds>`) la cui semantica invece è: *assegna in continuazione il risultato della parte destra alla parte sinistra. Se varia un valore utilizzato nella parte destra, rivalutala e riassegnala alla parte sinistra*. Dunque in questo caso

```
assign x = y + z;
```

assegna a x il valore della somma di y e z. Ogni volta che y o z variano, la loro somma viene nuovamente assegnata a x. In tutti i casi, si possono (solo ai fini della simulazione) introdurre dei ritardi, in unità di tempo, mediante la sintassi `#<ritardo>` utilizzata prima dell'assegnamento o durante l'assegnamento.

```
  #10 x = y + z;
```

aspetta 10 unità di tempo e quindi assegna la somma di y e z a x.

```
  x = #10 y + z;
```

calcola y+z subito ma effettua l'assegnamento ad x della somma solo dopo 10 unità di tempo.

Ai fini del nostro corso, non specificheremo come indicare l'unità di misura per il tempo e assumeremo che una unità corrisponda ad un t_p secondo la terminologia del libro di testo.

Si noti infine che il left hand side di un assegnamento *deve* essere un registro (non si possono fare assegnamenti a wire!), ma che l'assegnamento continuo può essere utilizzato per pilotare i `wire`.

- cicli
  si possono eseguire comandi in un ciclo utilizzando il for e il while, con sintassi praticamente identica a quella del C:
  - ciclo for:

```
      for(i=0; i<=N; i=i+2)
        begin
        end

```

    con limitazioni sul tipo di incremento (solo i=i +/- valore) e sul tipo di test (solo < <= > >=)
  - ciclo while:

```
      while(cond)
        begin
        end

```

    con limitazione sul fatto che il corpo deve contenere una temporizzazione;

  da notare che il `for` è sintetizzabile, ovvero può essere utilizzato nella definizione di un componente come mezzo per includere nel componente stesso un numero definito di altri componenti, mentre il `while` non è sintetizzabile, ma può essere utilizzato nei *testbench*.

- condizionali:

```
    if(cond)
      <ramo-then>
    else
      <ramo-else>

```

  con il ramo else facoltativo

- scelta multipla:

```
    case(espressione)
      valore1: begin ... end
      valore2: begin ... end
      ...
      default: begin ... end
    endcase

```

  In questo caso il caso `default` è facoltativo.

Un caso particolare di comando che si può utilizzare per la definizione di un modulo è il blocco `generate`. Un blocco generate puà essere utilizzato per generare un certo numero di istanze di componenti. L'esempio che segue fa vedere come possiamo utilizzare il `generate` per instanziare una serie di multiplexer da due ingressi di un singolo bit per realizzare un multiplexer da due ingressi da N bit ciascuno.

```
module commutatore_nbit_generative(z,x,y,alpha);

  parameter N = 32;

  output [N-1:0]z;
  input  [N-1:0]x;
  input  [N-1:0]y;
```

```
8   input  alpha;
9
10  genvar i;
11
12  generate
13    for(i=0; i<N; i=i+1)
14    begin
15      mux2 t(z[i],x[i],y[i],alpha);
16    end
17  endgenerate
18
19 endmodule
```

Alla linea 10 dichiariamo la variabile da utilizzare per la generazione (e per il relativo ciclo for). Alla riga 12 iniziamo il blocco generative. Il for all'interno del blocco crea N istanze del modulo `mux2`. La scelta dei parametri attuali delle istanze in base al valore della variabile `genvar i` permette di realizzare i collegamenti come desiderato: l'istanza i del multiplexer riceve in ingresso i bit $i-$esimi di `x` e `y`, il segnale di controllo `alpha` e produce in uscita il bit $i-$esimo di `z`.

L'implementazione di un modulo secondo il modello "behavioural" è quella che si ottiene utilizzando i comando behavioural di tipo assegnamento, `if-then-else` e `case` , in blocchi `always`, `generate` o `assign`.

Si noti che un ciclo "while(true)" di fatto corrisponde a livello di modulo ad un:

```
1  always @(*)
2    begin
3      ...
4    end
```

1.5 Direttive

I comandi che possiamo utilizzare per la simulazione sono tutte direttive che iniziano col segno dollaro `$`. Le direttive permettono di salvare la traccia di esecuzione di una simulazione, di terminare la simulazione stessa o di stampare valore delle variabili utilizzate sul terminale. Fra i comandi che possiamo utilizzare per controllare la simulazione, citiamo:

- `$dumpfile(``nomefile'');`
  permette di eseguire un dump di tutte le variabili nel programmma, in modo da poter analizzare il risultato della simulazione con un programma tipo gtkwave successivamente

- `$dumpvars;`
  permette di fare un dump di tutti i valori delle variabili del modulo nel tempo all'interno del file specificato con la `$dumpfile`. Si possono passare parametri alla `$dumpvars`. In particolare, `$dumpvars(k,top)` eseguirà il dump di tutte le variabili del modulo top e dei moduli annidati fino a a k livelli (se $k = 0$ di tutti i moduli).

- `$time;`
  restituisce il valore del tempo corrente (in unità di tempo)

- `$display(formato, lista variabili);`
  mostra il contenuto delle variabili nella lista, secondo il formato (opzionale). La stringa di formato (simile a quella della printf del C) utilizza `%d`, `%b`, `%t` e `%h` per visualizzare valori in decimale, binario, di tempo e esadecimale, rispettivamente. Dunque `$display(``X vale %b'', x)` fa vedere il valore di `x` in binario.

- `$monitor(formato,lista variabili);`
  funziona come la display, ma esegue la stampa ogni volta che le variabili cambiano valore

- `$finish;`
  termina la simulazione. Di solito il testbench (modulo senza parametri che istanzia un componente e lo testa) conclude il blocco `initial` che contiene il main della simulazione con uno statement `#NN $finish;` ovvero attende un certo intervallo di tempo e poi termina.

Ora che sappiamo anche quali sono le direttive utilizzabili in un modulo, possiamo vedere un esempio completo di modulo di test. Cosideriamo l'esempio di clock descritto nella sez. 1.4. Proviamo a testarne il funzionamento con un modulo di prova:

```
module clock();   // definiamo un modulo che testa un segnale clock

reg clock;        // registro da 1 bit, che utilizziamo per
                  // contenere il valore del clock nel tempo

initial           // alla partenza del circuito
begin
  clock = 0;      // il valore del clock e' 0
end

always            // while (true)  ... cioe' per sempre
begin
  #10 clock = 1;  // aspetta 10 unita' di tempo poi metti clock a 1
  #1  clock = 0;  // poi aspettane 1 e metti clock a 0
end               // e ricomincia il while(true)

initial           // questo e' il main del programma di test
begin             // segnala quando cambia clock
  $monitor("%t %d",$time,clock);

  #30             // aspetta 30 unita' di tempo
  $finish;        // e termina la simulazione
end
endmodule
%end{verbatim}
```

Se abbiamo creato il programma in un file "clock.vl" e compiliamo il programma con un comando:

```
iverilog clock.vl -o clock
```

successivamente possiamo eseguire il modulo di test lanciando da shell:

```
./clock
```

ottenendo l'output:

```
marcod@sony-duo-11:~/Verilog/Libro/Dispa$ iverilog clock.vl -o clock
marcod@sony-duo-11:~/Verilog/Libro/Dispa$ ./clock
                   0 0
                  10 1
                  11 0
                  21 1
                  22 0
marcod@sony-duo-11:~/Verilog/Libro/Dispa$
```

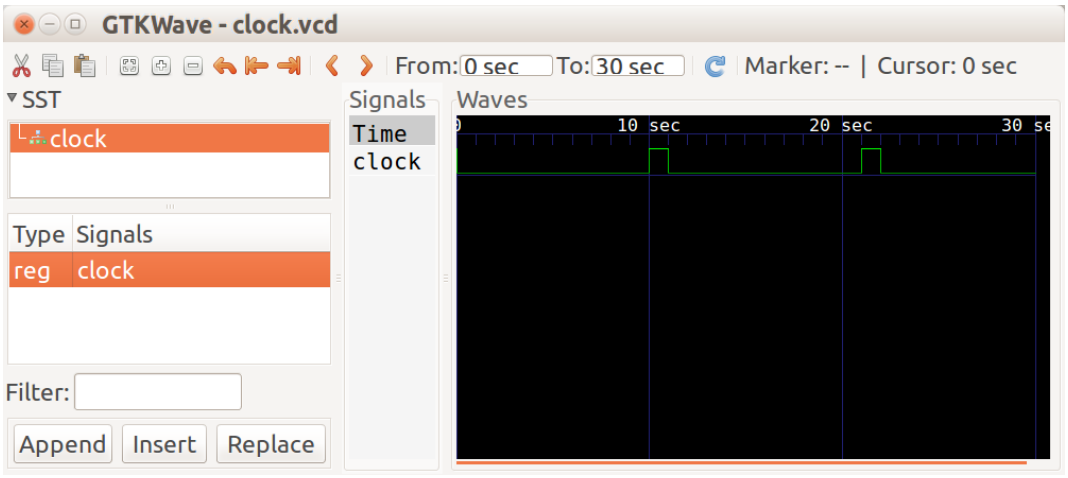
Alternativamente, possiamo visualizzare l'andamento della simulazione con `gtkwave`. Introduciamo due direttive

```
$dumpfile("clock.vcd");
$dumpvars;
```

subito dopo la `$monitor`, compiliamo e eseguiamo come fatto prima e lanciamo `gtkwave` passandogli come parametro il nome del file utilizzato nella `dumpfile`:

```
gtkwave clock.vcd
```

Si aprirà una finestra sulla quale possiamo selezionare in alto a sinistra (blocco "SST") il nome del modulo, prendere la variabile clock che comparirà in basso a sinistra nella lista della variabili del modulo ("Type Signals") e portarla nella lista in alto al centro (Colonna "Signals"), e vederne quindi l'andamento nel tempo:


GTKWave - clock.vcd
From: 0 sec
To: 30 sec
Marker: -- | Cursor: 0 sec
SST
clock
Type
Signals
reg
clock
Filter:
Append
Insert
Replace
Signals
Time
clock
Waves
10 sec
20 sec
30 se

Capitolo 2

Reti combinatorie

Queste note sono un super riassunto di quello che serve per programmare e testare (funzionalmente) componenti logici che siano reti combinatorie utilizzando Verlog (non System Verilog!).

2.1 Componenti descritti con tabelle di verità

Verilog permette di definire un componente utilizzando tabelle di verità, utilizzando un modulo di tipo `primitive`.

- L'intestazione del modulo richiede la parola chiave `primitive`, il nome del modulo e la lista che contiene il segnale in uscita (a sinistra, normalmente. Il segnale è per forza da un bit solo) e i segnali in ingresso.
- Il modulo termina con una `endprimitive`
- ciascuno dei segnali in ingresso è definito utilizzando la parola chiave `input` seguita dal nome del segnale. L'unico parmetro di uscita è definito da un identificatore preceduto dalla parola chiave `output`
- Il corpo del modulo è costituito da una tabella di verità compresa fra le keyword `table endtable`.
- Ciascuna riga della tabella di verità deve contenere:
  - tanti valori (ciascuno $\in \{0, 1, ?\}$) quanti sono gli ingressi, rappresentati nell'ordine
  - seguiti da un carattere "`:`"
  - seguito da uno 0 o un 1 che rappresenta il valore dell'uscita nel caso gli ingressi siano quelli specificati a sinistra dei `:`
  - ad esempio, una riga
    ```
    0 1 0 :  1
    ```
    a fronte dei parametri formali
    ```
    (output z, input x, input y, input w)
    ```
    indica che `z` varrà `1` quando `x = 0`, `y = 1` e `w = 0`.
- il valore `?` indica un non specificato.
- tutte le combinazioni di ingresso devono essere definite nella tabella di verità. Non è possibile abbreviare una `table` omettendo le righe con uscita `0` come invece facciamo quando disegnamo le taelle di verità con carta e penna, per semplicità.

2.1.1 Esempio: calcolo della somma di due bit e di un riporto iniziale

La somma di due bit e un bit di riporto fa `0` se tutti i bit sono `0` o se solo 2 dei tre bit sono `1`, fa `1` se esattamente uno dei tre ingressi è `1` oppure se lo sono tutti e tre. Quindi il modulo può essere scritto come segue:

```
primitive fa_somma(output s, input r, input x1, input x2);

   table
      0 0 0 : 0 ;
      0 0 1 : 1 ;
      0 1 0 : 1 ;
      0 1 1 : 0 ;
      1 0 0 : 1 ;
      1 0 1 : 0 ;
      1 1 0 : 0 ;
      1 1 1 : 1 ;
   endtable

endprimitive
```

2.1.2 Esempio: multiplexer con due ingressi da 1 bit

In questo caso possiamo scrivere la tabella di verità in modo compatto utilizzando valori di ingresso "don't care" (non specificati) rappresentandoli con il simbolo ?.

```
primitive mux2x1(output z, input c, input x1, input x2);
   table
      0 0 ? : 0 ;
      0 1 ? : 1 ;
      1 ? 0 : 0 ;
      1 ? 1 : 1 ;
   endtable

endprimitive
```

Il fatto che occorra specificare un'uscita per tutte le combinazioni degli ingressi impedisce di tralasciare le righe con uscita pari a 0. Il modulo che segue compila ma non funziona, in quanto per le uscite non speicificate anzichè 0 si osserverà un'uscita col valore speciale `x` (non specificato).

```
// QUESTO NON FUNZIONA (INIZIO)
primitive mux2x1(output z, input c, input x1, input x2);

   table
      0 1 ? : 1 ;
      1 ? 1 : 1 ;
   endtable

endprimitive
// QUESTO NON FUNZIONA (FINE)
```

2.2 Componenti descritti con espressioni booleane

Per definire un componente utilizzando expressioni dell'algebra booleana, utilizziamo moduli introdotti dalla keyword `module` invece che `primitive`. Nel corpo del modulo utilizzeremo il comando di "assegnamento continuo" `assign` che, seguito da uno statement di assegnamento, assegna il valore del risultato dell'espressione a destra dell'uguale alla variabile a sinistra dell'uguale *ogni volta che cambia uno dei valori di ingressi coinvolti nell'espressione a destra dell'uguale*. Per esempio `assign x = y;` assegna il valore di `y` a `x` ogni volta che `y` cambia. Si possono utilizzare operatori che rappresentano le operazioni booleane AND, OR e NOT:

Operatore	**Rappresentazione verilog**
and (bitwise)	&
or (bitwise)	|
not	~
and (logical)	&&
or (logical)	||
not	!

(da notare che per valori da 1 bit gli operatori bitwise sono equivalenti a quelli logici)

Differentemente dai moduli `primitive` in un `module` possiamo definire un numero arbitrario di uscite. Per ognuna delle uscite dovremmo utilizzare uno statement `assign` separato.

Per convenzione[1], le uscite sono elencate per prime nella lista dei parametri del modulo `module` (esattamente come avviene per l'unica uscita di un modulo `primitive`).

Infine, nel calcolo dell'espressione da assegnare che si trova a destra del simbolo = possiamo utilizzare l'espressione condizionale `( cond ? then : else)` come in C/C++.

2.2.1 Esempio: calcolo della somma di due bit e di un ri- porto iniziale

```
module somma(output riporto, output z,
             input riportoiniziale, input x, input y);

  assign

    z = (~riportoiniziale & ~x & y) |
        (~riportoiniziale & x & ~y) |
        (riportoiniziale & ~x & ~y) |
        (riportoiniziale & x & y);

  assign

    riporto = (~riportoiniziale & x & y) |
              (riportoiniziale & ~x & y) |
              (riportoiniziale & x) ;

endmodule
```

In questo esempio osservate due cose:

- il modulo definisce due bit di uscita, non uno solo, e
- nel calcolo del riporto abbiamo considerato con il terzo termine in OR sia la penultima che l'ultima riga della tabella di verità corrispondente, visto che le righe differiscono per il solo input `y` e dunque è come se avessimo raccolto un $(\overline{y} + y)$

Inoltre, avremmo potuto codificare in modo più compatto la parte del riporto. Guardando la tabella di verità del riporto:

```
primitive fa_riporto(output s, input r, input x1, input x2);

   table
      0 0 0 : 0 ;
      0 0 1 : 0 ;
      0 1 0 : 0 ;
      0 1 1 : 1 ;
      1 0 0 : 0 ;
      1 0 1 : 1 ;
      1 1 0 : 1 ;
      1 1 1 : 1 ;
   endtable

endprimitive
```

possiamo osservare che il riporto è 1 quando:

- il riporto iniziale è 0 e entrambi gli ingressi sono 1, oppure
- il riporto iniziale è 1 e almeno uno degli ingressi è 1.

questo si può tradurren nel `module somma` descritto poche righe sopra a questa sostituendo la seconda `assign` con questo codice:

```
assign riporto = (riportoiniziale == 0 ? (x & y) : (x | y);
```

[1] non è necessario, ma conviene seguire la convenzione nel nostro codice

2.3 Componeneti descritti in modo strutturale

Il terzo e ultimo tipo di componenti che consideriamo sono quelli descritti in modo "strutturale" ovvero come reti di altri componenti predefiniti come `primitive`, `module` o a loro volta in modo strutturale.

Per definire un componente come rete di sottocomponenti occorre:

- definire in modulo `module` con un proprio nome e la sua lista di segnali in uscita e in ingresso;
- definire tanti `wire` quanti sono i collegamenti necessari fra un segnale in uscita da uno dei moduli componenti e un segnale in ingresso di un altro modulo componente
- dichiarare un'istanza di modulo per ognuno dei moduli componenti. Le istanze di modulo si dichiarano utilizzando come tipo il nome (identificatore) utilizzato nella definizione del modulo[2], un identificatore che da' il nome all'istanza e una lista di parametri attuali, che verranno collegati nell'ordine ai parametri formali del modulo.
- utilizzare i nomi dei segnali in ingresso e uscita (parametri formali) e degli eventuali `wire` così definiti come parametri attuali delle istanze dei moduli, rispettando la semantica dei collegamenti che vogliamo implementare.

La struttura di un modulo così fatto sarà dunque qualcosa tipo:

```
module NOME(....);

  wire ...;

  NOMETIPOMODULO ID1(...);
  ...
  NOMETIPOMODULO IDK(...);

endmodule
```

2.3.1 Esempio: sommatore di due bit

Utilizziamo come componenti i due moduli `primitive` definiti nella sezione 2.1.1 e 2.2, ovvero i moduli `fa_somma` e `fa_riporto`.

Entrambi i moduli prendono in ingresso gli stessi parametri in ingresso del modulo che calcola la somma, ma uno calcola il bit di risultato e l'altro calcola il bit di riporto.

```
module fulladder(output riporto, output risultato,
                 input riportoiniziale, input x1, input x2);

   fa_riporto m1(riporto, riportoiniziale, x1, x2);
   fa_somma   m2(risultato, riportoiniziale, x1, x2);

endmodule
```

2.4 Esempio: sommatore di due numeri da 2 bit

Supponendo di avere a disposizione un modulo `fulladder` come quello definito nella sezione 2.3.1, possiamo ottenere un `fulladder2` che opera su numeri a due bit collegando il riporto in uscita di un `fulladder` che somma i due bit meno significativi dei due ingressi e il riporto iniziale (presumibilmente `0`) ad un `fulladder` che somma i due bit più significativi dei due ingressi al riporto dell'altro `fulladder`, generando il bit più significativo del risultato **e** il bit di riporto finale (vedi schema in Fig. 2.1).

Tenendo presente che segnali da più di un bit si possono dichiarare facendo precedere l'identificatore da una coppia di interi fra parentesi quadre, separati da `:`, che rappresentano il valore del primo e dell'ultimo indice da utilizzare per accedere i singoli bit (vedi sezione 2.5), il modulo può essere definito come segue:

[2] l'identificatore che è stato utilizzato fra la parola chiave `primitive` o `module` e la lista dei parametri formali

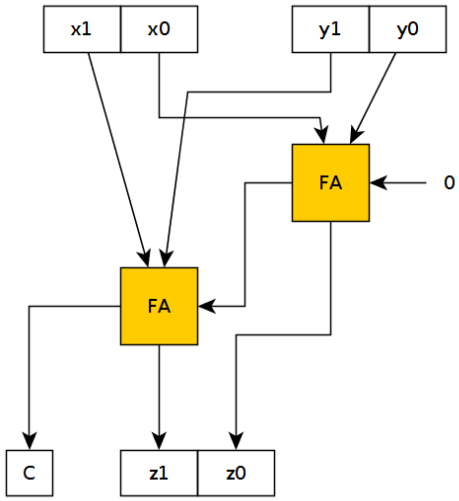


Figura 2.1: Addizionatore di due numeri da 2 bit costruito utilizzando due addizionatori di numeri da 1 bit con riporto

```
module add2(output riporto, output [1:0]somma,
            input ripin, input [1:0]x1, input [1:0]x2);

   wire     rips;

   fulladder fa0(rips, somma[0], ripin, x1[0], x2[0]);
   fulladder fa1(riporto, somma[1], rips, x1[1], x2[1]);

endmodule
```

Si noti il `rips` che serve unicamente a collegare il riporto in uscita dal primo modulo al riporto in entrata al secondo modulo. Il nome del wire compare quindi

- al posto del parametro attuale che corrisponde al formale riporto oin uscita del primo modulo, e
- al posto del parametro attuale che corrsiponde al formale riporto in ingresso del secondo modulo.

2.5 Variabili e costanti

2.5.1 Variabili da più di un bit

In Verilog possiamo dichiarare variabili che rappresentano registri (stato interno) mediante la parola chiave `reg`, e fili (collegamenti fra moduli) mediante la parola chiave `wire`. Ai registri possono essere assegnati valori nel programma di test, mentre dei `wire` ha senso solo leggere che valore portano o utilizzarli per collegamenti. Qualsiasi dichiarazione di una variabile, sia `reg` che `wire` ottenuta indicando solo l'identificatore da utilizzare definisce valori da 1 bit:

- `wire x;`
  definisce un filo capace di trasportare un singlo bit,
- `reg y;`
  definisce un registro capace di memorizzare un singolo bit.

Qualora volessimo utilizzare più bit (in un registro o in un filo) dobbiamo far precedere l'indentificatore da una coppia di parentesi quadre che al loro interno contengono un indice che rappresenta il bit più significativo e un indice che rappresenta il bit meno significativo, separati dal simbolo `:` (vedi Fig. 2.2). La possibilità di utilizzare un

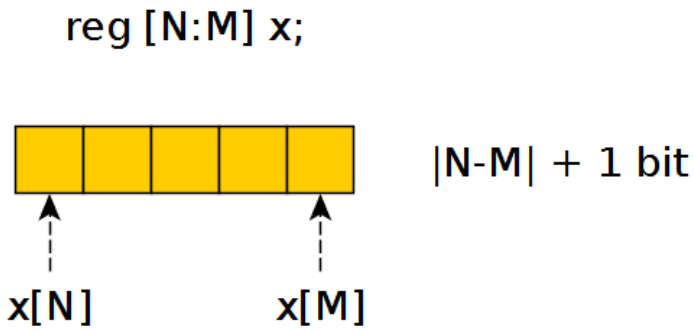


Figura 2.2: Dichiarazione di un registro da più bit

formalismo che permette di indicare range di indice diversi offre possibilità che si adattano a qualunque esigenza di rappresentazione degli indici:

- `reg [7:0] b1;`
  dichiara un registro da 8 bit (valore assoluto di (7-0) più 1). Il bit più significativo è `b1[7]` e quello meno significativo è `b1[0]`. In questo caso l'indice indica il peso del bit: se `b1[i]==1` allora il suo peso è 2^i altrimenti il suo peso è 0.

- `reg [0:7] b;2`
  dichiara un registro da 8 bit (valore assoluto di (7-0) più 1). Il bit più significativo è `b1[0]` e quello meno significativo è `b1[7]`.

- `reg [1:8] b3;`
  dichiara un registro da 8 bit (valore assoluto di (7-0) più 1). Il bit più significativo è `b1[1]` e quello meno significativo è `b1[8]`. L'indice indica la posizione del bit, secondo l'ordinamento "naturale" da destra a sinistra, partendo da 1.

2.5.2 Costanti

In Verilog, le costanti possono essere espresse indicando quanti bit occupano, la base e un numero espresso in quella base:

- `4'b0111` indica la costante 7_{10} rappresentata su 4 bit in binario. Quindi il "4" sta per 4 bit, la "b" per binario e la stringa "0111" rappresenta il valore

- `4'd7` indica la stessa costante, espressa in **d**ecimale

- `4'o07` indica la stessa costante, espressa in base **o**ttale

- `8'xff` indica il numero 255, espresso in esadecimale (he**x**adecimal)

2.5.3 Giustapposizione

Due valori da un certo numero di bit (per esempio n ed m bit) possono essere utilizzati al posto di numeri da $n+m$ bit indicandoli, nell'ordine voluto, fra parentesi graffe. L'espressione Verilog `{2'b01,4'd4}` indica il numero formato dalla giustapposizione dei valori binari 01 e 0100 quindi il valore `6'b010100`.

2.5.4 Campi di un valore da n bit

Possiamo estrarre una configurazione di bit adiacenti da un valore di n utilizzando fra parentesi quadre, dopo l'identificatore, l'indice di partenza e quello di arrivo del campo da considerare:

- se `b` fosse dichiarato come `reg [7:0] b;` (un byte), allora per prendere il nibble meno significativo potrei utilizzare `b[3:0]` e per quello più significativo `b[7:4]`
- per mascherare la parte bassa del valore `b` potrei utilizzare l'espressione `{b[7:4],4'b0000}` oltre che, naturalmente, la classica espressione che utilizza una operazione di tipo AND con una costante "maschera" ovvero `b & 8'b00001111`[3].

2.6 Moduli parametrici

A volte è utile definire moduli parametrici. Verilog permette di definire identificatori come parmetri con un certo valore all'interno di un modulo con la sintassi `parameter ID = value;`. L'identificatore può essere usato ovunque nel modulo (incluso nella lista dei parametri) per denotare il valore `value`. Il valore del parametro può essere cambiato in fase di istanziazione facendo precedere all'identificatore dell'istanza una coppie di parentesi tonde precedute dal cancelletto che contengono la lista (ordinata) dei valori da assegnare ai parametri del modulo. Se c'è un unico parametro, allora le parentesi racchideranno un unico valore.

2.6.1 Multiplexer da due ingresso con numero di bit parametrico

Un multiplexer che sceglie fra due ingressi, ciascuno di N bit può essere programmato come segue:

```
module mux(output [N-1:0] z,
           input ctrl, input [N-1:0] x1, input [N-1:0] x2);

  parameter N = 8;

  assign
    z = (ctrl == 0 ? x1 : x2);

endmodule
```

Qualora volessimo utilizzare un multiplexer che scegli fra valori da 16 bit invece che da 8 bit, dovremmo istanziarlo come segue:

```
  reg [15:0] x1;
  reg [15:0] x2;
  reg ctrl;
  wire [15:0] z;

  mux #(16) mux16(z,ctrl,x1,x2);
```

2.6.2 Ritardi

Nel testbench (programma di prova di un modulo) abbiamo utilizzato la sintassi `#3 x=0;` che significa "attendi 3 unità di tempo e poi assegna 0 a x". Possiamo denotare l'attesa di 5 unità di tempo inserendo il comando `#5;`. Possiamo anche definire un ritardo anche nelle `assign` di un modulo (facciamo precedere all'identificatore cui si assegna il valore un termine `#n` che indichi il ritardo fra la valutazione della parte destra e l'assegnamento del valore risultante alla parte sinistra dell'espressione, per modellare in modo esplicito i ritardi[4]. Noi intenderemo un `#1` come un singolo Δt. Volendo quindi associare un ritardo ai nostri moduli, potremo anche vedere che succede a livello temporale nella comparsa dei risultati dopo il cambiamento degli ingressi dei nostri moduli sotto test. Per esempio, potremmo modificare il modulo `somma` della sezione 2.2.1 come segue:

[3] in forma più compatta `b & 8'x0f`

[4] questo vale solo per la simulazione di un circuito, non per la sintesi

```
module somma(output riporto, output z,
             input riportoiniziale, input x, input y);

  assign
    // ritardo di due delta t : uno per il livello AND
    // e uno per il livello  OR
    #2 z = (~riportoiniziale & ~x & y) |
           (~riportoiniziale & x & ~y) |
           (riportoiniziale & ~x & ~y) |
           (riportoiniziale & x & y);

  assign

    #2 riporto = (~riportoiniziale & x & y) |
                 (riportoiniziale & ~x & y) |
                 (riportoiniziale & x) ;

endmodule
```

In questo modo potremo vedere come le uscite del sommatore avvengano dopo 2 unità di tempo dalla stabilizzazione degli ingressi. Senza modificare il programma che crea un addizionatore di numeri da due bit a partire da questo fulladder da 1 bit (vedi sezione 2.4), potremmo anche vedere che il risultato finale si ha dopo $4\Delta t$ per via del collegamento in cascata dei due fulladder (ciascuno con ritardo da $2\Delta t$).

Capitolo 3

Reti Sequenziali

Queste note ricapitolano quello che serve per programmare e testare (funzionalmente) componenti logici che siano reti sequenziali sincrone utilizzando Verlog (non System Verilog!). Queste note danno per scontato quanto descritto nelle note sull'implementazione di reti combinatorie in Verilog.

3.1 Verilog behavioural

Il linguaggio Verilog può essere utilizzato per programmare componenti in modo molto simile a come si programmano funzioni e procedure in un linguaggio imperativo. In particolare, un `module` può essere realizzato a partire da:

- dichiarazioni di parmetri (`parameter <nome>=<valore>`);
- dichiarazioni di `reg` (stato interno) e `wire` (collegamenti);
- istanze di altri moduli (di tipo `primitive` o `module`) utilizzanod le variabile `reg` o `wire` dichiarate precedentemente e/o le variabili che fanno parte della lista dei parametri formali del modulo come parametri attuali dell'istanza del modulo;
- blocchi di tipo `initial`, i cui comandi vengono eseguiti all'attivazione ("accensione") del modulo stesso;
- blocchi di tipo `always`, i cui comandi vengono eseguiti ogni volta che valgono le condizioni di attivazione dell'`always` e, in particolare:
  - per comandi `always begin ... end`, i comandi fra begin ed end sono eseguiti in continuazione. Questo tipo di blocco `always` richiede che all'interno sia presente almeno uno statement con un ritardo (ovvero con un'espressione `#...`);
  - per comandi `always @( ... lista variabili ...) begin end`, i comandi sono eseguiti ogni volta che una dele variabili nella lista cambia valore. In questo caso la lista delle variabili può contenere, separati dalla virgola o nomi di variabili o espressioni contenenti le parole chiave `posedge` o `negedge` seguite da una variabile. In questo caso l'esecuzione dei comandi nel blocco scatta non già quando varia il valore della variabile ma solo quando tale valore passa da 0 a 1 (`posedge`) o da 1 a 0 (`negedge`);
- comandi di tipo `assign`, per assegnare in modo continuo il valore di un'espressione a una variabile.

Esitono molte altre possibilità, che noi non consideriamo in questa sede, essendo queste più che sufficienti per lo scopo del corso.

I comandi che possiamo utilizzare all'interno dei blocchi appena descritti hanno una sintassi stile C e comprendono:

- **assegnamenti** che possono essere sincroni (segno =) o asincroni (segno <=). Quelli sincroni avengono in modo bloccante. La sequenza di comandi

```
a = 0;
b = a & c;
```

assegna 0 a b indipendentemente dal valore di c, perchè *prima* viene eseguito l'assegnamento di a e *poi* quello di b. La sequenza di comandi:

```
a <= 0;
b <= a & c;
```

esegue i due assegnamenti in parallelo e alla fine a vale 0 e b vale il risultato dell'and fra il valore che aveva a prima dei comandi e il valore di c.

- **condizionali** nella forma `if(...) <comando>;` o `if(...) <comando> else <comando>;`
- **scelta condizionale** nella forma

```
case(<variabile>)
<valore>: <comando>
<valore>: <comando>
...
default: <comando>
endcase
```

(dove il `default` può essere omesso)

Non esite un comando for come lo intendiamo in C. Esiste un for cosiddetto "generativo" che permette di dichiarare una serie di componenti. L'indice del for va dichiarato precedentemente come `genvar` e successivamente si può utilizzare un blocco

```
generate
  for( .... )
    begin
      ...
    end
endgenerate
```

dove all'interno del for si possono dichiarare *istanze di moduli* utilizzando nei parametri attuali delle istanze l'indice generato dal for. Per esempio, il codice che segue istanzia una serie di moduli che prendono ognuno un bit di una variabile x e generano ognuno un bit di una variabile y:

```
genvar i;
...
generate
 for(i=0; i<N; i=i+1)
   begin
     modulo m(y[i], x[i]);
   end
endgenerate
```

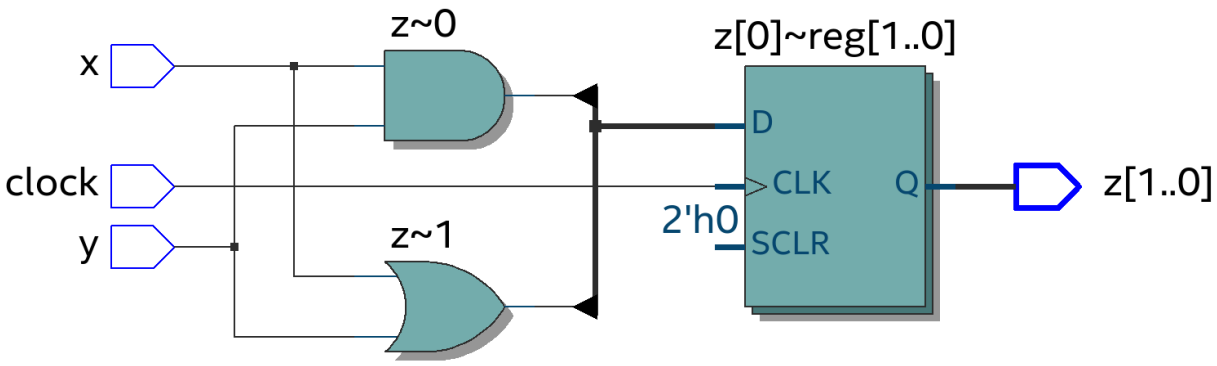
In conclusione, con il modo behavioural di programmare i componenti `module` di Verilog possiamo definire in modo programmatico:

- cosa fare per inizializzare il modulo (statement o blocco `initial`);
- cosa fare quando variano i valori in ingresso (statement o blocchi `always` e `assign`).

Nei comandi di assegnamento valgono regole diverse a seconda di come sia stata dichiarata la variabile di cui vogliamo cambiare il valore. All'interno dei blocchi behavioural `initial` e `always` si possono assegnare valori (con assegnamento sincrono o asincrono) alle sole variabili definite come `reg`. In uno statement `assign` si possono assegnare valori alle sole variabili dichiarate come `wire`. Le variabili che compaiono in `output` nella lista dei parametri formali del modulo devono essere intese come variabili di tipo `wire`. Si può comunque cambiarne il tipo utilizzando fra la parola chiave `output` e l'identificatore della variabile la parola chiave `reg`. I tool di sintesi saranno in grado di evincere il tipo corretto per la variabile di uscita. Ad esempio, un modulo tipo:

```
module m(output reg [1:0]z, input x, input y, input clock);

  always @(posedge clock)
  begin
    z[1] <= x & y;
    z[0] <= x | y;
  end

endmodule
```

genererà un registro per contenere l'uscita z, di fatto definendo un componente di logica sequenziale. La sintesi del `module` con Quartus genera infatti il seguente circuito:



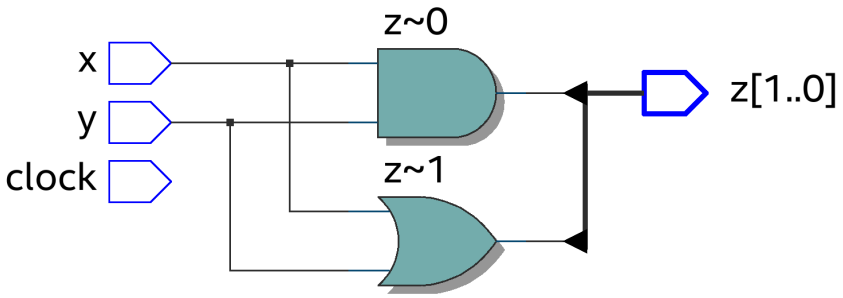
Un modulo tipo:

```
module m(output reg [1:0]z, input x, input y, input clock);

  always @(*)
  begin
  z[1] <= x & y;
  z[0] <= x | y;
  end

endmodule
```

oppure un modulo tipo:

```
module m(output [1:0]z, input x, input y);

  assign
     z[1] <= x & y;
  assign
     z[0] <= x | y;

endmodule
```

generano invece entrambi un circuito senza registri (cioè della logica combinatoria):



a dimostrazione del fatto che la sintesi è determinata, oltre che dalla dichiarazione del `reg` anche da suo utilizzo.

3.2 Reti sequenziali (modello strutturale)

Un primo modo di realizzare le reti sequenziali è quello di programmarle come reti formate dai tre componenti: le due reti combinatorie per il calcolo delle uscite e del prossimo stato interno e il componente registro di stato. Le componenti per il calcolo delle uscite e del prossimo stato interno sono reti combinatorie e possono essere programmate come visto nelle note "Reti combinarorie in Verilog", ovvero utilizzanto una o più moduli di tipo `primitive` o un singolo modulo di tipo `module`. Per la realizzazione del componente registro utilizziamo un `module` programmato in modo behavioural che rispetta la semantica del registro come vista a lezione:

```
1  module registro(output [N-1:0]z, input [N-1:0]x, input en, input clk);
2
3     // si puo' definire la lunghezza del registro come parametro del modulo
4     parameter N = 8;
5
6     // questo e' il dispositivo fisico che contiene lo stato
7     reg [N-1:0] s;
8
9     // inizializzazione (visto che non abbiamo il reset)
10    initial
11      s = 0;
12
13    // funzione di transizione dello stato interno:
14    // quando il clock va alto, in presenza di enable
15    // memorizza il valore che trovi in ingresso
16    always @(posedge clk)
17      begin
18   if(en==1)
19     s = x;
20      end
21
22    // il valore dell'uscita e' sempre il valore del registro
23    assign z = s;
24
25 endmodule // reg
```

Il modulo è parametrico. Per default definisce un registro da 8 bit (definizione del `parameter N` alla riga 4). All'interno definiamo un `reg` Verilog che conterrà lo stato del nostro registro (riga 7). Il blocco `initial` serve ad inizializzare il registro a 0 (righe 10–11). Il blocco `always @{posedge clk}` che va dalla riga 16 alla 20 controlla la scrittura nel registro: ad ogni fronte di salita del segnale di clock (posedge) se è a 1 il segnale di abilitazione in scrittura (`en`) si memorizza nel registro quanto presente in quel momento sull'ingresso. L'`assign` della riga 23 fa' sì che l'uscita del registro corrisponda sempre al suo contenuto, indipendentemente dal valore del segnale `en`.

Utilizzando questo modulo, la generica struttura di un modulo che implementa una rete sequenziale sarà una cosa tipo quella che segue:

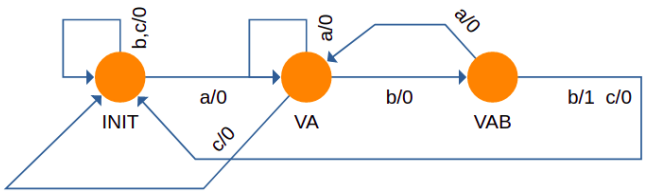
```
1  module ReteSeq(output [...]uscita, input [...]ingresso, input clock);
2
3    // dichiarazione dei wire che permettono di collegare l'uscita
4    // del registro di stato alla rete che calcola le uscite e a
```

```
5   // quella che calcola lo stato interno successivo e che permettono
6   // di collegare la rete che calcola lo stato interno successivo
7   // all'ingresso del registro di stato
8
9   wire uscitaregistro;
10  wire ingressoregistro;
11
12  // dichiarazione del modulo registro per lo stato interno
13  registro #(...) statointerno(uscitaregistro, ingressregistro, ...);
14
15  // dichiarazione del modulo che calcola il prossimo stato interno
16  prossimostato next(ingressoregistro, uscitaregistro, ingresso):
17
18  // dichiarazione del modulo che calcola l'uscita
19  // se la rete fosse di mealy avremmo
20  uscita z(uscita, ingresso, uscitaregistro);
21  // se fosse di moore sarebbe
22  // uscita z(uscita, uscitaregistro);
23
24 endmodule
```

Da notare alla linea 13 la riscrittura del parametro N del modulo registro (`#(...)`) che permette di definire un registro con l'esatto numero di bit richiesti.

3.2.1 Esempio: riconoscitore di sequenze "abb"

Consideriamo l'automa di Mealy in figura, che riconosce le sequenze "abb" all'interno di sequenze di caratteri appartenenti all'insieme `{a,b,c}`.



Lo stato iniziale ì `INIT`, `VA` è lo stato in cui ci ricordiamo di aver visto una a e `VAB` quello in cui ci ricordiamo di aver visto una a seguita da una b. Immaginiamo di codificare i tre stati `{INIT,VA,VAB}` come segue: `INIT=00`, `VA=01` e `VAB = 11`. E analogamente utilizziamo `00,01,11` per codificare rispettivamente `a,b,c`.

Con queste convenzioni, la tabella di verità per la funzione delle uscite sarà (s1 ed s0 rappresentano il bit più significativo dello stato e quello meno significativo, rispettivamente. x1 ed x0 sono i bit che rappresentano l'ingresso corrente):

s1	s0	x1	x0	z
0	0	-	-	0
0	1	-	-	0
1	1	0	1	1
1	1	1	-	0
1	1	0	0	0
1	0	-	-	0

Pertanto la funzione che calcola l'uscita potrà essere scritta come:

$$z = s_1 s_2 \overline{x_1} x_0$$

e quindi realizzata mediante il modulo Verilog:

```
module z(output zeta, input [1:0]s, input [1:0]x);

   assign zeta = s[1]&s[0]&(~x[1])&x[0];

endmodule // z
```

Per il calcolo dello stato interno abbiamo un tabella di verità diversa:

s1	s0	x1	x0	s1's0'
0	0	0	0	0 1
0	0	-	1	0 0
0	1	0	0	0 1
0	1	1	1	0 0
0	1	0	1	1 1
1	1	0	0	0 1
1	1	-	1	0 0

Possiamo utilizzare un modulo con due assign, una che controlla il primo bit dell'uscita (s1') e una che controlla il secondo (s0')[1]:

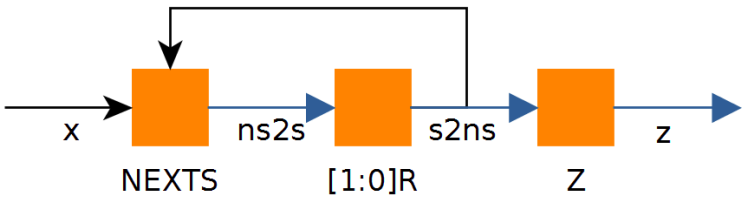
```
module nexts(output [1:0]s1,  input [1:0]s, input [1:0]x);

   assign
     s1[1] = (~s[1]) & s[0] & (~x[1]) & x[0];

   assign
     s1[0] = ((~x[1])&(~x[0])) | ((~s[1]) & s[0] & (~x[1]));


endmodule // z
```

Con questi due moduli e il modulo registro precedentemente descritto, possiamo rappresentare la rete sequenziale di Mealy che implementa l'automa secondo il modello strutturale come segue:

```
module fsm_me(output y, input [1:0]x, input clock);

   // wire necessari a connettere i componenti
   // uscita della rete combinatoria che calcola il nuovo stato
   wire [1:0] ns2s;
   // uscita del registro di stato
   wire [1:0] s2ns;

   // il registro di stato (enable sempre a 1: ogni ciclo di clock scrive)
   registro #(2) stato(s2ns,ns2s,1'b1,clock);

   // rete combinatoria che calcola il valore del prossimo stato a partire
   // dallo stato corrente e dagli ingressi (RETE DI MEALY)
   nexts prossimostato(ns2s,s2ns,x);
   // rete combinatoria che calcola l'uscita a partire
   // dallo stato corrente e dagli ingressi (RETE DI MEALY)
   z zeta(y, s2ns, x);

endmodule
```

Lo schema implementato è esattamente quello della figura che segue:

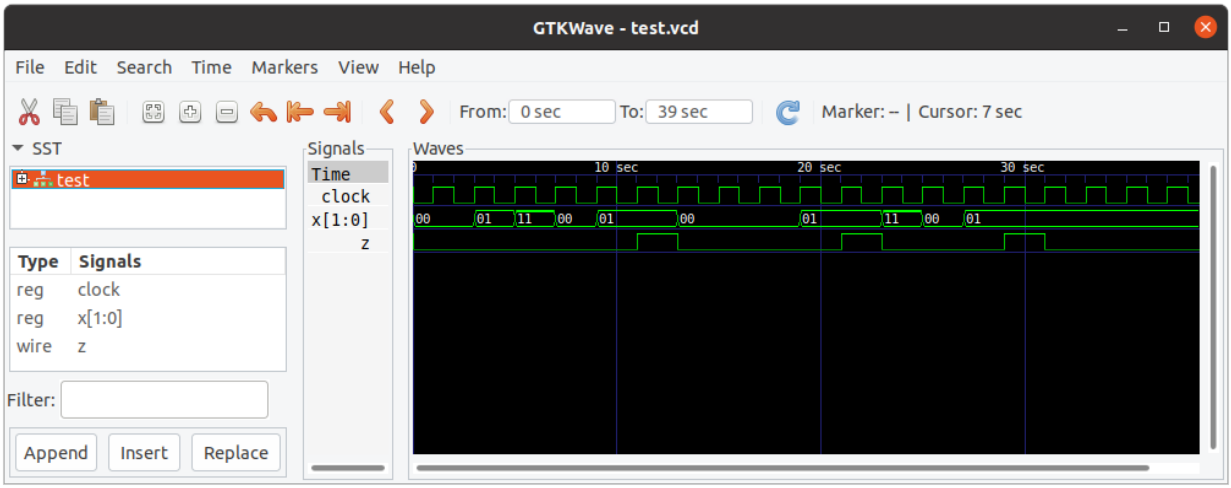
[1] l'espressione booleana per s0' è in realtà ottenuta studiando la relativa mappa di Karnaugh; diversamente avremmo dovuto avere un or di 4 termini, visto che nella colonna s0' abbiamo 4 "1"



Il comportamento della rete può essere testato utilizzando il testbench che segue:

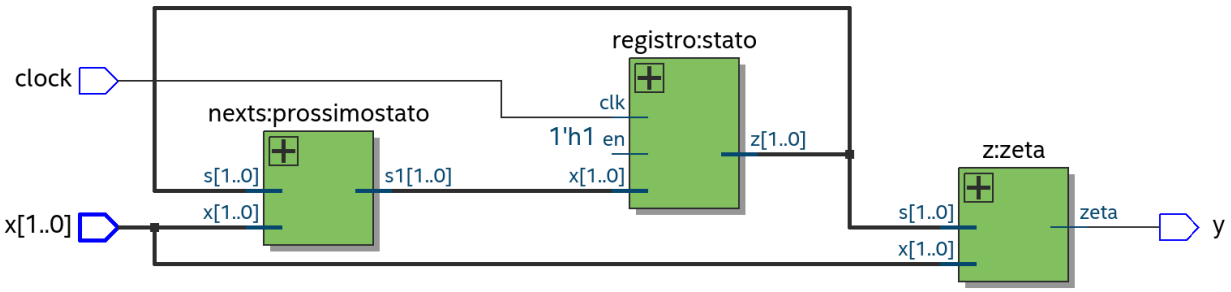
```
module test();

   // ingressi
   reg [1:0] x;
   // clock;
   reg       clock;
   // uscita
   wire      z;


   // modulo sotto test
   fsm_me mealy(z, x, clock);

   // generazione del segnale di clock
   always
     begin
  #1 clock = ~clock;
     end

   // main
   initial
     begin
  $dumpfile("test.vcd");
  $dumpvars;

  clock = 0;
  x = 0;  // x = A

  // x = B
  #3 x = 1;
  // x = C;
  // #2 x = 3;
  #2 x = 2'b11;
  // x = A
  #2 x = 0;
  #2 x = 1;
  #2 x = 1;


  // sequenza a a a b b c a b b
  #2 x = 0;
  #2 x = 0;
  #2 x = 0;
  #2 x = 1;
  #2 x = 1;
  #2 x = 3;
  #2 x = 0;
  #2 x = 1;
  #2 x = 1;

  #10 $finish;
     end
endmodule // test
```

Il risultato dell'esecuzione del testbench visualizzato con gtkwave fa vedere che effettivamente vengono riconosciute tutte e sole le sequenze abb:

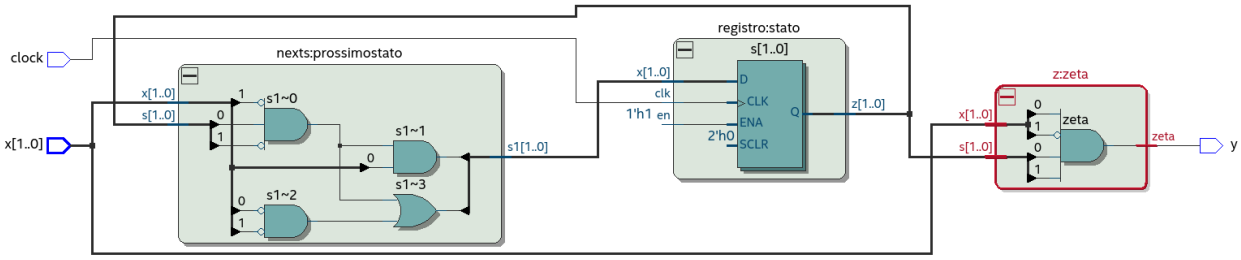


al sesto ciclo clock alto, dopo aver "visto" una a (00) e due b (01 per due clock alti) e più avanti sempre nelle stesse condizioni.

La compilazione con Quartus genera il circuito riportato in figura:



in cui si riconoscono chiaramente la logica per il calcolo del nuovo stato interno (a sinistra), qualla per il calcolo dell'uscita (and a destra) e infine il registro (hw) utilizzato per mantenere lo stato interno (al centro). "Aprendo" i vari blocchi si vede come i blocchi di logica combinatoria corrispondano alle porte utilizzate nel codice e come il `registro` sia implementato con blocchi registro hardware.



3.3 Reti sequenziali (modello behavioural)

Il modello "behavioural" permette di programmare la rete sequenziale invece che di realizzarla come collegamento di moduli di registro e di logica combinatoria esistenti. In questo caso quello che si fa' è programmare un modulo `module`:

- utilizzando un `reg` per mantenere lo stato interno,

- utilizzando un altro `reg` per rappresentare il prossimo stato interno,
- utilizzando un blocco `initial` per inizializzare lo stato interno,
- utilizzando un blocco `always @(ingressi, stato)` per calcolare il prossimo stato interno,
- utilizzando un blocco `always @(posedge clock)` per aggiornare il registro di stato con il valore calcolato come prossimo stato interno,
- utilizzando una o più `assign` per generare le uscite.

La struttura del modulo dovrebbe essere quindi qualcosa del tipo:

```
module ReteSeq(output [...]uscita, input [...]ingresso, input clock);

  reg [...] stato, prossimostato;

  intial
    stato <= ...;

  always @(posedge clock)
    stato <= prossimostato;

  always @(ingresso, stato)
    begin
    // calcolo di prossimostato
    end

  assign
    uscita = ... ;

endmodule
```

Si noti che gli assegnamenti a stato nel blocco `initial` e a prossimostato nel blocco `always` sono assegnamenti asincroni (<= invece che semplicemente =). Questo fa' sì che avvengano in parallelo. Se così non fosse, potremmo avere un comportamento non corretto con conseguenze che possono impattare anche la correttezza del calcolo dell'uscita.

3.3.1 Riconoscitore di sequenze abb

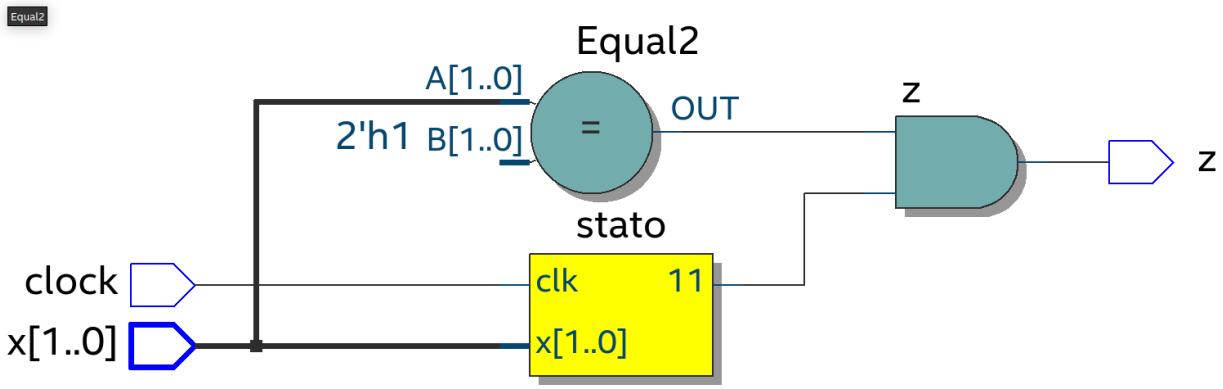
Secondo i principi appena descritti, il riconoscitore di sequenze (rete di Mealy) descritto nella sezione 3.2.1 può essere implementato come segue:

```
`define INIT 2'b00
`define VA   2'b01
`define VAB  2'b11

`define A 2'b00
`define B 2'b01
`define C 2'b11

module fsm_me(output z, input [1:0] x, input clock);

   reg [1:0] stato;
   reg [1:0] nuovostato;

   initial
     stato = `INIT;

   always @(posedge clock)
     begin
  stato <= nuovostato;
     end

   always @(x, stato)
     begin
```

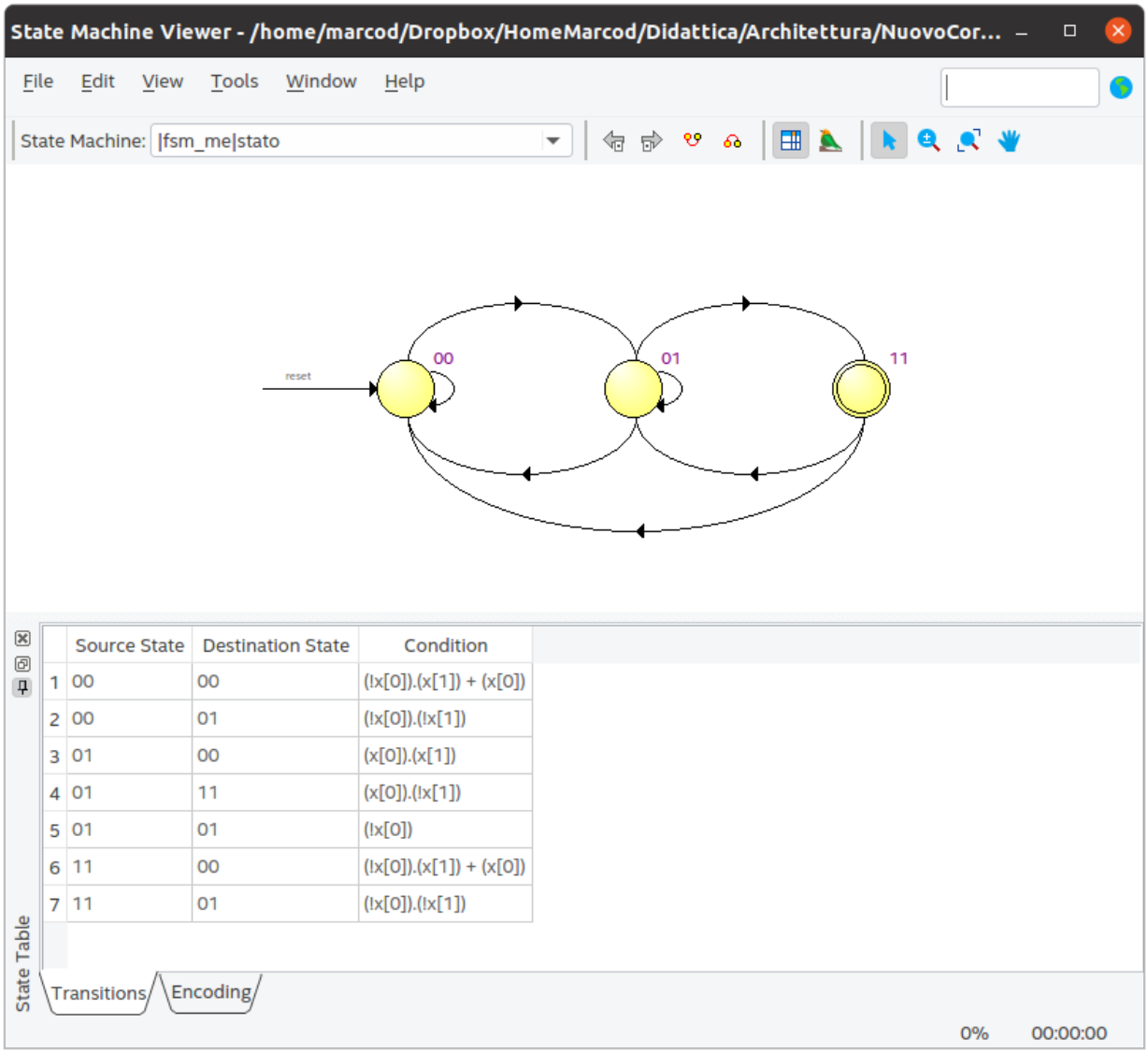
```
24   case(stato)
25     `INIT :
26       begin
27          nuovostato <= (x == `A ? `VA : `INIT);
28       end
29     `VA :
30       begin
31          nuovostato <= (x == `B ? `VAB : (x == `C ? `INIT : `VA));
32       end
33     `VAB :
34       begin
35          nuovostato <= (x == `A ? `VA : `INIT);
36             end
37   endcase // case (stato)
38      end
39
40    assign
41      z = (((stato == `VAB) && (x == `B)) ? 1 : 0);
42
43 endmodule // fsm_be
```

Si noti che abbiamo utilizzato costanti simboliche sia per la rappresentazione dei diversi stati che per la rappresentazione dei diversi caratteri.

Questa volta, Quartus sintetizza una rete con componenti nettamente diversi:



ed in particolare il blocco in basso al centro è inteso essere un automa. Aprendone la descrizione come automa vediamo questo:



	Source State	Destination State	Condition
1	00	00	(!x[0]).(x[1]) + (x[0])
2	00	01	(!x[0]).(!x[1])
3	01	00	(x[0]).(x[1])
4	01	11	(x[0]).(!x[1])
5	01	01	(!x[0])
6	11	00	(!x[0]).(x[1]) + (x[0])
7	11	01	(!x[0]).(!x[1])

che è esattamente (e correttamente) l'automa da cui eravamo partiti, derivato però dal codice Verilog del listato.
i

Capitolo 4

Materiale sul libro di testo

In questa sezione mettiamo in evidenza cosa si può utilizzare del libro di testo [2] e cosa invece differisce sostanzialmente, essendo peculiare del System Verilog e non incluso nel Verilog standard.

Parte	**Note**
Reti combinatorie: sez 4.1, 4.2 e 4.3 (esclusa 4.2.8)	Questa parte può essere utilizzata quasi così com'è. L'unica vera differenza fra il System Verilog e il Verilog è l'utilizzo della parola chiave `logic` al posto delle due parole chiave `reg` e `wire` del Verilog. `logic` indica una variabile e il suo tipo (registro o wire) è derivato dall'uso che se ne fa. `iverilog` accetta la dichiarazione `logic` ma apparentemente la utilizza come se fosse una dichiarazione di registro. La Sez 4.2.8 non va considerata, visto che on abbiamo trattato i valori `x` e `z` Gli esercizi con le `tristate` non li abbiamo discussi.
Registri, costrutti behavioural e reti sequenziali: Sez. 4.4, 4.5, 4.6	In questa parte il libro utilizza pesantemente i nuovo costrutti `always_comb` e `always_ff` che non fanno parte del Verilog. La maggior parte degli esempi non compila con `iverilog`. Caso per caso andrà considerato del codice equivalente che però rispetti la sintassi Verilog.
Sez. 4.7	Sostanzialmente da non considerare. I tipi di dato da utilizzare sono descritti nella dispensa
Sez. 4.8	Verilog utilizza la sintassi leggermente diversa presentata nelle dispense.
Sez. 4.9	La struttura dei testbench è la stessa del Verilog, ma alcuni costrutti sono introdotti nel System Verilog. Da utilizzare la sintassi delle dispense.

Capitolo 5

Sintesi

Verilog può essere utilizzato per derivare la progettazione di un componente fisico che implementi il circuito descritto dal programma. Questo processo viene di solito indicato con il termine *sintesi*. Esistono tutta una serie di strumenti che permettono di utilizzare Verilog (o altri linguaggi RTL) per la sintesi di circuiti, che possono essere divisi in due classi fondamentali:

- gli strumenti per la programmazione di FPGA, e
- gli strumenti per la progettazione VLSI.

5.1 Programmazione di FPGA

Gli strumenti di programmazione delle FPGA generano, a partire dal sorgente Verilog, una configurazione per una specifica FPGA (Field Programmable Gate Array), ovvero per un dispositivo che contiene da decine di migliaia a centinaia di milioni di *celle*, organizzate in griglie regolari, ciascuna delle quali può essere configurata in tre diversi modi:

- per calcolare una funzione di un piccolo numero di bit (normalmente da 3 a 7) che calcola un singolo bit,
- per implementare un singolo bit di memoria (latch o flip flop),
- per eseguire il routing di qualcosa calcolato dalle celle adiacenti verse altre celle adiacenti.

Ogni modello di FPGA dispone di celle diverse sia per numero che per disposizione e per capacità di calcolo. FPGA "piccole", possono avere qualche migliaio di delle. Quelle più grosse arrivano a decine di milioni di celle. Inoltre, le FPGA hanno di norma al loro interno colonne di celle "dedicate" ovvero "macro" celle (notevolmente più grosse delle celle normali) che possono eseguire semplici operazioni aritmetiche (normalmente in virgola mobile) tipo *multiply-and-add*[1] oppure che implementano piccoli moduli di memoria di qualche Kbyte. La presenza di queste macro celle permette di realizzare circuiti più complessi, a patto di riuscire e implementare nel resto delle celle la logica che permette di alimentare i moduli di calcolo o di calcolare i valori da scrivere nei moduli di memoria.

Il risultato dell'esecuzione del processo di sintesi di un circuti con gli strumenti di programmazione delle FPGA sarà dunque un file di configurazione (una specie di file di boot della FPGA) che, quando caricato dal componente FPGA la programma in modo che le diverse celle e macro celle si comportino come necessario per implementare il circuito da cui siamo partiti. Parte integrante del file di configurazione (normalmente chiamato *bitstream*) è la definizione di quali piedini del chip FPGA vengano utilizzati per i segnali di ingresso, di uscita e per importare segnali particolari, tipo il segnale di clock.

I maggiori costruttori di FPGA mettono a disposizione strumenti diversi per la programmazione dei loro chip. Altera ha Quartus, Xilinx ha Vivado e in entrambi i casi esiste una versione che non necessita di licenza e che può essere utilizzata per progetti di limitate dimensioni e che producono file di configurazione per le FPGA di gamma bassa (quelle più piccole ed economiche). E' da tenere in conto che il processo di generazione del file

[1] tipica operazione per calcolo scientifico: prende due operandi, ne calcola il prodotto e lo somma a un registro che, partendo da 0, accumula la somma di tutti i prodotti calcolati fino a quel momento

di configurazione comprende fasi che di per sè sono molto complesse (gli algoritmi sono nella classe NP) che richiedono l'applicazione di diversi tipi di euristiche. Ad esempio, come mappare l'insieme di celle necessarie sull'insieme di celle disponibili è un problema di graph placement dimostrabilmente non polinomiale nel caso in cui si cerchi la soluzione ottimale. Questa situazione ha due tipi di conseguenze che vale la pena di citare:

- il tempo di esecuzione del processo di sintesi è notevole. Semplici circuiti possono richiedere uno o più minuti di calcolo per produrre il file di configurazione su un processore stato dell'arte per laptop/server. Circuiti più complessi richiedono ore di calcolo;

- come sempre quando si usano euristiche, un eventuale ottimizzazione "a mano" può produrre risultati migliori sia in termini di occupazione delle celle sulla FPGA (usarne meno) che in termini di performance (ottenere una configurazione che implementa il circuito in modo più veloce).

5.2 Programmazione di VLSI

[2] Gli strumenti di progettazione di circuiti VLSI (Very Large Scale Integration) permettono di progettare tutte le operazioni necessarie a costruire un chip che implementi il circuito descritto dal programma Verilog. Partendo dal programma Verilog si genera quindi una *netlist* dei componenti necessari (porte logiche, blocchi predefiniti (e.g. ALU o memorie), etc.) e dei relativi collegamenti. Quindi si passa alla progettazione del layout sul chip (disposizione dei componenti sulla superficie di silicio disponibile), alla generazione delle maschere che servono per le fasi di produzione del chip stesso. Il risultato finale è la produzione di un chip che implementa il circuito. Questo processo, differentemente da quello per la programmazione delle FPGA è una cosa i cui tempi si misurano in giorni o mesi. Ciascuna delle fasi è soggetta a procedure di verifica dei risultati ottenuti che possono richiedere molto più tempo della fase stessa di produzione di quei risultati. L'intero processo è estremamente costoso e solo i grandi produttori si possono permettere di intraprenderlo.

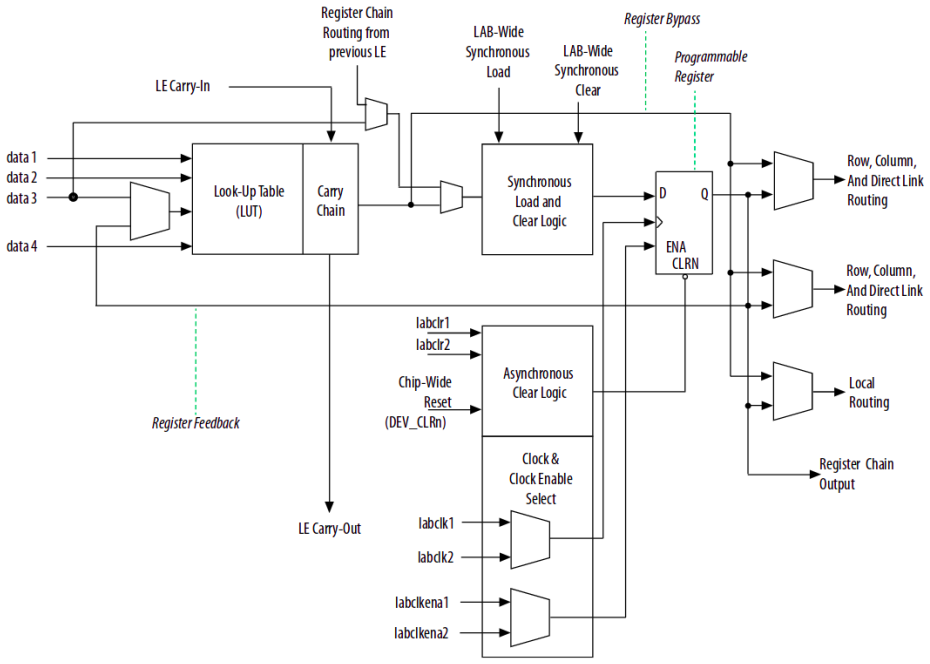
5.3 Esempio di sintesi con Quartus Lite Intel

Facciamo vedere a solo scopo illustrativo come posso ottenere un l'implementazione di un circuito descritto in Verilog su una FPGA. Illustriamo solo i passi necessari per vedere come il circuito verrà implementato e non diciamo nulla di come in effetti produrre il file di configurazione, che richiede azioni più specifiche e che dipendono anche dal tipo di dispositivo FGPA considerato.

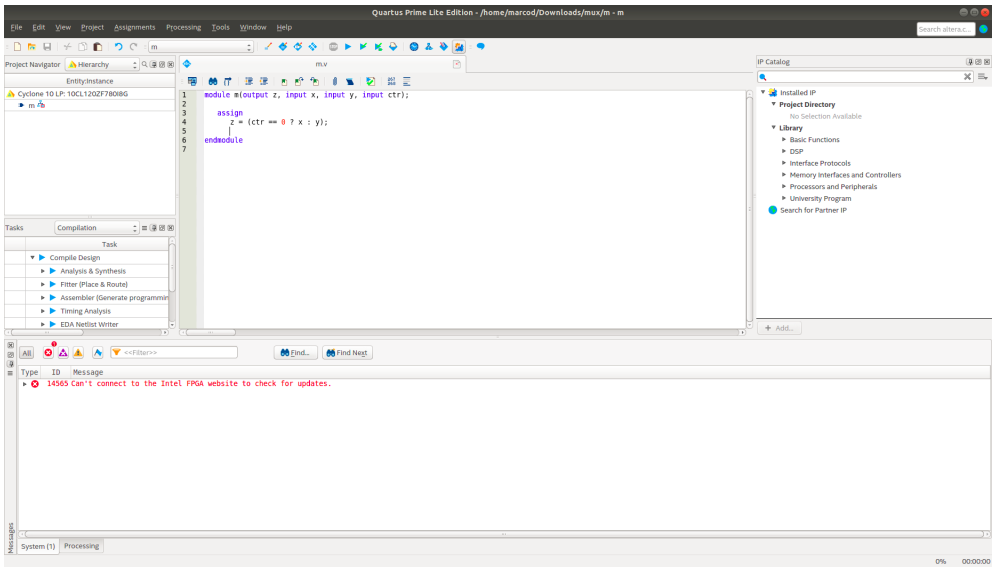
Supponiamo di voler implementare un semplice circuito mutiplexer, che accetta 2 ingressi da 1 bit e produce una uscita di 1 bit. L'uscita riporta il segnale presente su uno dei due ingressi scelto a seconda della configurazione di un ulteriore ingresso di controllo.

Il primo passo è quello di aprire il software di sviluppo e creare un progetto. Nella fase di creazione del progetto viene richiesto quale tipo di linguaggio si utilizza (Verilog o VHDL, per esempio) e quale FPGA vogliamo utilizzare. In questo esempio abbiamo scelto di utilizzare una Cyclone 10 con 119088 celle, 576 macro celle moltiplicatore e ca. 4M bit in blocchi di memoria. A titolo informativo, il singolo "logic element" (ovvero cella) di questa FPGA è fatto cosi:

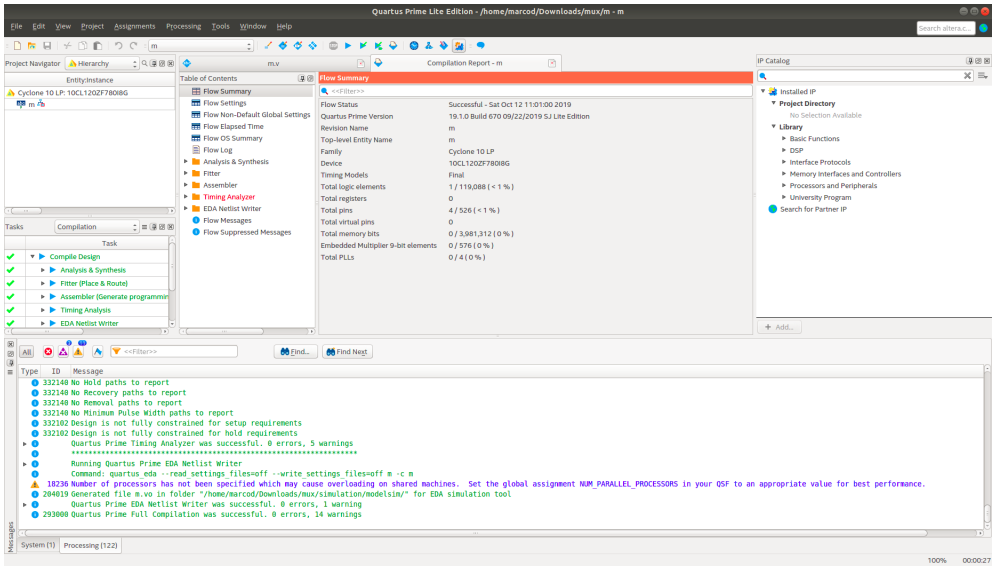
[2] La trattazione dell'argomento qui è assolutamente superficiale e ha il solo scopo di dare un'idea generale della differenze del processo rispetto alla sintesi per FPGA.



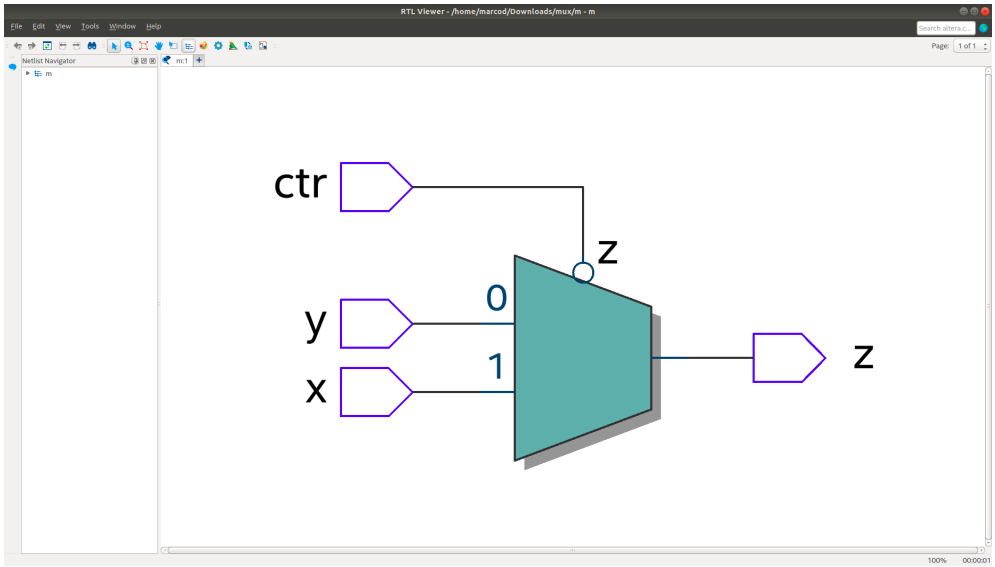
La schermata che segue riporta la configurazione iniziale, con il file Verilog del multiplexer (modulo "m").



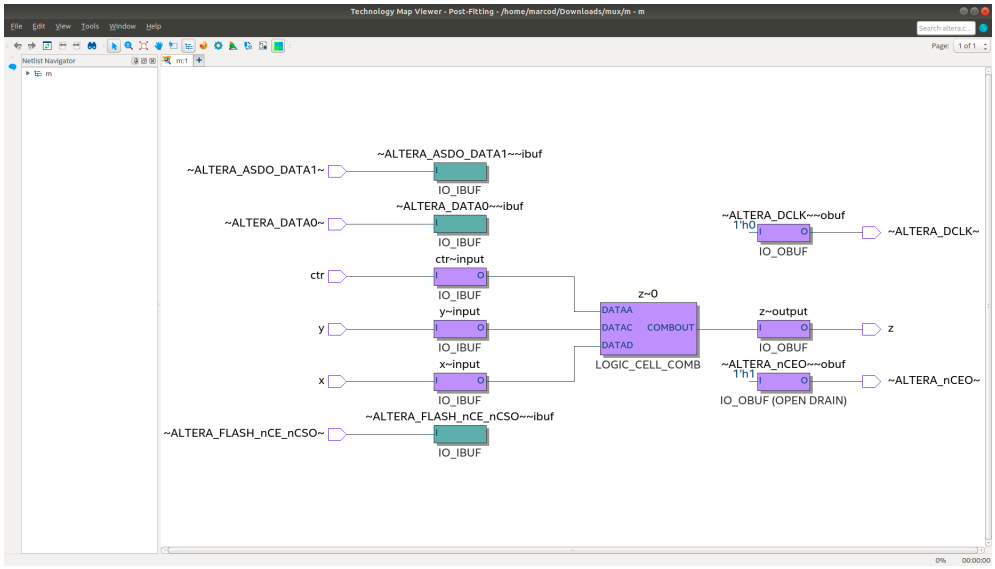
Selezionando dal menu "Processing" l'entry "Start compilation", dopo ca un minuto su un laptop con i5 dual core otteniamo quanto mostrato nella prossima snapshot:



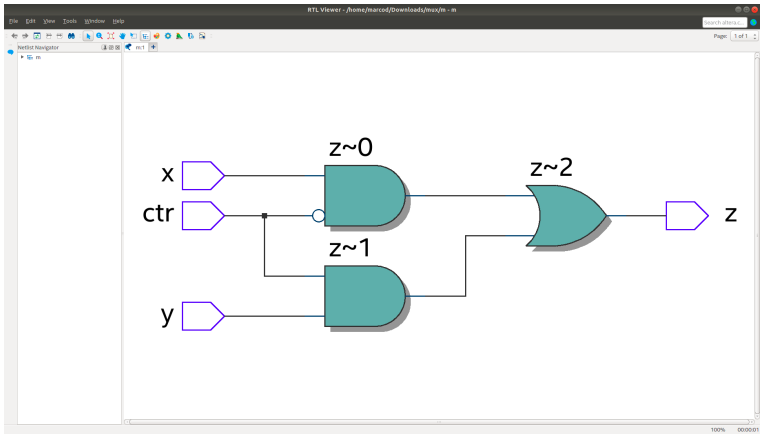
Adesso possiamo vedere cosa è stato generato come prodotto della sintesi, andando a scegliere dal menu "Tools" entry "Netlist" subentry "RTL":



I tool di sintesi hanno riconosciuto il multiplexer e lo hanno implementato come tale (1 cella che esegue il routing a seconda dell'ingresso di controllo). Se chiediamo invece che la vista RTL quella delle celle otteniamo però la seguente cosa:



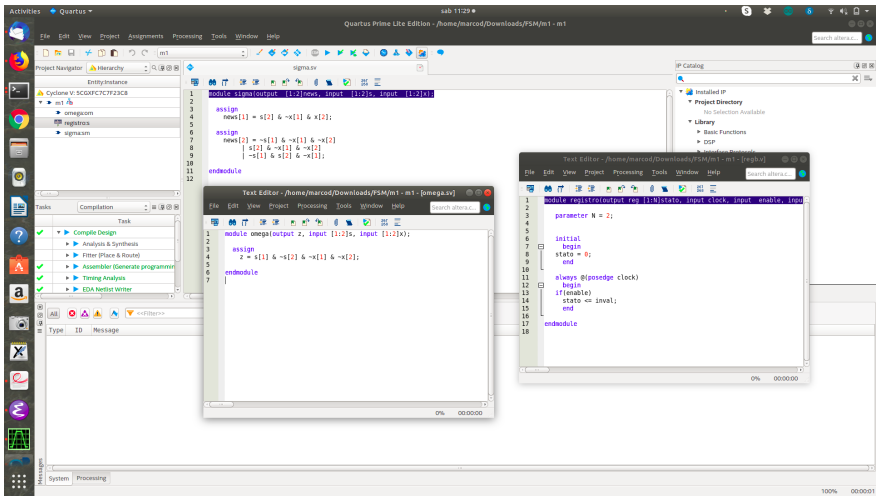
dove si vede chiaramente come il multiplexer sia stato ottenuto programmando una cella come una cella che implementa una rete logica da tre ingressi e un'uscita (la `LOGIC_COMB_CELL` al centro) al netto dei vari buffer utilizzati per pilotare i segnali. Se però il multiplexer lo descriviamo come espressione booleana (sostituiamo la assign del codice precedente con una che esegue `assign z = !ctr&x | ctr&y;`), otteniamo la netlist RTL:



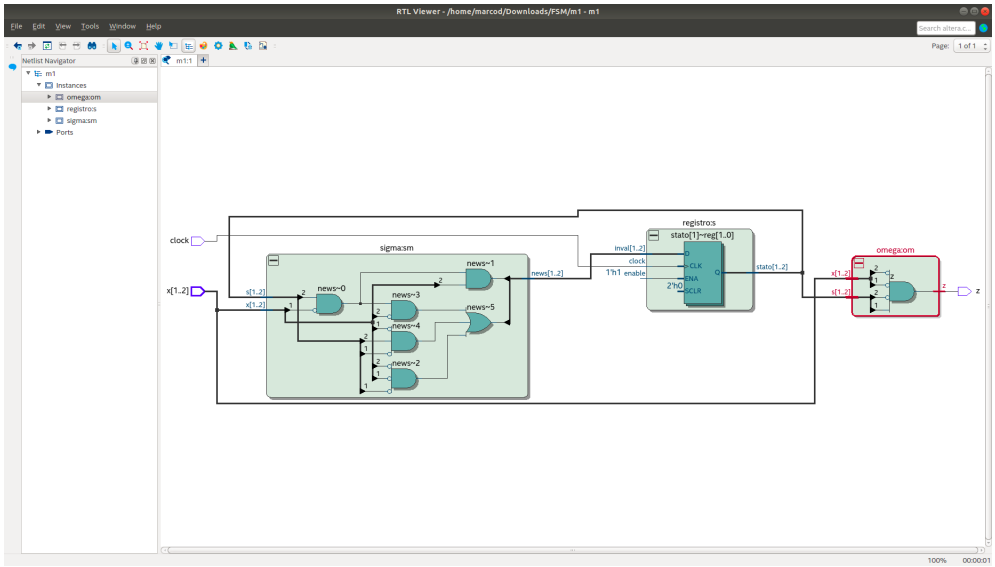
che corrisponde allo stesso tipo di utilizzo delle celle FPGA della figura con la `LOGIC_COMB_CELL` vista poco fa.

5.3.1 Sintesi di una rete sequenziale (modello strutturale)

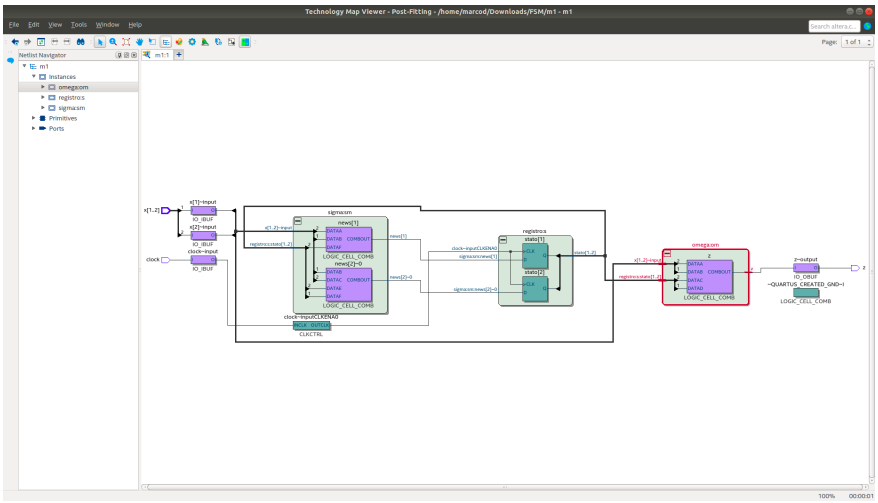
Se consideriamo qualcosa di più complesso, come per esempio la rete sequenziale che riconosce le stringhe "abba" sull'alfabeto `{a,b,c}` il processo di sintesi fornisce qualcosa di più significativo. In questo caso nel progetto includiamo i file per sigma, omega, e registro:



avviando la compilazione otteniamo:



cui corrisponde la configurazione di celle della FPGA:



Se invece di utilizzare il codice relativo al modello strutturale della rete sequenziale utilizzassimo il codice behavioural:

Text Editor - /home/marcod/Downloads/FSM/m1 - m1 - [m1.sv]

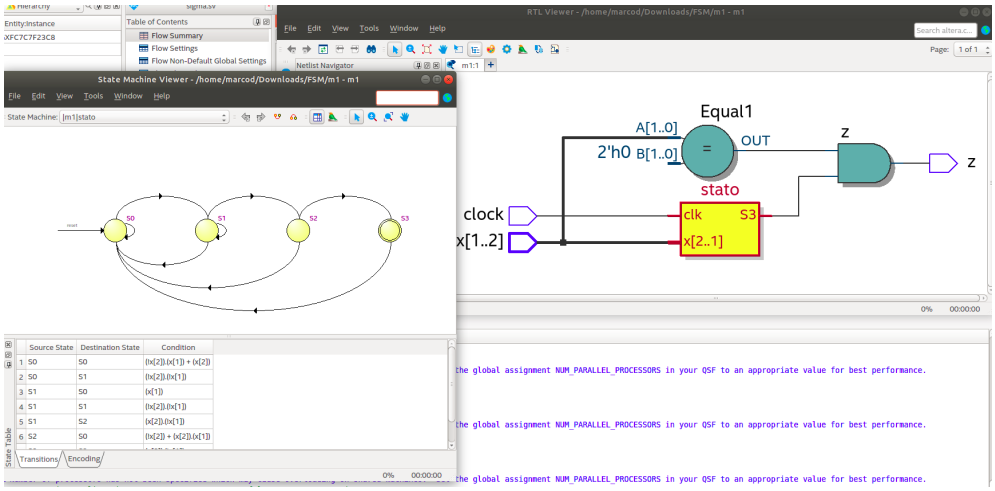
File Edit View Project Processing Tools Window Help

Search altera.c...

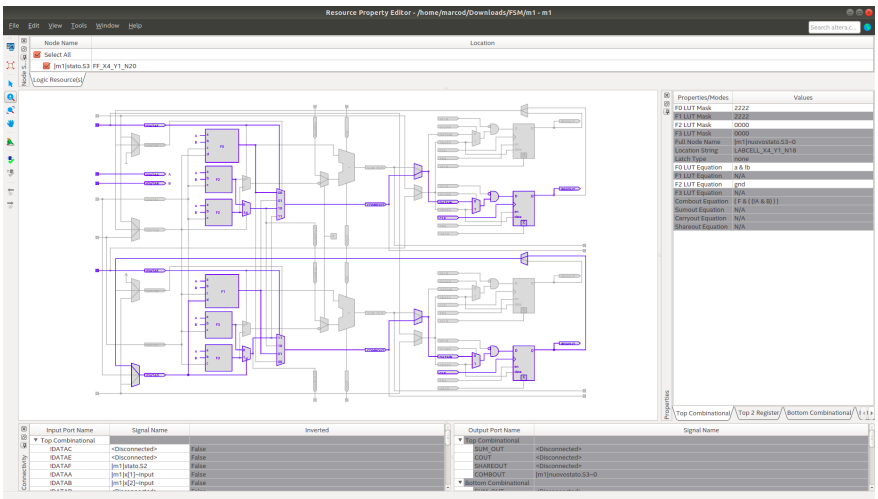
```
1  module m1(output z, input [1:2]x, input clock);
2
3     reg [1:2] stato;
4     reg [1:2] nuovostato;
5
6     parameter S0=2'b00;
7     parameter S1=2'b01;
8     parameter S2=2'b11;
9     parameter S3=2'b10;
10    parameter A=2'b00;
11    parameter B=2'b01;
12    parameter C=2'b11;
13
14    initial
15      stato = S0;
16
17    always @(posedge clock)
18       stato <= nuovostato;
19
20    always @(*)
21      begin
22    case(stato)
23      S0: nuovostato = (x == A ? S1 : S0);
24      S1: nuovostato = (x == B ? S2 : (x == A ? S1 : S0));
25      S2: nuovostato = (x == B ? S3 : (x == B ? S1 : S0));
26      S3: nuovostato = S0;
27    endcase // case (stato)
28      end
29
30    assign
31      z = ((stato == S3 && x == A) ? 1 : 0);
32
33 endmodule
34
```

100% 00:00:01

allora il risultato della sintesi sarebbe nettamente diverso:



Vedete che la sintesi ha prodotto un minimo di rete combinatoria per il calcolo dell'uscita e una macchina a stati finiti (la parte gialla, di cui la vista ad automa (finestra a sinistra) si ottiene cliccandoci sopra due volte. Questo corrisponde (vista chip planner) alla mappatura della figura che segue sulle celle della FPGA. Facciamo solo notare che ci sono delle ALU grigie (la parte grigia è quella che non viene utilizzata) che rappresentano la colonna di macro celle di cui abbiamo parlato nella sezione 5.1.



Capitolo 6

Installazione tool e manuali online

I tool che utilizziamo sono tutti Open Source e possono essere utilizzati sia sotto Linux che sotto Windows e Mac OS X.

Gli esempi e i dump video di queste note sono tutti stati testati utilizzando `Icarus iverilog` e `GTKWave`. Per Linux sono disponibili pacchetti che si possono istallare con un comando

```
apt install iverilog gtkwave
```

eseguito con i diritti di superutente. Windows e MAC OS/X necessitano di procedure di installazione leggermente diverse, comunque documentate sul sito dei tool:

- `http://iverilog.icarus.com/` per `iverilog`
- `http://gtkwave.sourceforge.net/` per `gtkwave`

6.1 Installazione Linux (UBUNTU)

Per installare iverilog:

- sudo apt install iverilog

Per installare gtkwave:

- sudo apt install gtkwave

6.2 Utilizzazione Linux

6.2.1 Compilazione

Un modulo Verilog `mod.v` e il relativo programma di test `test.v` possono essere compilati con il comando:

```
iverilog mod.v test.v -o eseguibile
```

In generale, dopo il nome del compilatore si debbono mettere i nomi di tutti i moduli necessari ad implementare l'unità, incluso il programma di test. Qualora mancasse un modulo, si ottiene un messaggio di errore tipo:

```
marcod@marcod-ThinkPad-E480:~/Didattica/Architettura/NuovoCorso/Esercizi/FSM/ABBA$ iverilog
    test-m1.v  m1.v regb.v sigma.v
m1.v:8: error: Unknown module type: omega
2 error(s) during elaboration.
*** These modules were missing:
        omega referenced 1 times.
***
marcod@marcod-ThinkPad-E480:~/Didattica/Architettura/NuovoCorso/Esercizi/FSM/ABBA$
```

E' evidente che ci siamo dimenticati del modulo che implementa omega.

6.2.2 Esecuzione della simulazione

Qualora si siano indicati tutti i file necessari, viene creato un eseguibile che, nel caso non ne sia stato specificato il nome utilizzando un'opzione `-o <nomefileeseguibile>`, sarà il file `a.out`. Il file eseguibile in realtà contiene codice da eseguire mediante l'interprete `vvp`. Le prime righe del file saranno qualcosa tipo:

```
marcod@marcod-ThinkPad-E480:~/Didattica/Architettura/NuovoCorso/Esercizi/FSM/ABBA$ head a.
    out
#! /usr/bin/vvp
:ivl_version "10.1 (stable)";
:ivl_delay_selection "TYPICAL";
:vpi_time_precision + 0;
:vpi_module "system";
:vpi_module "vhdl_sys";
:vpi_module "v2005_math";
:vpi_module "va_math";
S_0x5625aa43da70 .scope module, "testm1" "testm1" 2 1;
 .timescale 0 0;
marcod@marcod-ThinkPad-E480:~/Didattica/Architettura/NuovoCorso/Esercizi/FSM/ABBA$
```

Come si vede, la prima riga dice che il programma va eseguito utilizzando `/usr/bin/vvp`. Dunque il file può essere eseguito semplicemente dando il comando:

```
./a.out
```

oppure richiamando direttamente l'interprete con il comando:

```
vvp a.out
```

Se il programma di test genera le tracce di esecuzione con un `$dumpfile("file.vcd")`, i risultati possono essere analizzati con il comando

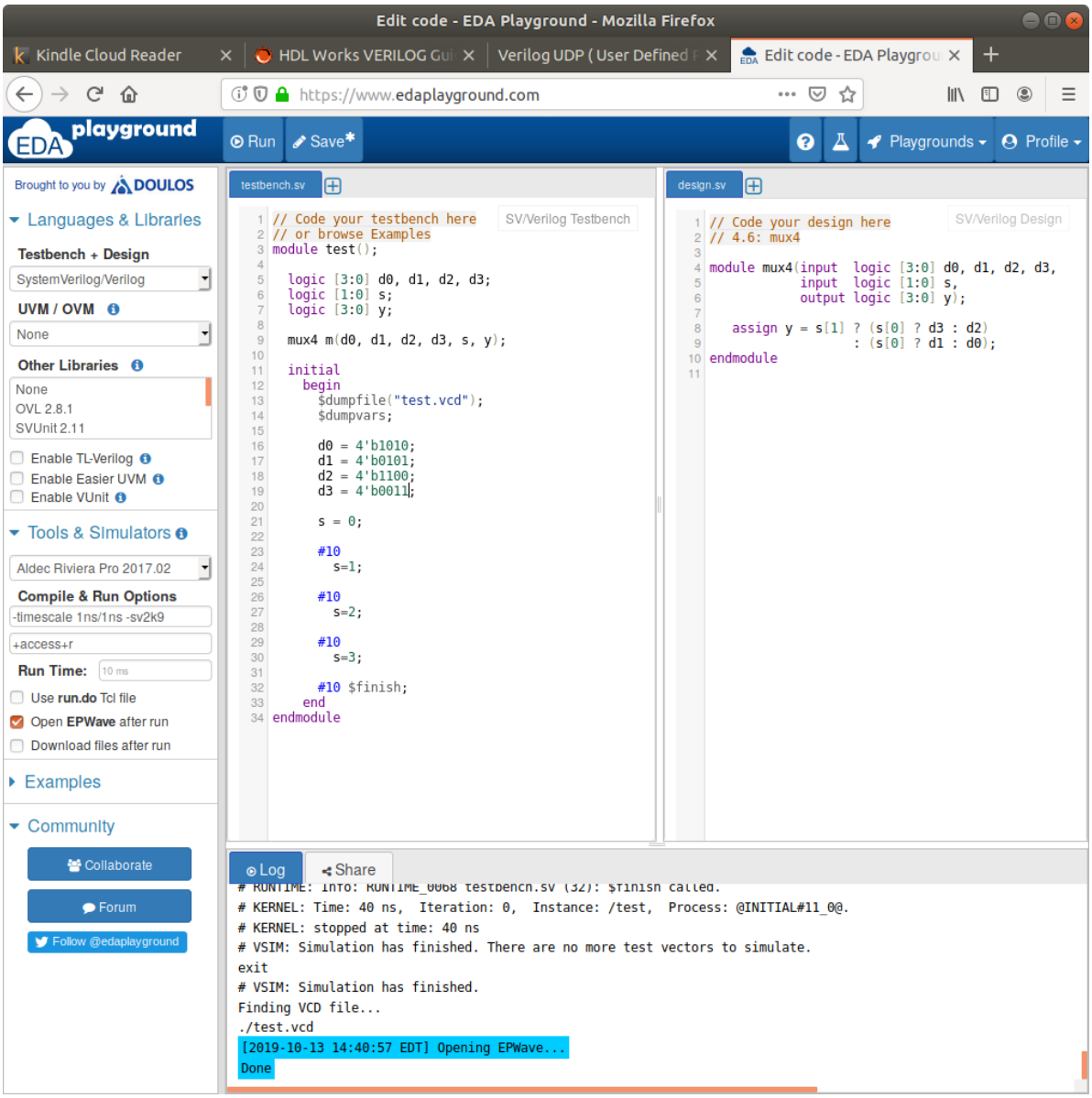
- `gtkwave file.vcd`

6.3 Strumenti online

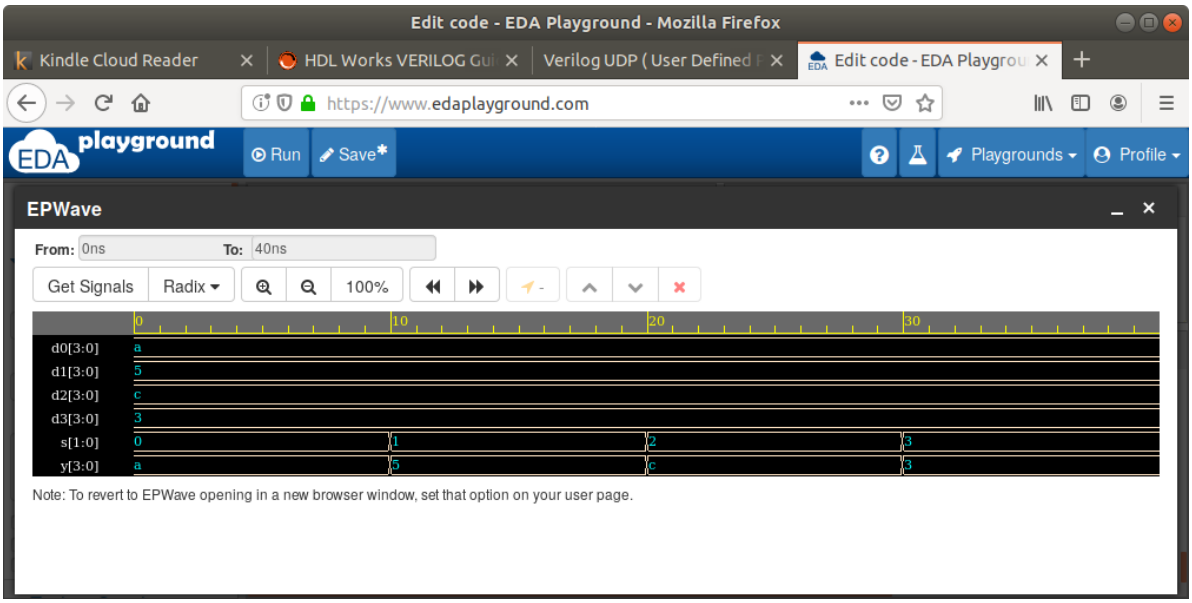
Infine, per sperimentare semplici esercizi Verilog o System Verilog, si può utilizzare il sito online

```
https://www.edaplayground.com/
```

Per esempio, potremmo simulare il comportamento del codice di `mux4` come descritto nel libro di testo (System Verilog) utilizzando il codice in figura:



Se selezioniamo “Open EPwave after run” sulla sinistra e, per esempio, il compilatore simulatore della Aldec Riviera, otteniamo una cosa tipo:



che ci permette di verificare il comportamento del modulo sotto test.

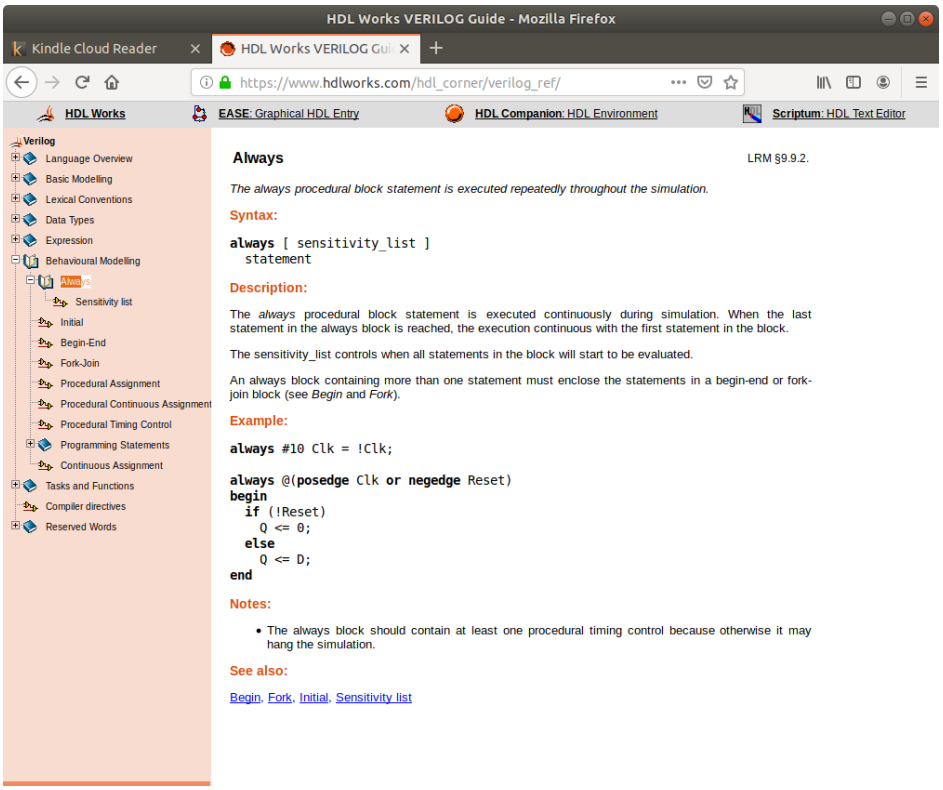
6.4 Materiale di consultazione

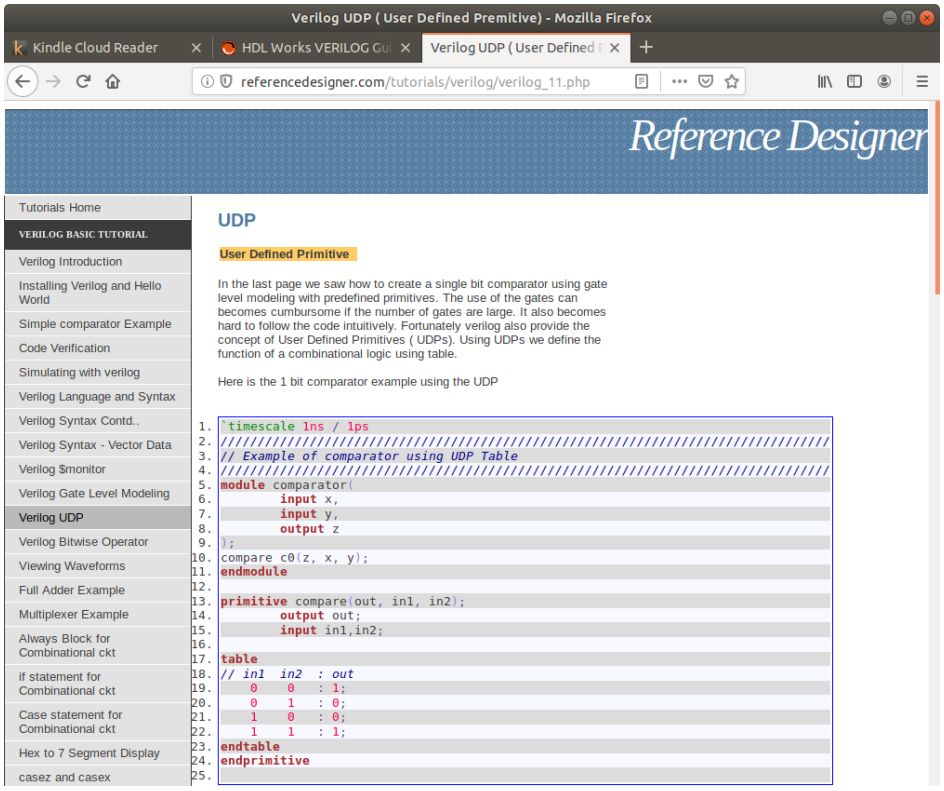
Un buon sito che descrive correttamente e in maniera concisa la sintassi del Verilog è quello che si trova all'indirizzo:

`https://www.hdlworks.com/hdl_corner/verilog_ref/`

Potrebbe essere utile come materiale di consultazione anche il sito web al link:

`http://referencedesigner.com/tutorials/verilog/verilog_01.php`




Verilog UDP (User Defined Premitive) - Mozilla Firefox
Kindle Cloud Reader
HDL Works VERILOG Gui
Verilog UDP (User Defined F
referencedesigner.com/tutorials/verilog/verilog_11.php
Reference Designer
Tutorials Home
VERILOG BASIC TUTORIAL
Verilog Introduction
Installing Verilog and Hello World
Simple comparator Example
Code Verification
Simulating with verilog
Verilog Language and Syntax
Verilog Syntax Contd..
Verilog Syntax - Vector Data
Verilog $monitor
Verilog Gate Level Modeling
Verilog UDP
Verilog Bitwise Operator
Viewing Waveforms
Full Adder Example
Multiplexer Example
Always Block for Combinational ckt
if statement for Combinational ckt
Case statement for Combinational ckt
Hex to 7 Segment Display
casez and casex
UDP
User Defined Primitive
In the last page we saw how to create a single bit comparator using gate level modeling with predefined primitives. The use of the gates can becomes cumbursome if the number of gates are large. It also becomes hard to follow the code intuitively. Fortunately verilog also provide the concept of User Defined Primitives (UDPs). Using UDPs we define the function of a combinational logic using table.
Here is the 1 bit comparator example using the UDP
1. `timescale 1ns / 1ps
2. //
3. // Example of comparator using UDP Table
4. //
5. module comparator(
6. input x,
7. input y,
8. output z
9.);
10. compare c0(z, x, y);
11. endmodule
12.
13. primitive compare(out, in1, in2);
14. output out;
15. input in1,in2;
16.
17. table
18. // in1 in2 : out
19. 0 0 : 1;
20. 0 1 : 0;
21. 1 0 : 0;
22. 1 1 : 1;
23. endtable
24. endprimitive
25.

Bibliografia

[1] D. M. Harris & S. L. Harris, Digital Design and Computer Architecture, Morgan Kaufmann, 2007, Capitolo 4 "Hardware Description Languages".

[2] D. M. Harris & S. L. Harris, Digital Design and Computer Architecture: ARM edition, Morgan Kaufmann, 2017.

[3] Peter M. Nyasulu, Introduction to Verilog, 2001, http://www.csd.uoc.gr/~hy220/2008f/lectures/verilog-notes/VerilogIntroduction.pdf

[4] Doulos, The Verilog Golden Reference Guide, 1996, www.fpga.com.cn/hdl/training/verilog\%20reference\%20guide.pdf

[5] Deepak Kumar Tala, Verilog Tutorial, 2003, www.ece.ucsb.edu/courses/ECE152/.../VerilogTutorial.pdf

[6] E. Madhavan, Quick reference for Verilog HDL, 1995, www.stanford.edu/class/ee183/handouts.../VerilogQuickRef.pdf

[7] S. A. Edardw, *The Verilog language*, slides Columbia University, 2002, http://www.cs.columbia.edu/~sedwards/classes/2002/w4995-02/verilog.9up.pdf

Parte II

Assembler

Capitolo 7

Tool

7.1 Workflow

Per compilare ed eseguire un programma scritto in Assembler ARM con gli strumenti GNU si devono sostanzialmente eseguire una serie di passi, secondo un workflow come quello descritto in Figura 7.1. Nel seguito useremo l'acronimo RbArm per caratterizzare le parti e gli strumenti che fanno riferimento all'utilizzo di tool GNU su architettura con processore ARM (e.g. Raspberry, Odroid, etc) e l'acronimo CcArm per caratterizzare le parti e gli strumenti che fanno riferimento all'utilizzo della tolchain cross compiler di GNU sotto LINUX. Come spiegato a lezione, gli strumenti CcArm si possono utilizzare sia sotto Linux che sotto Windows in maniera relativamente semplice:

- sotto Linux, è sufficiente installare tre pacchetti:
  `gcc-arm-linux-gnueabihf`, `qemu` e `gdb-multiarch`.
  Per installare i pacchetti, da promt di sheel root va digitato il comando
  `apt install gcc-arc-linux.gnueabinhf qemu gdb-multiarch`[1].

- sotto Windows, è necessario attivare l'ambiente WSL2 (istruzioni al link `https://docs.microsoft.com/it-it/windows/wsl/install-win10`)
  e successivamente l'app "Linux Ubuntu 20.04" dall'app store Microsoft. A quel punto, lanciando l'applicazione "ubuntu" si ottiene un terminale con una shell `bash` nel quale si possono installare i pacchetti `gcc-arm-linux-gnueabihf`, `qemu` e `gdb-multiarch` utilizzando la stessa procedura seguita in Linux.

Per quanto riguarda Mac OS/X, la cosa più semplice è quella di installare una macchina virtuale Ubuntu utilizzando Virtual Box. E' sufficiente installare la versione base e aggiungere i pacchetti mezionati qui sopra.

7.1.1 Scrittura del programma

Il programma viene scritto, normalmente in un file `.s` (detto *file sorgente*), utilizzando un normale editore di testo in grado di salvare in puro formato ASCII (text only). Nello scrivere il programma occorre rispettare la sintassi GNU che differisce dalla sintassi ARM ufficiale per poche ma importanti cose:

- le etichette sono seguite dal carattere ":" (in sintassi ARM non vanno messi i due punti);

- le direttive (vedi Sez. 7.2) sono tutte sostanzialmente diverse come sintassi.

7.1.2 Compilazione

Per produrre un binario occorre compilare il file sorgente. Questo può essere fatto

- o utilizzano `gcc` direttamente, con l'opzione `-c`, oppure chiamando direttamente l'assemblatore(comando `as`) ; RbArm

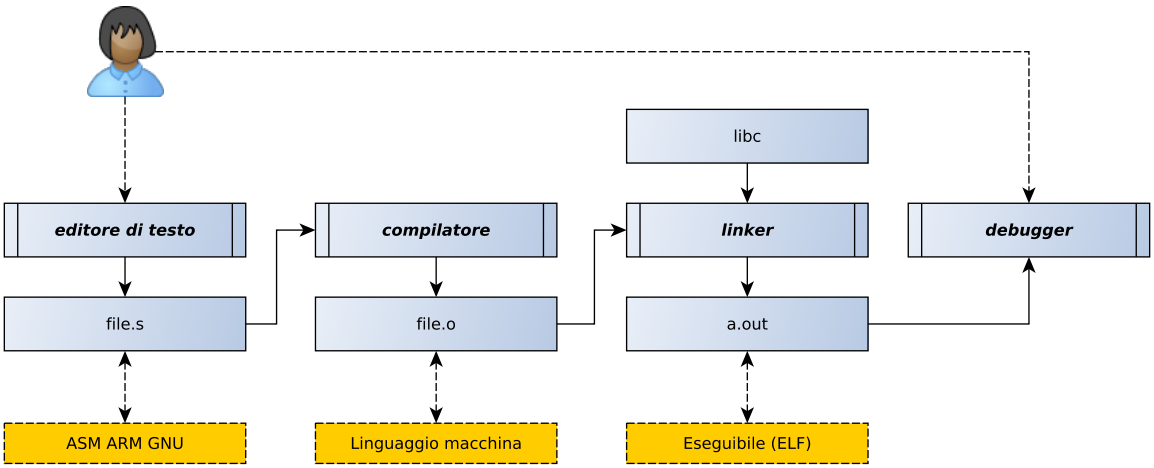


Figura 7.1: Workflow per la realizzazione di un programma in Assembler ARM con i tool GNU

- In caso di utilizzo dei tool cross-compiler i due comandi si chiamano rispettivamente: `arm-linux-gnueabihf-gcc` e `arm-linux-gnueabihf-as`. CcArm

L'output nei due casi è un file `.o` (con `gcc -c`) o `a.out` (con `as`) che è detto *file oggetto*. Per essere eseguito deve essere processato mediante il linker che provvede a collegare le librerie necessarie e ad aggiungere quanto serve a far eseguire il `main`.

7.1.3 Linking

Per collegare le librerie necessarie e rendere eseguibile il file oggetto è necessario invocare il linker passandogli come parametri tutti i file oggetto da includere nell'eseguibile. Il linker GNU è invocato utilizzando di nuovo `gcc` tipicamento con il parametro `-o nomeeseguibile`. Questo avviene invocando:

- `gcc file.o [file2.o ...] -o nomeeseguibile`;
- `arm-linux-gnueabihf-gcc file.o [file2.o ...] -static -o nomeeseguibile`
  L'opzione `-static` serve per includere le funzioni di libreria utilizzare direttamente nel codice. Dal momento che le librerie sorgenti ARM non si possono trovare nel posto dove stanno di solito le librerie (lì ci saranno quelle compilate in assembler x86), al momento dell'esecuzione `qemu-arm` non sarebbe in grado di risolvere i riferimenti dinamici.

L'output del linker è un file `nomeeseguibile` o, in assenza del `-o` che ne specifica il nome, il file `a.out`. Questo è detto *file eseguibile* o semplicemente *eseguibile*.

Compilazione e linking (non separati)

Naturalmente è possibile compilare e linkare il proprio codice utilizzando un unico comando, che è sempre `gcc` (RbArm) o `arm-linux-gnueabihf-gcc` (CcArm). Di fatto il comando `gcc` è una specie di vera e propria *shell di compilazione* che a seconda dei parametri e dei tipi di file passati come argomento effettua diverse funzioni che vanno dalla semplice compilazione alla generazione di un eseguibile completo (azione di default). Per compilare un programma assembler nel file `myProg.s` e generare un eseguibile `myProg.exe` possiamo utilizzare i comandi:

- `gcc -o myProg.exe myProg.s`, oppure
- `arm-linux-gnueabihf-gcc -o myProg.exe myProg.s`

[1] assumo che abbiate una distribuzione Linux Ubuntu

Qualora il codice in `myProg.s` chiamasse funzioni definite in altri file (e.g. `fun.s` e `util.s`) si può elencare tutti i file sorgente nella lista di parametri del comando `gcc`[2]:

- `gcc -o myProg.exe myProg.s fun.s util.s`, oppure RbArm
- `arm-linux-gnueabihf-gcc -o myProg.exe myProg.s fun.s util.s` CcArm

Notare che questo ha lo stesso effetto dell'utilizzo della compilazione separata e del linking eseguito successivamente, come avviene nella sequenza di comandi (RbArm, per CcArm vanno cambiati come al solito i nomi dei comandi `gcc` in `arm-linux-gnueabihf-gcc` e nel comando che effettua il linking (l'ultimo della serie qui sotto) va aggiunta l'opzione `-static` per evitare problemi con le librerie dinamiche quando eseguiremo il programma con una `qemu-arm myProg.exe`):

```
gcc -c myProg.s
gcc -c fun.s
gcc -c util.s
gcc -o myProg.s myProg.o fun.o util.o
```

7.1.4 Esecuzione

Per eseguire il programma eseguibile, da riga di comando se ne può digitare il nome. In caso di utilizzo degli strumenti cross-compiler, occorre utilizzare il comando `qemu-arm` seguito dal nome dell'eseguibile e dagli eventuali parametri.

- se avete compilato l'eseguibile su una macchina dotata di processore arm e tool GNU con un comando `gcc -o myProg ...` potete eseguire il programma con `./myProg` seguito dagli eventuali parametri di riga di comando; RbArm
- se avete usato il crosscompiler (arm-linux-gnueabihf-gcc -o myProg ...—) potete eseguire il programma con `qemu-arm myProg` seguito dagli eventuali parametri di riga di comando. Qualora si verificasse un problema di caricamento delle librerie, ricompilate l'eseguibile utilizzando anche il flag `-static`:
  `gcc -o myProg -static ...` CcArm

7.1.5 Debugging

Il debugger GNU si chiama gbd. La documentazione completa del debugger la trovate su `https://sourceware.org/gdb/current/onlinedocs/gdb/`.

Nel caso utilizziate una macchina con processore ARM,per eseguire il debugger basta: RbArm

- compilare con l'opzione `-g` (necessaria per mantenere nell'eseguibile la tabella dei simboli (etichette) del programma)
- eseguire il debugger con il comando `gdb myProg`.

Se invece state utilizzando il cross compilerla procedura è leggermente più complessa. CcArm

- compilate il programma con l'opzione `-ggdb3`
- lanciate il programma in modalità debug con il comando
  `qemu-arm -g 12345 myProg &`.
  La `&` a fine riga serve per mandare il programma in background, senza dover aprire un altro terminale
- aprite il debugger `gdb-multiarch` con i parametri
  `gdb-multiarch -q --nh -ex 'set architecture arm'`
  `-ex 'file myProg' -ex ' target remote localhost:12345'`
  Il numero di porta per il debugger può essere un numero qualunque di una porta effimera (da 1K a 32K-1) purchè libera. Deve essere utilizzato identico nel comando `qemu-arm -g` e nel comando che lancia il debugger dopo `localhost:`. Analogamente, nell'opzione `-ex 'file` va indicato il nome del programma eseguibile che è stato lanciato mediante il comando `qemu-arm -g`

[2] come avviene per la compilazione di programmi distribuiti in più file sorgenti in C

7.2 Direttive

Il programma assemblatore accetta istruzioni dell'assembler ARM e *direttive* ovvero istruzioni che non sono vere e proprie istruzioni assembler ma che servono per definire le varie sezioni del codice e/o eventuali aree di memoria riservate per i dati del programma[3].

Le direttive di interesse sono queste:

- `.data`
  definisce una sezioni di dati, ovvero con sole direttive che riservano aree di memoria. All'interno della sezione `.data` tipicamente troviamo direttive:
  - `.word`
    seguita da valori separati da virgole, che riserva un'area di memoria di tante parole quanti sono i valori che seguono, inizializzata con i valori che seguono la `.word`;
  - `.byte`
    seguita da valori separati da virgole, che riserva un'area di memoria di tanti byte quanti sono i valori che seguono, inizializzata con i valori che seguono la `.byte`;
  - `.fill`
    seguita da un numero intero (che deve essere un multiplo di 4), che riserva un'area di memoria di tanti byte quanto vale il parametro intero;
  - `.string`
    seguita da una stringa fra virgolette, che riserva un'area di memoria sufficiente a contenere i caratteri della stringa seguiti da un carattere con codice `0`.

  Tutte queste direttive sono normalmente precedute da un'etichetta, che può essere utilizzata per reperirne l'indirizzo base.

- `.text`
  definisce un'area di codice. All'interno della sezione troviamo normalmente una o più direttive
  - `.global <etichetta>`
    che denotano i simboli che devono essere resi noti al debugger[4]

  nonchè istruzioni assembler che costituiscono il codice vero e proprio.

Esempio di direttive

```
1     .data           @ definisce una sezione dati
2
3  a: .word 1,2,3,4   @ definisce un vettore di 4 parole
4                     @ che contengono i valori da 1 a 4
5                     @ e il cui indirizzo base e' a
6
7  b: .fill 32        @ riserva un area di memoria da 8 parole
8                     @ (32 = 8 * 4 byte) il cui indirizzo base e' b
9
10 c: .string "ciao"  @ riserva 5 byte per contenere i caratteri
11                    @ della stringa in codice ASCII e il
12                    @ carattere NULL (codice 0) di fine stringa
13
14    .text           @ definisce una sezione che contiene codice
15    .global main    @ il simbolo main viene esportato nella
16                    @ tabella dei simboli da utilizzare fuori
17    .global mdfun   @ simbolo mdfun esportato
18
19 main: mov r0, #0
```

[3] In realtà ci sono anche direttive che servono per la compilazione condizionale, per le macro, etc. ma noi non le abbiamo mai utilizzate nel corso

[4] più in generale resi pubblici in fase di linking

```
20      ldr r1, =d
21      ldr r1,[r1]
22      b cont
23 d:   .word 12345   @ esempio di costante in mezzo al codice
24                    @ va bene, basta evitare di "passarci"
25                    @ in esecuzione (va messa ad un indirizzo
26                    @ non "raggiunto" dal PC ...) come qui
27 cont: add r1, r1, #1
```

7.3 Chiamate di libreria

Il linker invocato mediante `gcc` permette di utilizzare alcune funzioni di libreria utili per gestire diverse funzionalità che in assembler puro sarebbero molto difficili da programmare. Tipicamente, `printf`, `gets`[5], `scanf`, `atoi`, etc.

Per le chiamate alla `libc` valgono le solite convenzioni dell'assembler ARM:

- i parametri in ingresso sono passati mediante R0, R1, R2, R3 e, se in numero maggiore a 4, mediante lo stack;
- il parametro di ritorno è restituito in R0;
- i registri R0-R3 non sono preservati, quelli da R4 in sù sì.

Dunque per chiamare una printf dovremmo utilizzare:

- R0 per contenere l'indirizzo della stringa di formato
- R1 per contenere il primo parametro (quello che va nel primo campo %), se presente
- R2 per contenere il secondo parametro (quello che va nel secondo campo %), se presente
- R3 per contenere il terzo parametro (quello che va nel terzo campo %), se presente
- altri parametri (se presenti) sullo stack.

Per esempio, il codice che segue chiama una `printf`:

```
1      .data
2      ...
3 f:   .string "Primo intero : %d secondo : %d \n"
4      ...
5      .text
6      ...
7      ldr R0, =f
8      mov R1, ...
9      mov R2, ...
10     bl printf
11     mov ..., R0
```

Al termine della `bl` il registro R0 contiene il numero di caratteri effettivamente stampati.

7.3.1 Funzioni `glibc`

Le funzioni messe a diposizione dalla `glibc` sono quelle che trovate alla pagina `https://www.gnu.org/software/libc/manual/html_node/Function-Index.html`. Nella tabella che segue ve ne riporto alcune che potrebbero essere utili da richiamare dal codice assembler.

[5] da utilizzare con tutte le cautele del caso e solo per i piccoli programmi realizzati come esercizi per il corso

Nome	Funzione
atoi, atof	conversione di stringhe in interi e float
malloc, calloc	allocazione di memoria sullo heap
rand, srand	generazione numeri pseudocasuali
getc	legge caratteri dallo standard input
isalnum, isalpha, isblank	testa tipo caratteri
memcpy	copia aree di memoria
memset	inizializza aree di memoria
printf, scanf	input/output
strcmp, strcpy, strlen	manipolazione stringhe

7.3.2 Malloc

Due parole in particolare per allocare memoria senza ricorrere a direttive tipo la `.fill`, ovvero dinamicamente. Possiamo chiamare la `malloc` della `libc` per questo scopo. La malloc prende un solo parametro (dimensione in bytes della memoria da allocare). Per allocare un certo numero di parole conviene dunque caricare il numero delle parole da allocare in un registro e successivamente moltiplicare il registro per 4 utilizzando una `LSL Rx, Rx, #2`. Il valore di ritorno della `malloc` sarà l'indirizzo base dell'area di memoria allocata. In caso di errore il valore restituito sarà `NULL` (ovvero 0). Il risultato della `malloc` andrà testato *sempre* prima di utilizzare l'area di memoria ottenuta, pena un errore di protezione.

Il codice che segue fa vedere come allocare un segmento di memoria di 8 parole in cui andiamo a mettere tutti valori 7.

```
1   .data
2 err:  .string "Errore nella malloc\n"
3 succ: .string "Malloc ha avuto successo (ind = %d)\n"
4
5   .text
6   .global main
7
8 main: mov r0, #8          @ 8 parole
9   lsl r0, r0, #2      @ *4 perche' sono byte
10  bl malloc           @ chiama la malloc
11  cmp r0, #0          @ controlla se ret = NULL
12  beq error           @ in caso segnala errore
13  mov r1, r0          @ salva indirizzo mem allocata
14  mov r0, #0          @ i = 0
15  mov r2, #7          @ valore da utilizzare per scrivere
16 loop:   str r2, [r1,r0]      @ x[i] = 7
17  add r0, r0, #4      @ i++ (in byte i+=4)
18  cmp r0, #32         @ siamo alla fine
19  beq fine            @ se si esci
20  b loop              @ altrimenti for i+1
21 error:  ldr r0, =err         @ in caso di errore
22  bl printf           @ stampa messaggio ed esci
23  mov r7, #1          @ exit
24  svc 0
25 fine:   ldr r0, =succ        @ stampa messaggio di successo
26  push {r1}           @ ind serve per la free, va salvato
27  bl printf           @ puo' sporcare r0-r3
28  pop {r1}            @ ripristino ind -> r1 dallo stack
29  mov r0, r1          @ chiama la free: ptr -> r0
30  bl free             @ free(memoria allocata)
31  mov r7, #1          @ exit
32  svc 0
```

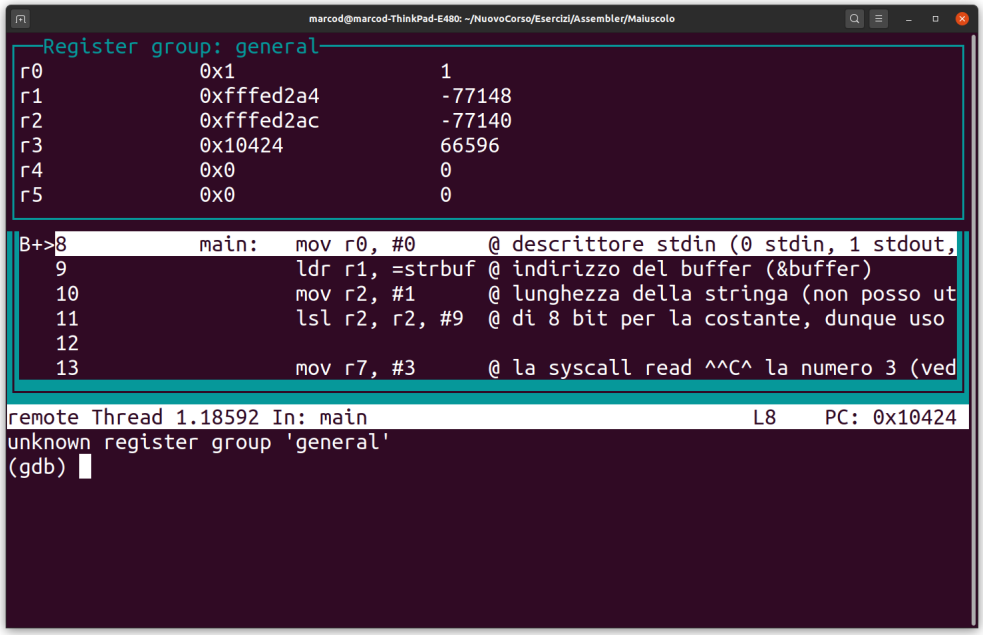
Dopo l'etichetta `fine` si stampa un messaggio di successo che indica quale indirizzo è stato restituito. In seguito chiamiamo una `free` per liberare la memoria allocata. Poichè l'indirizzo da liberare era in `r1` prima della call alla `printf` va salvato o in un registro alto o sullo stack, dal momento che i registri da `r0` a `r3` non sono mantenuti nelle chiamate (vedi Sec. 7.3). In questo caso salviamo il contenuto di `r1` sullo stack prima di passarlo alla `printf` (`push` alla riga 26) e poi lo recuperiamo dallo stack dopo il ritorno dalla `printf` (`pop` alla riga 28).

7.4 Single step execution

Riassumiamo i principali comandi che permettono di eseguire un programma step by step (una istruzione alla volta) utilizzando il debugger.

Una volta lanciato il debugger, potete cominciare mettendo un `breakpoint` al punto in cui volete cominciare a osservare l'esecuzione del programma step by step. Tipicamente, se volete osservare l'effetto delle istruzioni a partire dal `main`, al prompt date un comando `break main` (o, abbreviato, `b main`). Facendo partire il programma con il comando `run` (o, abbreviato, `r`) l'esecuzione si ferma immediatamente prima dell'esecuzione della istruzione etichettata con `main`. **Attenzione**: se si utilizzano gli strumenti crosscompiler, quando si fa partire il debugger multiarch l'esecuzione è già partita e dunque va dato un comando `continue` invece che `run` per procedere nell'esecuzione del programma, dopo l'eventuale `break main`. A questo punto: CcArm

- `tui reg general`
  fa vedere nel terminale nella parte alta il valore dei reigistri general, nella parte bassa il prompt di gdb e nella parte intermedia il codice assembler con l'istruzione che sta per essere eseguita in evidenza:



  Per vedere i registri in virgola mobile, si può indicare `float` al posto di `general`. `all` fa vedere invece tutti i registri (generali, floating point, di controllo, ...);

- `next`
  (abbreviato `n`) permette di eseguire la prossima istruzione. Nel caso sia una chiamata di funzione/procedura, la funzione/procedura viene eseguita per intero e il debugger si ferma alla prossima istruzione (quella che segue la `bl`);

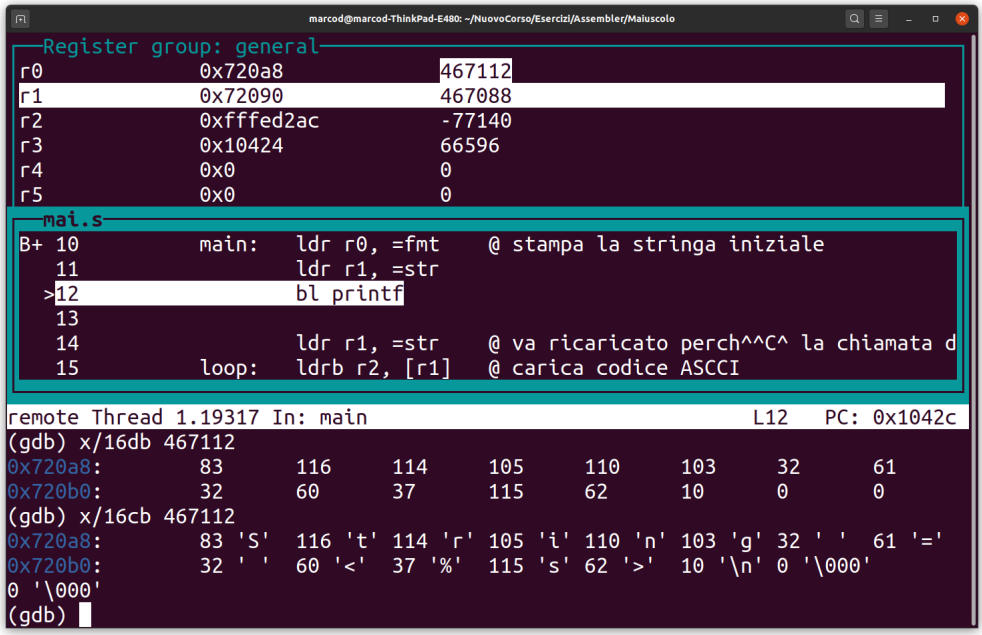
- `step`
  (abbreviato `s`) permette di eseguire la prossima istruzione. Qualora l'istruzione sia una `bl` il debugger entra nella funzione/procedura chiamata e si ferma alla prima istruzione della funzione;

- `continue`
  (abbreviata `c`) permette di continuare senza single step fino al prossimo breakpoint.
  Quest'opzione deve essere utilizzaquando si lavori con `gdb-multiarch` *invece* dell'opzione `run` per far partire l'esecuzione del programma! CcArm

- `x`
  serve per vedere il contenuto di un'area di memoria di indirizzo noto. Il comando prevede un caarattere `/` dopo la `x` seguito da

- un numero, che indica quanti elementi far vedere
- un formato, carattere che indica che formato utilizzare per i vari elementi (`d` per decimale, `x` per esadecimale, `b` per binario, ...)
- un'ampiezza, caratter che indica se vogliamo vedere parole (`w`), byte (`b`), ...
- un indirizzo, che è l'indirizzo base da cui cominciamo a vedere i valori

Per esempio `x/16dw 16384` ci farà vedere 16 parole (`w`) in formato decimale (`d`) a partire dall'indirizzo (decimale) 16384. Avremmo potuto utilizzare un indirizzo esadecimale facendolo precedere dal prefisso `0x` come al solito.



Nella figura si vede come possiamo osservare una stringa in memoria come codici decimali e come caratteri.

- `info registers`
  permette di vedere i registri anche senza attivare il modo `tui`. `info registers` fa vedere tutti i registri. Se ne vogliamo vedere uno solo basta far seguire il nome del registro: `info registers r0`, per esempio, fa vedere il solo contenuto del registro `r0` (si può abbreviare con `info reg r0`)

Esempio di esecuzione single step senza modo tui

Supponiamo di voler eseguire passo passo il programma di divisione fra interi listato nella Sez. 7.5. Lanciamo il debugger nel solito modo. Qui di sotto trovate l'output relativo all'esempio, in cui:

- linea 2: si lancia l'esecuzione del programma in modalità debug
- linea 4: si lancia il debugger multiarch
- linea 11: mettiamo un breakpoint all'etichetta `main` (entry point del codice)
- linea 13: facciamo partire il programma (è una `continue` invece che una `run` per via del fatto che stiamo utilizzando il cross compiler, vedi Sez. ??)
- line 17: il debugger si ferma mostrando il codice della prossima istruzione da eseguire (in questo caso, la prima, quella che corrisponde all'etichetta `main`)
- linea 18: `next`: esegui l'istruzione e fermati alla prossima. Diamo il comando `next` (abbreviato `n` per altre 5 volte (riga 20, 22, 25, 27 e 29). Un ritorno carrello senza immissione di caratteri al prompt di `gdb` ripete l'ultimo comando impartito.

- linea 31: dopo l'esecuzione della `sub r1 , r1 , r2` e prima dell'esecuzione della `sub r1 , r1 , r2` chiediamo quanto valga il registro `r1`: vale 14 perchè è partito da 17 e abbiamo appena fatto l'iterazione 0 della divisione
- line 33: `next` altra istruzione. A questo punto vedo che la prossima istruzione è la `b loop` e quindi il loop ricomincia. Decido di proseguire fino alla fine delle iterazioni, e quindi:
- linea 35: metto un breakpoint all'etichetta `end` e
- linea 37: continuo fino al prossimo breakpoint
- linee 42, 44: chiedo di vedere i valori di `r0` e `r1` (rispettivamente risultato e resto della divisione)
- linea 46: i risultati sono corretti, `continue`. Dal momento che non ci sono altri breakpoint attivi, il programma termina con la exit.

```
1  marcod@marcod-ThinkPad-E480
2  :~/NuovoCorso/Esercizi/Assembler/Div$ qemu-arm -g 12345 ./a.out &
3  [1] 24706
4  marcod@marcod-ThinkPad-E480
5  :~/NuovoCorso/Esercizi/Assembler/Div$ gdb-multiarch -q --nh -ex 'set architecture arm'
6   -ex 'file a.out' -ex ' target remote localhost:12345'
7  The target architecture is assumed to be arm
8  Reading symbols from a.out...
9  Remote debugging using localhost:12345
10 0x00010328 in _start ()
11 (gdb) b main
12 Breakpoint 1 at 0x10424: file div.s, line 7.
13 (gdb) c
14 Continuing.
15
16 Breakpoint 1, main () at div.s:7
17 7 main:   mov r1, #17 @ dividendo
18 (gdb) n
19 8   mov r2, #3  @ divisore
20 (gdb)
21 10    mov r0, #0  @ azzero il risultato
22 (gdb)
23 loop () at div.s:11
24 11  loop: cmp r1, r2  @ se divisore maggiore di dividendo -> fine
25 (gdb)
26 12    blt end
27 (gdb)
28 13    sub r1, r1, r2  @ altrimenti togli divisore dal dividendo
29 (gdb)
30 14    add r0, r0, #1  @ e incremente il risultato
31 (gdb) info register r1
32 r1             0xe                 14
33 (gdb) n
34 15    b loop    @ e ricomincia
35 (gdb) b end
36 Breakpoint 2 at 0x10444: file div.s, line 17.
37 (gdb) c
38 Continuing.
39
40 Breakpoint 2, end () at div.s:17
41 17  end:  mov r2, r1
42 (gdb) info register r0
43 r0             0x5                 5
44 (gdb) info register r1
45 r1             0x2                 2
46 (gdb) c
47 Continuing.
48 Il risultato e' 5 resto 2
49 [Inferior 1 (process 1) exited with code 032]
50 (gdb) q
```

```
[1]+  Exit 26                 qemu-arm -g 12345 ./a.out
marcod@marcod-ThinkPad-E480
```

7.5 Disassembler

Qualora di voglia visualizzare il contenuto di un file oggetto o eseguibile, possiamo utilizzare due comandi:

- `od`
  che può mostrare il contenuto di un file in vari formati (decimale, binario, esadecimale, ...) ma che non "disassembla" eventuali istruzioni assembler

- `objdump`
  che disassembla sia il contenuto di un file oggetto che quello di un file eseguibile. In particolare, per vedere il disassemblato di un file occorre utilizzare l'opzione `-d`:
  `objdump -d file.o`

Quando si utilizzi il cross compiler, invece che `objdump` dovremo utilizzare `arm-linux-gnueabihf-objdump`[6].
Ad esempio, supponiamo di avere il file che contiene la routine per che calcola la divisione intera di due numeri:

```
  .data
fm: .string "Il risultato vale %d resto %d\n"
  .text
  .global main
main:   mov r1, #17 @ dividendo
  mov r2, #3  @ divisore
  mov r0, #0  @ azzero il risultato

loop: cmp r1, r2  @ se divisore maggiore di dividendo -> fine
  blt end
  sub r1, r1, r2  @ altrimenti togli divisore dal dividendo
  add r0, r0, #1  @ e incremente il risultato
  b loop    @ e ricomincia

end:  mov r2, r1
  mov r1, r0
  ldr r0, =fm
  bl printf

  mov r7, #1  @ quindi esci con questo valore come esito
  svc 0
```

Se compiliamo il file con un `gcc -c div.s` otteniamo un `div.o` che risulta essere un file oggetto:

```
marcod$ arm-linux-gnueabihf-gcc -c div.s
marcod$ ls -lstr | tail -1
 4 -rw-rw-r-- 1 marcod marcod  836 nov  1 14:55 div.o
marcod$ file div.o
div.o: ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV), not stripped
marcod$
```

Richiedendo un `objdump -d` otteniamo il seguente output:

```
marcod$ arm-linux-gnueabihf-objdump -d div.o

div.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0: e3a01011  mov r1, #17
   4: e3a02003  mov r2, #3
```

[6] di solito i comandi nativi per RbArm sono disponibili in CcArm facendo procedere il nome del comando dal prefisso `arm-linux-gnueabihf`

```
11    8: e3a00000  mov r0, #0
12
13 0000000c <loop>:
14    c: e1510002  cmp r1, r2
15   10: ba000002  blt 20 <end>
16   14: e0411002  sub r1, r1, r2
17   18: e2800001  add r0, r0, #1
18   1c: eafffffa  b c <loop>
19
20 00000020 <end>:
21   20: e1a02001  mov r2, r1
22   24: e1a01000  mov r1, r0
23   28: e59f0008  ldr r0, [pc, #8]  ; 38 <end+0x18>
24   2c: ebffffe  bl  0 <printf>
25   30: e3a07001  mov r7, #1
26   34: ef000000  svc 0x00000000
27   38: 00000000  .word 0x00000000
28 marcod$
```

Come si vede, il disassemblatore mostra sia la versione esadecimale che la versione disassemblata delle parole nel file. Considerate la riga 15 del listato precedente:

```
10: ba000002  blt 20 <end>
```

Questa è un'istruzione di salto. I primi 4 bit dell'istruzione rappresentano la condizione di salto (vedi Tab. 6.3 del libro di testo). In questo caso il valore `b` rappresenta la configurazione 1101_2 ovvero la condizione `LT`[7]. I 4 bit che seguono sono utilizzati per rappresentare l'istruzione di salto (`a` $= 10_{10} = 1010_2$: i primi due bit (10) dicono che è un'istruzione di salto, i secondi che è un branch normale (non branch & link). GLi ultimi 24 bit (0x..02) sono l'offset. Tenendo conto che il PC al momento dell'esecuzione dell'istruzione contiene l'indirizzo dell'istruzione + 8 (per convenzione ARM), e che l'offset viene utilizzato come numero di parole, sommargli due significa saltare alla `mov` etichettata con la `end`. Infatti il PC + 8 sarebbe l'indirizzo della `add` alla linea 12, +1 parola sarebbe l'indirizzo della `b loop` alla riga 13 e +2 parole sarebbe l'indirizzo della `mov` alla linea 15.

Da notare che se disassemblassimo un file eseguibile, otterremmo tutta una serie di altre istruzioni che sono relative a ciò che viene linkato alle istruzioni che abbiamo scritto nel file `.s` Questa cosa si può vedere semplicemente considerando la dimensione del disassemblato:

```
1 marcod$ arm-linux-gnueabihf-gcc -c div.s
2 marcod$ arm-linux-gnueabihf-objdump -d div.o | wc
3      26      93     576
4 marcod$ arm-linux-gnueabihf-gcc  div.s
5 marcod$ arm-linux-gnueabihf-objdump -d a.out | wc
6     214    1046    7002
7 marcod$
```

Il disassemblato del file oggetto è di 26 linee, mentre quello del file eseguibile è di 214 linee. Lasciamo come esercizio l'opzione di vedere cosa ci sia effettivamente nel disassemblato dell'eseguibile.

7.6 Istruzioni compilate (*pseudo-istruzioni*)

L'assembler mette a disposizione delle istruzioni che sembrano istruzioni del linguaggio macchina ma in realtà non lo sono. Tipico esempio sono le istruzioni `push {...}` e `pop {...}` che permettono di inserire/togliere sullo/dallo stack un insieme di registri specificato fra parentesi graffe. Le due istruzioni vengono "compilate" dal programma assembler come load/store multiple (`LDM` e `STM`, vedi Sez. 6.3.7. del libro di testo).

Altro esempio è l'istruzione che permette il caricamento di costanti lunghe in un registro. L'istruzione:

```
1    ldr r0, =0xff00ff00
```

carica nel registro `r0` il valore `0xff00ff00` che rappresenta una costante da 32 bit. Dal momento che non possiamo esprimere una costante così lunga in un formato che prevede per *tutta* l'istruzione un codice da 32 bit, il programma assemblatore traduce questa istruzione nel seguente modo:

[7] che è vera se è vera $N \oplus V$, vedi Sez. 6.3.2 del libro di testo

- utilizza una parola del codice per contenere la costante da caricare, ovvero in vicinanza della posizione della `ldr r0, =...` nel codice piazza una `.data 0xff00f00f` La locazione viene scelta in modo che la `.data` venga saltata nell'esecuzione del codice
- sostituisce la `ldr r0, =0xff00ff00` con una `ldr` che utilizza come base il `PC` e come offset un offset opportuno che la faccia puntare alla `.data` di cui sopra.

Per esempio, il codice:

```
ldru.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <main>:
   0: e59f0004  ldr r0, [pc, #4]  ; c <main+0xc>
   4: e3a07001  mov r7, #1
   8: ef000000  svc 0x00000000
   c: ff00ff00  .word 0xff00ff00
```

7.7 System call

Per effettuare una system call occorre fare due cose:

- preparare i parametri da passare utilizzando le stesse convenzioni utilizzate per la chiamata di procedura/-funzione, ovvero utilizzare i registri da `r0` a `r3` (nell'ordine e per quanti ne sono necessari) per i primi 4 parametri e poi passare il resto dei parametri utilizzando lo stack.
- invocare la chiamata di sistema utilizzando una `svc 0` invece che una `bl` come si usa per chiamare procedure o funzioni. La `svc 0` ha un parametro che viene passato mediante `r7` che rappresenta il *numero* della chiamata di sistema da effetture.

La tabella che segue contiene i numeri di alcune delle syscall POSIX/Linux, fra quelle che sono più semplici da utilizzare e più utili per gli esercizi svolti nel nostro corso[8]:

Syscall	**Numero**	r0	r1	r2
exit	1	codice ritorno		
read	3	descrittore	ind buffer	dim buffer
write	4	descrittore	ind buffer	dim buffer
open	5	ind filename	flags	mode
close	6	descrittore		

(**NOTA**: i descrittori (normalmente già aperti) per standard input, output ed error sono rispettivamente 0, 1 e 2)

7.7.1 Esempio di chiamata di sistema: `read`

Supponiamo di voler leggere dei caratteri dallo standard input. La chiamata di sistema che legge caratteri da un file è (`man 2 read`):
`ssize_t read(int fd, void *buf, size_t count);` dove `ssize_t` è da intendersi sinonimo di intero a 32 bit, nel nostro specifico caso.

Lo standard input è sempre rappresentato dal descrittore (`fd`) 0.

Il codice che segue esegue una lettura da terminale:

```
  .data
buf:  .fill 1024                  @buffer da 1023 caratteri
msg:  .string "Immetti un numero:\n"   @ stringa di prompt
ris:  .string "Immessa %s (vale %d)\n" @ formato output programma

  .text
```

[8] la lista di tutte le syscall, con i loro parametri, la trovate a `https://syscalls.w3challs.com/?arch=arm_thumb`

```
7   .global main
8
9  main: push {lr}   @ salvo il punto di ritorno del main
10
11    ldr r0, =msg
12    bl printf   @ stampo il prompt (chiamata di fun)
13          @ adesso chiamo la read
14    mov r0, #0    @ parametri read : descrittore fd=0
15    ldr r1, =buf    @ parametri read : indirizzo buffer
16    mov r2, #1
17    lsl r2,r2,#10   @ parametri read : lunghezza buffer
18    mov r7, #3    @ read e' la syscall 3
19    svc 0       @ questa e' una chiamata di syscall
20    sub r0, r0, #1    @ r0 e' il numero di caratteri letti
21    ldr r1, =buf    @ rendo la stringa NULL terminated
22    mov r2, #0    @ costante NULL in r2
23    str r2, [r1,r0]   @ NULL al posto del \n
24    mov r0, r1    @ adesso la converto in intero
25    bl  atoi
26    mov r2, r0    @ val della stringa terzo param printf
27    ldr r0, =ris    @ primo param ind stringa di formato
28    ldr r1, =buf    @ indirizzo del buffer con stringa letta
29    bl printf   @ chiamo le print
30    pop {lr}    @ prima di restiuire il controllo,
31          @ ripesco l'inidirizzo di ritorno dallo stack
32          @ (LR sporcato dalle bl a printd e atoi ...)
33    bx lr     @ e ritorno
```

Il programma stampa un prompt (`printf` della linea 12) e chiama una `read` (`svc 0` della linea 19). L'utente deve digitare a questo punto un numero intero che verrà memorizzato come caratteri ASCII nell'area di memoria riservata con la `fill` all'indirizzo `buf`. Normalmente l'input è terminato con un ritorno carrello. Se immettiamo i tre caratteri '1', '2' e '3' seguiti da un ritorno carrello, la read restiuirà 4 (numero di caratteri letti) e il contenuto del buffer `buf` sarà (linee ottenute utilizzando il debugger):

```
1  (gdb) info reg r1
2  r1             0x2102c             135212
3  (gdb) x/8cb 135212
4  0x2102c:  49 '1'  50 '2'  51 '3'  10 '\n' 0 '\000'  0 '\000'  0 '\000'  0 '\000'
5  (gdb)
```

Con le linee 20–23 andiamo a sostituire il `\n` finale con un NULL:

```
1  (gdb) n
2  21    ldr r1, =buf    @ rendo la stringa NULL terminated
3  (gdb)
4  22    mov r2, #0    @ costante NULL in r2
5  (gdb)
6  23    str r2, [r1,r0]   @ NULL al posto del \n
7  (gdb)
8  24    mov r0, r1    @ adesso la converto in intero
9  (gdb) x/8cb 135212
10 0x2102c:  49 '1'  50 '2'  51 '3'  0 '\000'  0 '\000'  0 '\000'  0 '\000'  0 '\000'
11 (gdb)
```

quindi invochiamo una `atoi` (funzione di libreria `glibc`) per convertire la stringa in un intero:

```
1  (gdb) n
2  25    bl  atoi
3  (gdb)
4  26    mov r2, r0    @ val della stringa terzo param printf
5  (gdb) info reg r0
6  r0             0x7b                123
7  (gdb)
```

e finalmente stampiamo il messaggio che riporta stringa letta e valore calcolato (`printf` alla linea 29).

Notare che il programma funziona regolarmente (anche in presenza della syscall `read`) in entrambi gli ambienti (RbArm e CcArm) e in entrambi i modi (esecuzione diretta, esecuzione mediante debugger). E' possibile che eseguendolo col debugger prompt e caratteri immessi "sporchino" il layout delle finestre ottenuto mediante `tui reg`.

7.8 Terminazione di un programma

Un programma scritto in assembler si comporta come un programma scritto in C, ai fini della terminazione. Può terminare in due modi:

- con una `return`, dal momento che il `main` viene chiamato dal codice nel preambolo linkato dal `gcc` come ogni altra funzione. In questo caso è il codice della libreria C (anche nel caso il `main` sia scritto direttamente in assembler come nei nostri programmi) che si preoccupa della terminazione e di restituire il controllo al sistema operativo;
- con una `exit`; in questo caso è direttamente il `main` che invoca la `exit` per terminare il processo in esecuzione e resituire così il controllo al sistema operativo.

In assembler ARMv7 le due opzioni corrispondono a codici diversi.

- La `return` può essere implementata in diversi modi. Tipicamente con una `mov pc,lr`, una `mov r15, r14` o una `bx LR`. In tutti a casi, il registro `r0` conterrà il codice di ritorno della return;
- la `exit` richiede una syscall ed è implementata come spiegato nella Sez. 7.7 mediante il codice

```
1     mov R7,#1   @ numero di call = 1
2     svc 0       @ super visor call 0
```

Naturalmente anche la `exit` accetta in `r0` il valore da passare come codice di ritorno (`exit(codrit)`).

7.9 Docker

Il listato che segue è quando dovete includere nel `Dockerfile` per creare un docker con tutti gli strumenti necessari per compilare ed eseguire codice ArmV7 sia su macchine Windows/Linux con Intel che su macchine Mac OS/X sempre con processori Intel.

```
1  ### start from current ubuntu image
2  FROM ubuntu:groovy
3  MAINTAINER marcod
4  ### install base programming tools
5  RUN apt update
6  RUN apt install -y vim nano gcc-arm-linux-gnueabihf qemu-user gdb-multiarch
7  ### set up service ssh and net tools
8  RUN apt install -y ssh openssh-server net-tools
9  RUN service ssh start
10 ### fix user
11 RUN useradd -ms /bin/bash ae2020
12 USER ae2020
13 WORKDIR /home/ae2020
```

Seguendo le istruzioni sui tutorial Docker (e.g. `https://docs.docker.com/get-started/part2/`) potete semplicemente creare un container docker, utilizzarlo per esercitarvi con l'assembler e anche montare delle directory locali in modo che possano essere viste all'interno del container ed evitare così la necessità di copiare i file nel container e dal container.

Per esempio:

- `docker build -t marcod/ae2020 .` eseguito in una directory dove c'è il `Dockerfile` con i contenuti visti sopra crea il container
- `docker run -it marcod/ae2020 bash` lancia il container con una shell interattiva in cui potete utilizzare i cross-tool discussi in questo capitolo
- `docker run --mount src=/home/marcod/.../dirlocale, target=/home/ae2020/Dummy,type=bind -it marcod/ae2020 bash`
  lancia il container con la directory locale visibile come sottodirectory `Dummy` della home nel container

Notare che potrebbe essere necessari i diritti di root per compiere queste operazioni. Per uscire dal container basta lanciare il comando `exit` dalla shell. **Attenzione:** *eventuali file creati nel container (non nella eventuale directory montata come illustrato sopra) verranno persi!*

Capitolo 8

ARMv8

Queste note descrivono (per differenza con ARMv7) il sottoinsieme delle istruzioni che servono per la realizzazione dei semplici esercizi di programmazione assembler del corso di Architettura degli Elaboratori. L'insieme di istruzioni supportato in ARMv8 è più esteso di quanto non emerga da queste note.

8.1 Register file

Il register file ARMv8 è di 31 registri a 64 bit. I registri si chiamano `x0` ... `x30`. L'ultimo registro si chiama `xzr` e vale sempre 0 (è cablato a 0). La parte bassa dei registri `xi` si può riferire utilizzando il nome `wi`. Lavorando con i registri a 32 bit `wi` si può utilizzare il registro che vale sempre 0 con il nome `wzr`. Quando si carica un registro da 32 bit con una `ldr` la parte alta del registro viene automaticamente azzerata. Più precisamente, quando si usano i registri `wi` come sorgenti la parte superiore del registro `xi` è ignorata. Nel caso il registro venga utilizzato come destinazione, la parte superiore del registro `xi` viene azzerata.

Nel file dei registri non sono più presenti il program counter (PC) e lo stack pointer (SP), che diventano registri dedicati e possono essere riferiti rispettivamente come `PC` e `SP`. Rimane il registro `x30` che viene di default utilizzato come link register (e che può essere riferito con `LR`).

Lo stack pointer viene controllato ad hardware in ARMv8 e deve sempre essere un multiplo di 16. Non ci sono più le `push` e `pop`.

8.2 Direttive

Le direttive sono le stesse che abbiamo utilizzato per ARMv7 (aarch32). Va considerata la possibilità di utilizzare `.quad` per definire valori in memoria rappresentati su 64 bit. In particolare:

```
1    .data
2 x: .word 1,2,3,4
```

definisce un'area di memoria di 4 parole da 32 bit con base all'indirizzo `x` che contiene, nell'ordine, i valori 1, 2, 3 e 4. Invece il codice:

```
1    .data
2 y: .quad 1, 2, 3, 4
```

definisce un'area di memoria da 4 parole da 64 bit. Per caricare in un registro (e.g. `x2`) il valore da 32 bit `x[0]` possiamo utilizzare il codice

```
1   ldr x0, =x
2   mov x1, #1
3   ldr w2, [x0, x1]
```

che lascia nel registro `x2` il valore `0x0000000000000001`, dal momento che `w2` rappresenta la parte bassa di `x2`. Per caricare nel registro `x5` il valore da 64 bit `y[2]` possiamo invece utilizzare il codice:

```
1  ldr x3, =y
2  mov x4, #2
3  ldr x5, [x3, x4, lsl #3]
```

(Si noti che l'indice è moltiplicato per 8 in modo da indirizzare parole da 64 bit (8 byte), visto che la memoria rimane indirizzata al byte.

8.3 Istruzioni

La maggior parte delle istruzioni ARMv7 sono presenti anche in ARMv8. Una grossa differenza è costituita dall'assenza delle istruzioni condizionali. Non c'è più la possivilità di esecuzione condizionale delle operative (aritmetico logiche) ne' delle operazioni che operano sulla memoria (load e store). Rimangono i salti condizionali (branch e branch and link). Solo un certo numero di istruzioni operative (oltre la `cmp`) possono essere utilizzate per settare i flag di condizione: ADD/ADC, NEG/NGC, SUB, AND, BIC.

E' stata introdotta una istruzione `ret` che può prendere un parametro di tipo registro che di fatto sostituisce il `PC` con il valore di `LR` (se il registro non è presente) o col valore contenuto nel registro (se presente).

Le istruzioni di load e store possono operare con uno o due registri come sorgente (store) o destinazione (load) e possono operare su byte, parole da 64, 32 e 16 bit. Nel caso di byte e valori da 16 bit caricati in registri da 32 bit (`wi`) i valori possono essere estesi in segno.

8.4 Parametri per le chiamate di procedura/funzione

Avendo molti più registri a disposizione cambiano le convenzioni per le chiamate di funzione/procedura.

- i registri da `x0` a `x7` sono utilizzati per i parametri e non è richiesto al codice chiamato preservane il valore
- i registri dal `x19` in poi sono per i temporanei e il loro contenuto va preservato a carico del codice chiamato
- `x30` è utilizzato come `LR`
- i registri da `x8` a `x18` hanno usi diversi (vedi documentazione)

8.5 Tools

Sotto Linux, possiamo compilare ed eseguire assembler ARMv8 utilizzando il cross compiler GNU. Oltre alle cose da installare per ARMv7, va installato il pacchetto compilare per AARCH64, con un comando (assumiamo di lavorare sotto Ubuntu o Debian):

```
apt install gcc-aarch64-linux-gnu
```

L'installazione va fatta da shell root, come al solito. Questo pacchetto mette a disposizione i comandi

```
aarch64-linux-gnu-*
```

che sono gli stessi comandi che il pacchetto per ARMv7 metteva a disposizione come

```
arm-linux-gnueabihf-*
```

Per compilare ed eseguire un programma ARMv8 completo i passi consistono quindi:

- nel compilare il (o i) file assembler (estensione `.s`) con un comando `aarch64-linux-gnu-gcc simple.s -static -ggdb3 -`
  (il flag `-ggdb3` serve solo per il debugger)
- eseguire l'eseguibile risultante con un comando `qemu-aarch64 file.exe`
- se si vuole eseguire il programma col debugger, prima lanciare il comando in modalità debug col comando
  `qemu-aarch64 -g 12345 file.exe &` e successivamente lanciare il debugger con il comando
  `gdb-multiarch -q --nh -ex 'set architecture aarch64' -ex 'file file.exe' -ex 'target remote localhos`

I pacchetti `qemu` e `gdb-multiarch` devono essere installati come per ARMv7.

8.6 Esempi di codice

8.6.1 Semplice codice com chiamata di funzione

Il codice che segue somma due costanti e stampa il risultato, restituendo poi il controllo al chiamante. I commenti illustrano le principali differenze rispetto ad una ipotetica versione ARMv7.

```
  .text       /* commenti come in C */
  .global main      /* le pseudo istruzioni sono le stesse */


main: mov x0, #12     /* i registri si chiamano xi invece che ri */
  mov x1, #23
  add x2, x1, x0
  adds x0, x9, xzr    /* registro xzr sempre uguale a 0  */
  bne fine
ritorno:        /* chiamate di funzione sono come al solito, cambiano i nomi
             dei registri, ovviamente: x0-x7 param e retval ... */
  mov x2, x2      /* parametri sempre nei primi registri */
  ldr x0, =f
  sub sp, sp, #16     /* salvo link register per ritornare al punto giusto */
  str x30, [sp,#8]        /* push {x30} */
  bl printf
ciccio: ldr x30, [sp, #8]       /* pop {x30}  */
  add sp, sp, #16
  ret       /* ret invece che mov su PC: posso anche indicare il reg LR */
fine:   b ritorno

  .data
f:  .string "Risultato della somma %d\n"
```

8.6.2 IP

Il codice che segue permette di calcolare l'inner product fra due vettori di numeri interi da 64 bit definiti come costanti nella sezione `.data`

```
        .data
x:      .quad   1,2,3,4,5,6,7,8
y:      .quad   8,7,6,5,4,3,2,1
n:      .quad   8,0
f:  .string "IP = %d\n"

formato:.string "Il risultato vale %d\n"

        .text
        .global main
main:

  ldr x0, =x
  ldr x1, =y
  ldr x3, =n
  ldr x3, [x3]

  mov x4, xzr
  mov x5, xzr
for:  cmp x4, x3
  beq fine
  ldr x6, [x0, x4, lsl #3]
  ldr x7, [x1, x4, lsl #3]
  mul x7, x6, x7
  add x5, x5, x7
  add x4, x4, #1
  b for
fine:   mov x1, x5
  ldr x0, =f
  bl printf
```

```
31   mov x0, #0
32   mov x8, #93
33   svc #0
```

I vettori sono definiti utilizzando `.quad` per fare in modo che ad ogni valore vengano assegnati 64 bit (8 byte). Tutti i registri utilizzati, per gli indirizzi, per i valori caricati dalla memoria e per le altre variabili temporanee (variabile di iterazione, somma parziale) sono registri a 64 bit. Il risultato finale è stampato utilizzando una `printf`. Il programma termina con una `exit(0)` (il numero della syscall è 93, ed è passato in `x8`, mentre i parametri della syscall, come per le altre funzioni, sono passati nei registri da `x0` in poi).

Se avessimo voluto continuare ad utilizzare dati a 32 bit, avremmo potuto utilizzare il codice che segue:

```
1   .data
2 x:      .word   1,2,3,4,5,6,7,8   /* tutte parole da 32 bit */
3 y:      .word   8,7,6,5,4,3,2,1
4 n:      .word   8
5 f:  .string "IP = %d\n"
6
7         .text
8         .global main
9 main:   ldr x0, =x      /* indirizzi a 64 bit */
10  ldr x1, =y
11  ldr x3, =n
12  ldr w3, [x3]      /* carico una parola da 32 nella parte bassa di x3 */
13
14  mov x4, xzr
15  mov x5, xzr
16 for:  cmp x4, x3
17  beq fine
18  ldr w6, [x0, x4, lsl #2]  /* carico x[i] a 32 bit nella parte bassa di x6, la parte alta
      viene azzerata */
19  ldr w7, [x1, x4, lsl #2]
20  mul x7, x6, x7      /* moltiplicazione 64  bit * 64 bit -> 64 bit */
21  add x5, x5, x7
22  add x4, x4, #1
23  b for
24 fine:   mov x1, x5      /* call della printf, coi parametri nei primi registri */
25  ldr x0, =f
26  bl printf
27  mov x0, #0      /* exit(0) */
28  mov x8, #93
29  svc #0
```

Le load vengono effettuate in registri a 32 bit. La parte alta dell'omonimo registro da 64 bit viene inizializzata a 0. Una volta caricati tutte le operazioni che calcolano la moltiplicazione e la somma parziale e finale sono fatte a 64 bit utilizzando i registri `xi` corrispondenti.

8.6.3 Somma del valore dei caratteri che rappresentano interi in una stringa ASCII

Questo è il codice relativo all'esercizio SumInt della quarta prova di verifica intermedia. In una stringa, si cercano tutti i caratteri fra 0 e 9 e se ne somma il valore, calcolandone anche il numero. Il codice ASMv8 seguente calcola esattamente questo:

```
1   .data
2 s:  .string "abc123def456ghi7890cba"
3 f:  .string "La somma di %d numeri nella stringa %s vale %d\n"
4
5   .text
6   .global main
7
8 main: sub sp, sp, #16
9   str LR, [sp, #8]
10  ldr x0,=s
11  bl strlen /* x0 = strlen(s) */
12  ldr x1,=s /* x2 = @s        */
```

```
  mov w2,wzr  /* w2 = i <= 0    */
  mov w4,wzr  /* w4 = somma     */
  mov w5,wzr  /* w5 = #numeri   */

loop: cmp w2, w0
  beq end
  ldrb w3, [x1,x2]
  sub  w3, w3, #0x30
  cmp  w3, #0
  blt fine
      cmp w3, #9
  bgt fine
  add w4,w4,w3
  add w5,w5,#1
fine: add x2,x2,#1
  b loop

end:  ldr x0, =f
  mov x1, x5
  ldr x2, =s
  mov x3, x4
  bl printf

  ldr LR, [sp,#8]
  add sp,sp,#16
  ret
```

Abbiamo utilizzato registri a 32 bit per la somma, la variabile di iterazione e il contatore dei caratteri. Come al solito, i caratteri della stringa sono letti byte a byte (il codice ASCII usa un byte per carattere).

L'unico registro salvato sullo stack è il LR, visto che poi si chiama la `strlen` e quindi verrebbe sovrascritto. Per salvare LR sullo stack prima si alloca dello spazio (lo stack cresce verso il basso) in multipli di 16, poi si carica nella prima posizione libera LR sfruttando l'offset. Prima di resituire il controllo anìl chiamante, si recupera il LR dallo stack e si dealloca lo spazio riservato riportanto lo SP alla posizione iniziale.

Il programma termina resituiendo il controllo al codice che ha invocato il `main` (preambolo gcc).

8.6.4 Sort dei caratteri di una stringa passata come parametro da riga di comando

Il listato che segue implementa un ordinamento dei caratteri di una stringa passata come parametro `argv[1]`.

```
  .data
f:  .string "La stringa >> %s << e' lunga %d caratteri (%d scambi effettuati)\n"
  .text
  .global main

main: cmp x0, #1     /* se argc == 1 */
  beq fine           /* restituisci il controllo al chiamante senza fare niente */
  sub sp, sp, #16    /* altrimenti prepara spazio sullo stack */
  str lr, [sp, #8]   /* e salva il LR che verra' sporcato dalla chiamata alla strlen */

  ldr x0, [x1, #8]   /* argv[1]  indirizzo della stringa (+8 perche' puntatori a 64 bit) */
  str x0, [sp]       /* salvalo sullo stack per dopo */
  bl strlen      /* calcolo della lunghezza della stringa */

  /* bubble sort : for(i=0; i<n-1; i++) for(j=i+1; j<n; j++) if(s[i]>s[j]) swap; */
  mov x6, xzr      /* conto degli swap */
  ldr x3, [sp]       /* indirizzo della stringa dallo stack */
  mov x1, xzr    /* i */
  sub x2, x0, #1     /* n - 1 */
loopi:  cmp x1, x2     /* controllo fine ciclo : i<n-1 */
  beq finei

  add x2, x1, #1     /* j = i + 1*/
loopj:  cmp x2, x0         /* controllo fine ciclo : j<n */
  beq finej
  ldrb w4, [x3, x1]  /* s[i], serve un registro a 32 bit come destinazione del byte */
```

```
27   ldrb w5, [x3, x2]  /* s[j], idem */
28   cmp x4, x5     /* confronto */
29   ble cont     /* se sono gia' in ordine non scambia */
30   strb w4, [x3, x2]  /* altrimenti scambia */
31   strb w5, [x3, x1]
32   add x6, x6, #1     /* incrementa il conto degli scambi */
33 cont: add x2, x2, #1     /* j++ */
34   b loopj
35 finej:  add x1, x1, #1     /* i++ */
36   b loopi
37 finei:                    /* finito stampa la stringa ordinata */
38   ldr x1, [sp]       /* indirizzo della stringa */
39   mov x2, x0         /* lunghezza */
40   mov x3, x6         /* numero degli scambi */
41   ldr x0, =f         /* fmt string */
42   bl printf
43         /* adesso ritorna, tutti i registri utilizzati sono temporanei */
44   ldr lr, [sp, #8]   /* recupera link register */
45   add sp, sp, 16     /* rimette a posto lo stack, non interessa strlen */
46
47 fine: ret                /* restituisce il controllo al chiamante */
```

Prima controlliamo che ci sia un parametro (linee 6–7) e se non c'è si termina immediatamente. Successivamente si salva sullo stack il `LR` (righe 8–9). Quindi si recupera l'indirizzo della stringa e lo si salva sullo stack (righe 11-12) e si chiama la `strlen`. Da lì in poi il codice è molto simile a quello utilizzato per la quarta verifica intermedia. L'unica nota di rilievo riguarda l'uso dei registri da 32 bit per caricare i caratteri con le `ldrb`. Se si utilizzassero come destinazione i registri da 64 bit (`xi`) otterremo dei messaggi di errore:

```
1 bbstring.s: Assembler messages:
2 bbstring.s:27: Error: operand mismatch -- `ldrb x4,[x3,x1]'
3 bbstring.s:27: Info:    did you mean this?
4 bbstring.s:27: Info:      ldrb w4, [x3, x1]
```

dal momento che l'assembler non supporta il caricamento di un registro da 64 bit con un byte. Notare anche il modo usato per recuperare `LR` dallo stack e liberare le due posizioni occupate (linee 44–45).

8.7 Risorse (online)

La documentazione ufficiale Arm si trova praticamente tutta sul sito `developer.arm.com`. In particolare, alla pagina `https://developer.arm.com/documentation/ddi0487/latest/` trovate il manuale di riferimento per l'architettura ARMv8. La pagina `https://developer.arm.com/architectures/cpu-architecture/a-profile` riporta le principali caratteristiche dell'architettura V8. La pagina `https://developer.arm.com/architectures/learn-the-architecture/aarch64-instruction-set-architecture/single-page` permette di accedere a tutte le informazioni sull'instruction set.

Il sito `https://modexp.wordpress.com/2018/10/30/arm64-assembly/` è una cosa dove vengono spiegate le caratteristiche principali della archeittura a 64 bit.

Al sito `https://courses.cs.washington.edu/courses/cse469/19wi/arm64.pdf` trovate una quick reference card delle istruzioni ARMv8.

Parte III

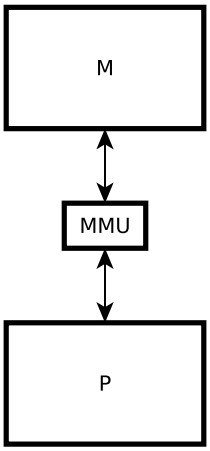
Microarchitettura

Capitolo 9

MMU

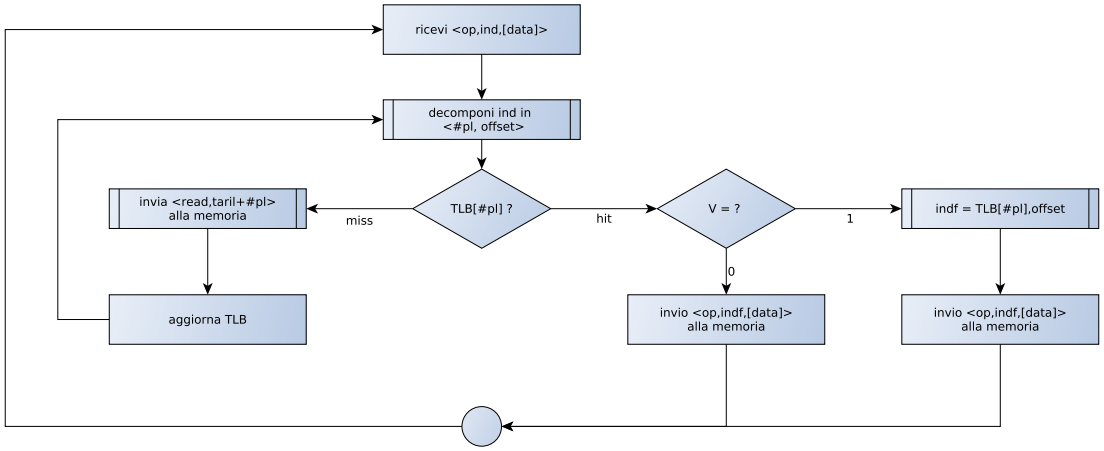
La traduzione dallo spazio di indirizzi virtuale a quello fisico è di norma responsabilità della TLB (*Translation Lookaside Buffer*), come descritto nel par. 8.4.3 del libro di testo. La TLB in realtà altro non è che una piccola unità cache che permette di effettuare la ricerca nella tabella delle pagine in modo molto efficiente.

Dal punto di vista didattico è più intuitivo forse pensare ad una vera e proprià unità (MMU, *Memory Management Unit*) interposta fra processore e sottosistema di memoria, come illustrato in figura.



La MMU in particolare conterrà una TLB, ovvero una cache completamente associativa che permette di mantenere la parte della tabella delle pagine del processo che rappresenta le corrispondenze fra numero di pagina logica e numero di pagina fisica per le pagine che fanno parte in quel momento del working set del processo.

La MMU colloquia col processore e col sottosistema di memoria e implementa il flowchart in figura.



Da notare che la MMU deve conoscere l'indirizzo base in memoria della tabella delle pagine. La tabella delle pagine è una struttura dati "di processo"[1] e questa informazione dovrà essere comunicata alla MMU dal processore P nel momento in cui si effettua la commutazione di contesto (il processo corrente viene (ri)mandato in esecuzione). Nell'ipotesi che l'indirizzo base della tabella delle pagine risieda in un apposito registro `ind_tabpag`, l'indirizzo di cui si chiederà la lettura in caso di fault di TLB al sottosistema di memoria sarà `ind_tabpag + #pl`. Il numero di pagina logica `#pl` deriva direttamente dalla parte alta dell"indirizzo logico. La tabella delle pagine avrà tante entry quante sono le pagine logiche del processo (testo, ovvero istruzioni, e dati, sia statici che dinamici). Ciascuna entry della tabella delle pagine sarà una parola che contiene il numero di pagina fisica corrispondente alla pagina logica e il bit di validità.

In particolare, la MMU:

- riceve da P indirizzi da tradurre, li traduce e li invia al sottosistema di memoria implementare l'operazione richiesta (load/fetch o store);

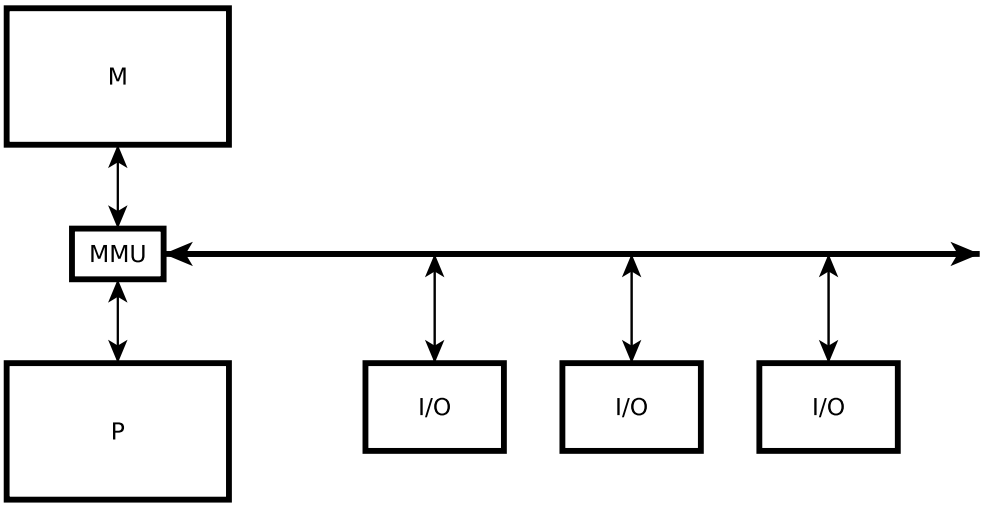
- comunica a P eventuali *fault di pagina*, ovvero tentativi di traduzione di indirizzi che corrispondono a pagine logiche che non sono state ancora caricate in memoria principale, e quindi non hanno una corrispondente pagina fisica valida nella tabella delle pagine;

- manda al sottosistema di memoria indirizzi fisici, associati alle operazioni richieste dal processore (fetch/load e store). In questo caso le riposte (dato letto, per la load/fetch o esito della scrittura) vengono passate direttamente a P;

- manda al sottosistema di memoria richieste di letture di elementi della tabella delle pagine non presenti nella TLB interna.

Visto il ruolo riservato alla MMU, viene naturale pensare che la MMU possa anche occuparsi della redirezione delle operazioni di lettura/scrittura destinate alle memorie interne dei dispositivi di ingresso/uscita sul bus di I/O. In questo caso, lo schema diventa quello della figura che segue:

[1] ogni processo ne ha una diversa



La MMU conosce l'insieme degli indirizzi riservati per l'ingresso/uscita nello spazio di indirizzamento virtuale[2] e ogni volta che ne traduce uno controlla se l'indirizzo appartenga o meno allo spazio dell'ingresso uscita:

- se l'indirizzo è relativo allo spazio di ingresso/uscita, l'indirizzo tradotto e tutti i dati relativi all'operazione richiesta (load o store, eventuale dato da scrivere) sono rediretti sul bus di I/O;
- diversamente si passa tutto al sottosistema di memoria.

9.1 MMU in Verilog

Il listato che segue fa vedere un esempio di implementazione della MMU in Verilog. La MMU non tratta gli indirizzi di I/O e assumiamo di avere in qualche modo noto l'indirizzo della tabella delle pagina in memoria. Inoltre non vengono considerati particolari protocolli di comunicazione con memoria e processore (si assume di avere interazioni sincrone e indicatori a livelli).

```
1  //
2  // mmu: interazione diretta col processore
3  //      non ci sono meccanismi di sincronizzazione, che andrebbero comunque presi in
       considerazione
4  // consideramo pagine da 4K (come sul libro), quindi 12 bit di offset e 20 bit di numero di
       pagina
5  // indirizzi logici e fisici da 32 bit e dati da 32 bit
6  //
7  module MMU(// interfaccia processore
8       input [31:0]      indlogico,      // indirizzo da tradurre
9       output reg        faultdipagina,  // viene passato al processore in caso di fault di
       pagina
10      input       op,             // 0 -> read, 1 -> write
11      input [31:0]      data,           // da scrivere, in caso
12      // interfaccia memoria
13      output reg [31:0] indfisico,
14      output reg [31:0] dataout,        // serve solo per passare eventuali dati in scrittura
15      output reg        opout,
16      input [31:0]      datain,
17      output reg        reqm,
18      input       rdym,
19      input       clock);         // clock
20
21    // indirizzo : 12 bit offset di pagina e 20 numero di pagina
22    // questa e' la mia cache da 4 posizioni//
```

[2] potrebbe essere anche lo spazio fisico, dipende da quando viene fatto il controllo, se prima o dopo la traduzione dell'indirizzo da virtuale a fisico

```
23   reg [19:0]          ipl[4];      //   numero di pagina logica (chiave)
24   reg [19:0]          ipf[4];      //   numero di pagina fisica (valore)
25   reg           p[4];        //   bit di presenza (valore)
26
27
28   // registro di stato
29   // stato 0: accetta richieste dal processore
30   // stato 1: richiede pagina alla memoria
31   // stato 2: aggiorna cache e ricomincia
32   //
33   reg [1:0]         stato;
34
35   // registri interni
36   reg           faultcache ;  // segnale di fault
37   reg [1:0]         quale;        // uscita codificatore
38   reg [31:0]          tabril;       // base in memoria della tabella di rilocazione del
      processo
39   reg [1:0]         vittima;      // entry da selezionare per il rimpiazzo (politica rr)
40
41
42   initial
43     begin
44  // inizializzazioni necessarie
45  reqm          <= 0;            // nessuna richiesta diretta al sottosistema di memoria
46  faultdipagina <= 0;            // nessuna communicazione di fault di pagina diretta a P
47  stato         <= 0;            // stato iniziale, attesa richiesta di traduzione indirizzo
48  vittima       <= 0;            // gestione round robin delle associazioni
49  tabril        <= 32'd16384;    // inizializzato all'esecuzione del processo
50
51  // inizializzazione della cache TLB
52  // + IPL (chiave)       +   IPF                 + P (bit V nel libro) +
53  ipl[0] = 22'd16;  ipf[0] = 22'd1024;  p[0] = 1; //  |  0x10        | 0x200   | 1 |
54  ipl[1] = 22'd128; ipf[1] = 22'd16;  p[1] = 1; //  |  0x80        | 0x010   | 1 |
55  ipl[2] = 22'd0;   ipf[2] = 22'd8192;  p[2] = 1; //  |  0x00        | 0x2000  | 1 |
56  ipl[3] = 22'd1;   ipf[3] = 22'd8193;  p[3] = 1; //  |  0x01        | 0x2001  | 1 |
57     end
58
59   always @(posedge clock)
60     begin
61  case (stato)
62    2'd0: begin // stato iniziale, accetta richieste di traduzione indirizzi
63
64       // cerca indirizzo fisico nella cache
65       // controlla fault
66       faultcache = ~(indlogico[31:12] == ipl[0] |
67          indlogico[31:12] == ipl[1] |
68          indlogico[31:12] == ipl[2] |
69          indlogico[31:12] == ipl[3]);
70
71       // se non fault passa sopra indirizzo fisico
72       if(!faultcache)
73         begin
74      // e nel caso quale sia l'hit
75      quale = 0;  // default, tanto non serve
76      if(indlogico[31:12] == ipl[0])
77        quale = 0;
78      if(indlogico[31:12] == ipl[1])
79        quale = 1;
80      if(indlogico[31:12] == ipl[2])
81        quale = 2;
82      if(indlogico[31:12] == ipl[3])
83        quale = 3;
84
85      if(p[quale] == 1)
86        begin
87           $display("hit in TLB[%d]: IPL %d -> IPF %d ",
88        quale, ipl[quale],ipf[quale]);
89           indfisico <= {ipf[quale],indlogico[11:0]};
```

```
90            opout <= op;
91            dataout <= data;
92            reqm <= ~reqm;
93            // rimane nello stesso stato iniziale
94            stato <= 0;
95          end   // if (p[quale] == 1)
96        else    // non c'e' fault ma il bit di presenza dice che la pagina non e' in memoria
97          begin // restituire un fault di pagina da gestire ad OS
98             faultdipagina <= ~faultdipagina;
99             // a ritorna immediatamente allo stato "normale" per tradurre altri indirizzi
100            stato <= 0;
101         end
102          end
103        else // in caso di fault
104          begin // richiedi lettura della posizione corrispondente alla pagina logica nella
     tabril alla memoria
105       $display("fault TLB: IPL %d\ncerco all'indirizzo %d+%d ",indlogico[31:12], tabril,
     indlogico[31:12]);
106       indfisico <= tabril + indlogico[31:12];
107       opout <= 0;
108       reqm <= ~reqm;
109       // passa allo stato successivo di attesa dati dalla memoria di livello superiore
110       stato <= 1;
111          end
112     end // case: 2'd0
113     2'd1: // in questo caso attendo una parola dalla memoria
114       begin
115          if(rdym == 1)
116      begin
117         // il dato in ingresso e' il numero di pagina fisica con il suo bit di presenza
118         $display("Ricevuto da M IPF %d per IPL %d",
119            datain[21:0], indlogico[31:12]);
120         ipf[vittima] = datain[21:0];
121         p[vittima] = datain[22];
122         ipl[vittima] = indlogico[31:12];
123         // posso tornare allo stato inziale e ricominciare
124         stato = 0;
125         // cambiando la vittima
126         vittima = vittima+1;
127      end
128       end // case: 2'd1
129   endcase // case (stato)
130
131      end // always @ (posedge clock)
132
133 endmodule // MMU
```

Capitolo 10

Ingresso uscita

In questo capitolo delle note vediamo tre cose in particolare, che sul libro di testo sono solo accennate oppure non sono neppure trattate: memory mapped I/O, DMA e trattamento delle interruzioni. Tutte e tre hanno un ruolo fondamentale nella gestione dell'ingresso/uscita.

10.1 Memory mapped I/O

Per memory mapped I/O si intende la possibilità di accedere ed interagire con i dispositivi di ingresso uscita utilizzando le istruzioni messe a disposizione dal linguaggio assembler per interagire con la memoria, ovvero le istruzioni di load e store.

L'astrazione che utilizziamo per modellare un dispositivo di ingresso uscita è quella dell'unità firmware, ovvero un dispositivo hardware che quando viene acceso comincia ad eseguire un ciclo infinito nel quale attende che gli venga ordinato di eseguire un comando, lo esegue, ne riporta i risultati e ricomincia la prossima iterazione:

```
dispositivoI/O:
  while(true) {
    read(comando, parametri);
    results = exec(comando, parametri);
    writeback(results);
  }
```

Consideriamo una dispositivo di input qual'è la tastiera. Il tipico comando che gli può essere impartito è quello di lettura di in carattere. L'esecuzione del comando comporta l'attesa dello schiacciamento di un tasto da parte dell'utilizzatore della tastiera. Il risultato da restituire, quando il tasto sia stato effettivamente premuto, è il codice del tasto che è stato schiacciato. Supponiamo per il momento che questa sia l'unica operazione che possa essere richiesta al dispositivo tastiera.

Il dispositivo disporrà tipicamente di alcuni registri che permettono di memorizzare comandi, parametri e risultati delle operazioni che sono in grado di eseguire. Nel caso della tastiera ci possiamo immaginare di avere:

- un registro da 1 bit che sia interpretato come ordine di lettura di un tasto: se il bit è a 0, non è richiesta alcuna lettura, se il bit è a 1, deve essere letto un carattere;
- un registro lungo abbastanza per contenere il numero di bit utilizzati per rappresentare uno qualunque dei tasti che possono essere premuti. Su una tastiera tradizionale, il codice relativo ad un tasto potrebbe corrispondere alle sue coordinate riga/colonna. Di norma ci sono 6 righe di tasti (compresi i tasti Fn) e una quindicina di colonne diverse (spesso non perfettamente verticali). Per rappresentare un tasto serviranno quindi ca 21 bit. Immaginiamo che il tasto possa dunque essere rappresentato su una configurazione da 32 bit, contando anche che servono altri bit per indicare se il tasto è stato premuto insieme a uno shift, control, alt, meta, fn ...

Con queste ipotesi, il ciclo di funzionamento del nostro dispositivo tastiera potrebbe essere:

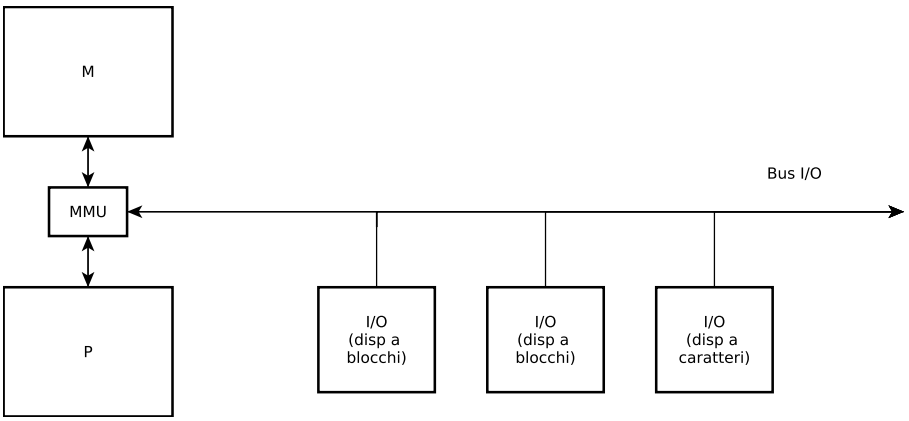


Figura 10.1: Bus I/O

```
while(true) {
  if(ordinedilettura == 1) {
     ... attendi pressione tasto ...;
     codice = codice(tastoPremuto);
     0 -> ordinedilettura;
  }
}
```

La tecnica del memory mapped I/O permette di utilizzare le normali istruzioni di load e store (LDR e STR nel caso di ARM) per andare a scrivere e leggere i registri interni dell'unità. In pratica, supponiamo di avere a disposizione degli indirizzi codificati in modo da essere dirottati sull'unità di I/O invece che nella memoria. Per esempio, immaginiamo che i registri della nostra tastiera corrispondano agli indirizzi 1024 e 1025.

Un programma che voglia leggere un dato dalla tastiera dovrebbe quindi eseguire un codice tipo:

```
1       mov r0, #1024   @ indirizzo base del dispositivo
2       mov r1, #1      @ costante 1
3       str r1, [r0]    @ ordina la lettura di un tasto
4 wait: ldr r1, [r0]    @ controlla termina lettura
5       cmp r1, #1
6       beq wait        @ e in caso attendi
7       ldr r0, [r0, #1]@ carica codice del tasto letto
8       mov pc, lr      @ ritorno al chiamante
```

> **ATTENZIONE**: questo modo di interagire con l'ingresso uscita è assolutamente inefficace: vengono eseguite istruzioni a vuoto (ciclo "wait") in attesa che l'utente schiacci un tasto. Un ritardo di un paio di secondi da parte dell'utente a schiacciare il tasto dopo che è stata fatta partire la routine di lettura implica, su una macchina con un clock da 1GHz, l'esecuzione di qualcosa come 2sec/3nsec istruzioni in attesa che venga letto il carattere! (vedi Sez. 10.3)

Ma come facciamo a fare in modo che le load e le store del codice abbiano effetto sul dispositivo e non siano interpretate come normali load e store sul sottosistema di memoria?

Il processore riconosce alcuni indirizzi come appartenenti allo spazio di I/O (è questo il vero e proprio concetto di memory mapped I/O) e, tramite la MMU, redirige le richieste relative a questi indirizzi sul bus che collega processore e dispositivi di I/O (vedi Fig. 10.1). Tutti i dispositivi di I/O vedono passare le operazioni, ma ognuno intercetta solo quelle che fanno riferimento agli indirizzi che gli sono stati assegnati. Per esempio, il nostro dispositivo tastiera controllerà quali sono gli indirizzi indicati nelle load e store che passano sul bus e tratterrà per se, ed eseguirà opportunamente, solo quelli relativi all'indirizzo 1024 e 1025. Lo schema semplificato che implementa questa soluzione è quello riportato nella Fig. e9.1 del libro[1].

[1]Il capitolo 9 del libro, che parla dell'ingresso uscita, può essere scaricato dal sito web dell'editore in forma PDF

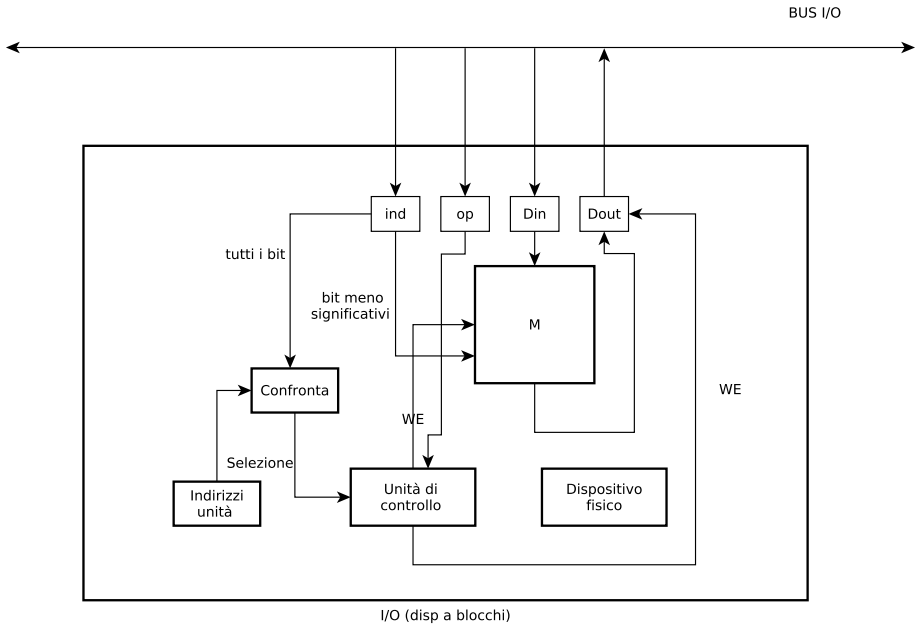


Figura 10.2: Unità di I/O

Lo schema della Fig. e9.1 in realtà fa riferimento a un modello molto semplice in cui l'unità di ingresso uscita accede esclusivamente un registro in ingresso e di conseguenza produce un dato in uscita. Il dispositivo di I/O in oggetto riceverà tutti i dati relativi alle operazioni di lettura scrittura effettuate nello spazio di indirizzamento di ingresso uscita.

Per come lo abbiamo descritto, il memory mapped I/O fa in realtà corrispondere un certo insieme di indirizzi a indirizzi di una memoria interna alla unità di ingresso uscita. Vediamo come questo funziona con un semplice esempio.

Immaginiamo che gli indirizzi dedicati all'I/O siano gli indirizzi che vanno da 1024 a 2048 e che gli indirizzi 1024 a 1025 debbano corrispondere a 2 locazioni di memoria di un certo dispositivo di I/O (la nostra tastiera). Come prima cosa, le operazioni di load e store eseguite dal processore verranno dirottate verso

- il sottosistema di memoria se l'AND dei bit IND[31:10] è 0 oppure l'OR dei bit IND[31:11] è 1
- il sottosistema di I/O nessuna delle due condizioni precedenti è vera.

Quando l'unità che gestisce la tastiera vede passare un indirizzo, controlla che sia compreso nell'intervallo $[1024, 1025]$, ovvero che valga IND[10]=1 e OR{IND[31:11], IND[9:0]}=0 oppure che IND[10]=1, IND[0]=1 e OR{IND[31:11], IND[9:1]}=0. Se questa condizione è vera, utilizza l'ultimo bit come indirizzo della propria memoria interna di due posizioni ed esegue l'operazione richiesta dal processore (LDR o STR). Diversamente ignora l'operazione (la Fig. 10.2 illustra un possibile schema di implementazione della unità che controlla la tastiera interagendo col bus di I/O).

La tecnica del Memory Mapped I/O richiede una fase di negoziazione fra dispositivo e sistema operativo (codice che di fatto si occupa delle interazioni con i dispositivi di ingresso uscita). Gli indirizzi riservati all'I/O sono definiti dal costruttore del processore. Come questi vengano mappati sugli indirizzi riconosciuti dai dispositivi come propri è normalmente oggetto di negoziazione durante l'inizializzazione dei dispositivi "plug-and-play".

E' da notare che gli indirizzi utilizzati per indirizzare i registri dei dispositivi di ingresso uscita potrebbero non essere soggetti al normale processo di traduzione da indirizzi logici a indirizzi fisici nella MMU o che la MMU stessa potrebbe implementare una traduzione dall'indirizzo riservato dal costruttore del processore a quello invece realmente utilizzato nel dispositivo.

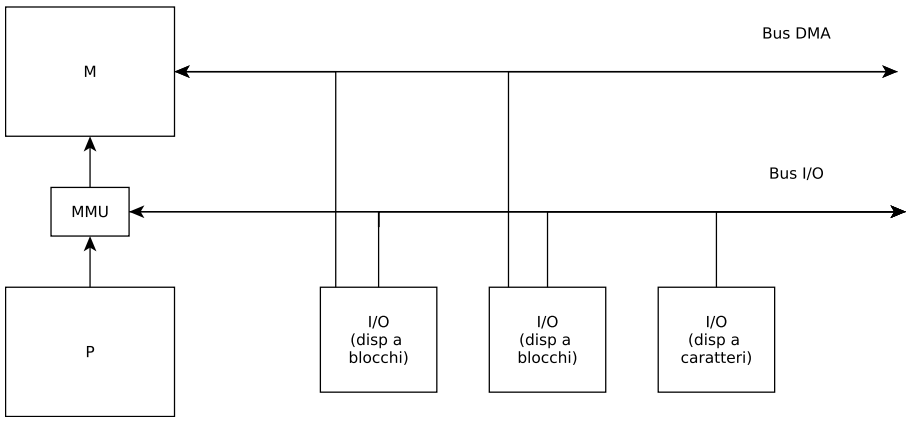


Figura 10.3: Sistema con periferiche in DMA

10.2 DMA

DMA sta per *direct memory access* ed è una tecnica che si usa per implementare i dispositivi di ingresso uscita in modo da permettergli un accesso controllato alla memoria centrale.

I dispositivi di I/O, per come li abbiamo visti nella Sez. 10.1 dovrebbero lavorare esclusivamente sulle loro memorie interne anche quando devono trattare operazioni che coinvolgono moli di dati relativamente grosse. Ad esempio, una unità di controllo di un disco lavora normalmente con operazioni su *blocchi*, ovvero mette a disposizioni operazioni elementari di lettura e scrittura di un blocco del disco, di dimensioni dell'ordine del Kbyte. In questo caso, una ipotetica operazione di lettura richiederebbe che il processore trasferisse il blocco dalla memoria del dispositivo alla memoria centrale mediante una serie di load. Questo è chiaramente un problema, perché può impegnare il processore per un tempo relativamente lungo.

La tecnica del DMA consente ad un dispositivo di accedere direttamente alla memoria centrale del sistema. In pratica (vedi Fig. 10.3):

- il dispositivo è collegato alla memoria centrale mediante un *BUS DMA* e può effettuare letture e scritture in memoria quando ha ottenuto la possibilità di utilizzare il bus;
- il bus DMA è condiviso fra tutte le periferiche in grado di lavorare in DMA;
- il processore passa alle periferiche, quando ordina l'esecuzione di una certa operazione, gli eventuali indirizzi necessari a completare l'operazione direttamente in memoria centrale (questo avviene tramite il bus di I/O);
- la memoria diventa un dispositivo che deve essere in grado di soddisfare le richieste di lettura e scrittura che arrivano da più soggetti (processore, via interfaccia di memoria standard, e periferiche, via bus DMA) e deve pertanto essere dotato di una unità controllo più sofisticata.

Con queste assunzioni, un'operazione di lettura dal disco su un dispositivo che supporta sia Memory Mapped I/O che DMA avviene tramite i seguenti passi:

1. il sistema operativo (programma in esecuzione sul processore), invia alla periferica utilizzando il memory mapped I/O l'ordine di esecuzione della lettura che contiene:
   - la specifica che si tratta di una operazione di *lettura*;
   - l'*indirizzo del blocco* di disco interessato
   - l'*indirizzo in memoria* del buffer da utilizzare per i dati letti da disco;
2. il dispositivo legge i dati dal disco in un proprio buffer interno. Questa operazione deve avvenire in un buffer interno per ragioni di temporizzazione: attendere l'accesso al bus DMA e conseguentemente alla memoria centrale potrebbe risultare troppo lungo (o in ritardo) rispetto ai tempi dettati dal trasferimento dei dati dal dispositivo, una volta arrivati al blocco desiderato;

3. infine, il dispositivo acquisisce, interagendo con l'arbitro del bus, il controllo del bus DMA e avvia il trasferimento del blocco letto nella posizione di memoria il cui indirizzo base è stato passato dal sistema operativo fra i parametri dell'operazione di lettura.

10.2.1 Dispositivi DMA

I dispositivi che normalmente sono dotati di DMA sono quelli cosiddetti *a blocchi*, ovvero quelli che operano su blocchi di dato. Tipicamente, si tratta di dispositivi di memorizzazione di massa (dischi a stato solido e non), di rete (interfacce di rete Ethernet), interfacce video, etc. Quelli che non sono dotati di DMA sono invece quelli cosiddetti *a caratteri* che comprendono dispositivi come tastiere, mouse, tavolette grafiche, etc.

10.3 Interruzioni

Un'interruzione è un evento asincrono[2] rispetto all'esecuzione del programma sul processore. Nel caso più generale, le interruzioni o eccezioni vengono utilizzate per scopi diversi:

- per gestire le operazioni di ingresso uscita; tenendo conto del fatto che il completamento di una operazione di I/O richiede tempi molto lunghi rispetto alla scala di tempi richiesta dal processore per eseguire le istruzioni assembler, si utilizzano le interruzioni per segnalare il completamento di una operazione di I/O, ordinata dal processore utilizzando memory mapped I/O. In questo modo, il processore è svincolato dalla necessità di attendere esplicitamente il completamento delle operazioni di ingresso uscita e può utilizzare il tempo speso dall'unità per completare l'operazione richiesta per svolgere altri compiti;
- per gestire situazioni di errore conseguenti ad azioni specifiche eseguite dal programma (fetch di una istruzione il cui codice non è riconosciuto come uno dei codici delle istruzioni assembler implementate dal processore, fault di pagina, divisione per 0).
- per eseguire chiamate di sistema, ovvero per chiamare parti di codice assembler con diritti diversi da quelli normalmente assegnati agli utenti (in generale con minimi livelli di privilegio);

Le interruzioni sono generate da dispositivi esterni al processore, quali dispositivi di I/O, ma anche dal sottosistema di memoria, per esempio in caso la memoria voglia segnalare un fault di pagina. Le eccezioni (interruzioni sincrone) sono invece quelle generate direttamente dal processore in caso di errori di vario genere.

Le interruzioni vengono *sentite* dal processore alla fine di ognuna delle iterazioni del ciclo *fetch-decode-execute*, come già accennato nelle parti precedenti del corso. Il processore, come tutte le unità firmware, implementa un ciclo simile al seguente:

```
while(true) {
  IR = fetch(PC);
  decode(IR);
  execute(IR);
  update(PC);
  if(interrupt) {
    interrupt_management();
  }
}
```

Ad ogni iterazione, in assenza di interruzioni, il processore preleva, decodifica ed esegue una singola istruzione assembler[3].

Quando si verifica un'interruzione, il suo trattamento prevede una sequenza di passi standard:

- l'istruzione in esecuzione viene completata e lo stato del processore viene aggiornato di conseguenza (per esempio, viene calcolato il PC corrispondente alla prossima istruzione da eseguire e, se necessario, viene effettuato il writeback dei risultati nel file dei registri o nella memoria dati);

[2] in caso di interruzioni legate ad errori che si possono verificare nel processore, sono in realtà eventi sincroni, diretta conseguenza di azioni intraprese durante l'esecuzione di istruzioni assembler da parte del processore. In questo caso sarebbe più corretto parlare di "eccezioni". Nel caso di architetture ARM il termine eccezione viene spesso utilizzato per tutti i casi, ovvero per le interruzioni (asincrone) e per le eccezioni vere e proprio (sincrone).

[3] per non complicare la spiegazione stiamo assumendo di lavorare con un processore single o multi-cycle, non pipeline

Mode	Privileged	Purpose
User	No	Normal operating mode for most programs (tasks)
Fast Interrupt (FIQ)	Yes	Used to handle a high-priority (fast) interrupt
Interrupt (IRQ)	Yes	Used to handle a low-priority (normal) interrupt
Supervisor	Yes	Used when the processor is reset, and to handle the software interrupt instruction `swi`
Abort	Yes	Used to handle memory access violations
Undefined	Yes	Used to handle undefined or unimplemented instructions
System	Yes	Uses the same registers as User mode

Figura 10.4: Stati del processore ARM

- si salva parte dello stato del processore (tipicamente almeno PC, LR e SP) e si procede ad eseguire una parte di codice assembler che
  - dipende dal tipo di interruzione che si è verificata, e
  - contiene l'intero codice necessario a gestire quel tipo di interruzione

  L'esecuzione del codice di trattamento avviene in uno *stato* particolare, a seconda del tipo di interruzione, in generale con privilegi diversi (maggiori) rispetto ai privilegi disponibili in nello stato "user" (stato in cui un normale utente del processore esegue i propri programmi);
- al termine, si ripristina lo stato del processore e, al prossimo ciclo **while(true)** si procede ad eseguire la "prossima" istruzione del programma che era stato interrotto, quella corrispondente al PC calcolato quando ci siano accorti dell'interruzione, prima di saltare alla routine di trattamento.

Se volessimo essere più precisi, il ciclo fetch-decode-execute presentato nel listato precedente andrebbe presentato in modo leggermente diverso, ovvero:

```
while(true) {
  try {
    IR = fetch(PC);
    decode(IR);
    execute(IR);
    update(PC);
  } catch (exception e) {
    exception_management();
  }
  if(interrupt) {
    interrupt_management();
  }
}
```

Il listato evidenzia come, in presenza di eccezioni, venga immediatamente eseguito un handler. Se si è verificato un errore durante la execute (o la fetch), viene immediatamente invocato l'handler dell'eccezione, senza di fatto aggiornare il PC. Quando l'handler risolve il problema ritorna all'esecuzione del codice originale ripetendo l'istruzione che ha causato l'eccezione, visto che il PC non è stato ancora aggiornato.[4] ARM riconosce un certo numero di stati diversi, riassunti nella Fig. 10.4. Gli stati Fast Interrupt e Interrupt sono gli stati che vengono utilizzati per il trattamento delle interruzioni generate dai dispositivi di ingresso uscita. Gli stati Abort e Undefined sono utilizzati per trattare le eccezioni generate da accessi illegali in memoria (per fetch o ldr/str) e per quelle generate da tentativo di esecuzione di una istruzione (assembler) illegale. Lo stato Supervisor viene utilizzato per l'esecuzione delle chiamate di sistema (istruzioni SVC), che sono anche loro gestite come "interruzioni software".

Ciascuno di questi stati dispone di alcuni registri duplicati. La Fig. 10.5 indica quali registri sono duplicati nei vari "modi" (stati) dei processori ARM. Vengono sempre messi a disposizione, nei vari modi che servono il trattamento

[4]Questo meccanismo permette, per esempio, il trattamento del fault di pagina. La MMU genera un'eccezione, l'handler provoca l'esecuzione della parte di sistema operativo che si occupa del page fault. Quando il trattamento del page fault da parte del sistema operativo è terminato, il programma che aveva generato il fault viene fatto ripartire e riparte esattamente dal punto in cui si era fermato, cioè ordinando lo stesso accesso in memoria che aveva causato il fault. Questa volta l'accesso avverrà senza problemi visto che il sistema operativo ha appena effettuato i passi necessari a portare in memoria centrale la pagina mancante.

User	System	Fast Interrupt	Interrupt	Supervisor	Abort	Undefined
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	**R8_fiq**	R8	R8	R8	R8
R9	R9	**R9_fiq**	R9	R9	R9	R9
R10	R10	**R10_fiq**	R10	R10	R10	R10
R11	R11	**R11_fiq**	R11	R11	R11	R11
R12	R12	**R12_fiq**	R12	R12	R12	R12
R13 (SP)	R13 (SP)	**R13_fiq**	**R13_irq**	**R13_svc**	**R13_abt**	**R13_und**
R14 (LR)	R14 (LR)	**R14_fiq**	**R14_irq**	**R14_svc**	**R14_abt**	**R14_und**
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)
Program Status Registers						
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_fiq	**SPSR_irq**	**SPSR_svc**	**SPSR_abt**	**SPSR_und**

Figura 10.5: Registri duplicati (per modo operativo) in ARM

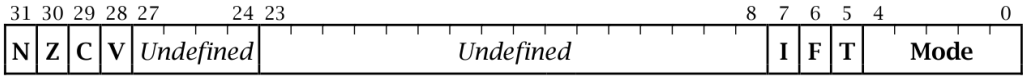


Figura 10.6: Contenuto del registro CPSR

delle interruzioni, registri duplicati per LR e SP. Questo significa che non occorre preoccuparsi di salvare, quando si passa ad uno di questi modi operativi, nessuno dei due registri. Questi registri vengono ripristinati quando si ritorna dal trattamento dell'interruzione che provoca il cambiamento di stato. Inoltre, tutti i modi mettono a disposizione un registro SPSR (Saved Program Status Register) che mantiene lo stato del CPSR al momento del passaggio di stato in modo che tale registro possa essere ripristinato al rientro dal trattamento dell'interruzione. Il modo Fast Interrupt mette a disposizione una copia anche dei registri general purpose da R8 a R12. Questo modo è stato previsto per quelle interruzioni che richiedono un trattamento molto breve e che necessita di pochi registri general purpose. Nel caso ne bastino 5, possiamo utilizzare durante il trattamento di una fast interrupt i registri R8–R12 senza bisogno di salvarli sullo stack: nel modo fast interrupt infatti usiamo le copie (grige nella figura) e non i registri originali.

In una gran parte dei casi, il codice eseguito per trattare un'interruzione è eseguito *ad interruzioni disabilitate*, ovvero facendo in modo che ulteriori interruzioni non possano essere prese in considerazione se non dopo che il trattamento dell'interruzione corrente sia effettivamente terminato.

Questo effetto si ottiene, in ARM, intervenendo sul registro CPSR (Current Program Status Register) che contiene due bit (I ed F) che, se messi a 1, rispettivamente mascherano (fanno in modo che non vengono trattate) le interruzioni normali e quelle "fast" (vedi oltre). Ci sono anche una serie di casi in cui un'interruzione può essere a sua volta interrotta da una interruzione il cui trattamento sia più urgente e semplicemente possa essere completato in modo molto veloce senza disturbare il trattamento dell'interruzione corrente. In questo caso occorre includere nel codice per il trattamento delle interruzioni "interrompibili" tecniche di programmazione che assicurano che non vengono persi (sovrascritti) dati in caso di interruzioni di interruzioni[5].

Le interruzioni vengono segnalate al processore utilizzando appositi pin del processore che portano all'unità controllo 1 o più bit che vengono utilizzati, quando sono messi a 1, per segnalare tipi diversi di interruzioni. Arm per esempio, ha a disposizione, fra gli altri, due segnali diversi per segnalare un'interruzione "normale"

[5]un po' come quando si programmano routing ricorsive e si utilizza lo stack invece che i registri per il passaggio dei parametri per fare in modo che siano supportate le chiamate ricorsive alla funzione

0	reset
4	undef instruction
8	sw interrupt (SVC)
12	abort (fecth da indirizzo illegale)
16	abort (ldr/str indirizzo illegale)
20	reserved
24	IRQ
28	FIQ

Figura 10.7: Tipo di interruzioni ARM

(bit IRQ) oppure un'interruzione "fast" (bit FIQ). Le interruzioni di tipo fast vengono utilizzate per segnalare eventi che possono essere trattati molto rapidamente. Quelle di tipo normale richiedono tempi di trattamento più lunghi. Normalmente, a ciascuno dei due tipi di interruzioni possono corrispondere interruzioni generate da dispositivi diversi. Per esempio, dischi e schede di rete genereranno entrambe un'interruzione IRQ. Hardware dedicato provvederà a presentare al processore un unico segnale (da un bit) IRQ risultato dell'OR di tutti gli IRQ delle diverse unità di controllo delle periferiche e messo in AND con la negazione del bit `I` del registro CPSR. Il motivo dell'interruzione è di norma posto in un indirizzo particolare di memoria, noto alla routine che gestisce le interruzioni, che lo può quindi controllare per capire quale dei dispositivi ha veramente messo IRQ a 1.

In particolare, quando si verifica un'interruzione il processore, una volta completata l'esecuzione dell'istruzione corrente ed aggiornato lo stato del processore esegue una serie ben precisa di passi[6]:

- salva il valore del program counter corrente (è l'indirizzo di ritorno dall'interruzione, di fatto) nella copia del registro LR disponibile in ognuno dei modi operativi
- salva il CPSR nel SPSR
- setta il modo (stato) del processore nella CPSR (vedi Fig. 10.8). Per accedere le diverse parti del registro di stato vengono utilizzate due istruzioni particolari: la MRS (move status to register) che permette di copiare il contenuto del CPSR in un registro generale (per esempio, `MRS R0, CPSR` lo copia nel registro generale R0) in modo da poter successivamente manipolarne il contenuto (testare o settare/azzerare particolari bit o configurazioni di bit) e l'istruzione duale MSR (move register to status) che scrive un registro generale nel CPSR[7] (per esempio, `MRS CPSR, R0` copia il contenuto di R0 nel CPSR). In certi casi, sono disponibili modi per indicare flag particolari del CPSR come destinazioni delle MRS. Per esempio, `MSR CPSR_F, #1` setta il bit di mascheramento delle fast interrupt.
- setta il bit T della CPSR a 0 in modo che il processore esegua istruzioni normali anziché Thumb
- a seconda dei casi, disabilita interruzioni normali e/o fast settando i bit F e I del registro CPSR
- salta ad eseguire il codice che si trova all'indirizzo `0x0000000` + tipo dell'interruzione, dove il tipo dell'interruzione è definito come in Tab. 10.7.

Dualmente, il ritorno dal trattamento interruzioni esegue le seguenti azioni:

- sposta il contenuto del registro LR in PC[8];
- copia il registro SPSR in CPSR, cosa che ripristina il modo di funzionamento precedente all'interruzione, e
- azzera i bit di mascheramento delle istruzioni, in caso fossero stati settati a 1 (questi sono i bit I e F della parola di stato in CPSR).

[6] l'elenco che segue è sempre riferito all'architettura ARM, architetture differenti potrebbero eseguire azioni leggermente diverse o utilizzare indirizzi diversi

[7] versioni diverse del processore hanno modalità leggermente diverse di utilizzo di queste due istruzioni, per esempio nel modo utilizzato per nominare i registri di stato (CPSR, APSR, ...). Queste istruzioni potrebbero essere disponibili solo in certi "modi", a seconda della classe di processori ARM utilizzati

[8] in alcuni casi dopo averci sommato un piccolo offset

Mode Bits		Processor Mode (Abbreviation)	Accessible Registers
Bin	Hex		
10000	10	User (usr)	PC, R14-R0, CPSR
10001	11	Fast Interrupt (fiq)	PC, R14_fiq-R8_fiq, R7-R0, CPSR, SPSR_fiq
10010	12	Interrupt (irq)	PC, R14_irq, R13_irq, R12-R0, CPSR, SPSR_irq
10011	13	Supervisor (svc)	PC, R14_svc, R13_svc, R12-R0, CPSR, SPSR_svc
10111	17	Abort (abt)	PC, R14_abt, R13_abt, R12-R0, CPSR, SPSR_abt
11011	1B	Undefined (und)	PC, R14_und, R13_und, R12-R0, CPSR, SPSR_und
11111	1F	System (sys)	PC, R14-R0, CPSR

Figura 10.8: Modi ARM (rappresentazione nel CPSR)

Vediamo, per chiarire i concetti, come avviene il trattamento di una sw interrupt (il tipo di interruzione (eccezione, in verità) causato dall'esecuzione di una istruzione SVC 0).

L'istruzione SVC è rappresentata in memoria da 8 bit con il codice corrispondente all'istruzione e 24 bit utilizzati per rappresentare il parametro (imm24):

```
[COND|1111|     24 bit immediate   ]
```

L'esecuzione della `SVC` comporta i passi che abbiamo dettagliato poco sopra, al termine dei quali si salta ad eseguire l'istruzione all'indirizzo 0x00000008. Questa istruzione dovrebbe saltare immediatamente un pezzetto di codice simile al seguente:

```
1  LDR R1, [LR, -4]          ; carica in R1 la svc
2  BIC R1, #0xff000000       ; cancella 8 bit piu' significativa
3  LDR R0, =svcvec           ; base vettore con indirizzi svc[i]
4  LDR PC, [R0, R1, LSL #2]  ; salto alla routine svc[i]
```

Parte IV

Parallelismo

Capitolo 11

Forme di parallelismo

Queste note sono un riassunto molto conciso delle cose che servono per comprendere le forme di parallelismo utilizzate nei processori che abbiamo visto nell'ultima parte del corso. Prima prendiamo in considerazione quali sono le *misure* di interesse per il parallelismo. Poi facciamo vedere alcuni esempi di parallelismo nel processore visti da questo punto di vista un po' più generale di quello sbrigativamente adottato nel libro di testo.

11.1 Computazioni sequenziali e parallele

Introduciamo con alcuni piccoli esempi il concetto di attività parallela in contrapposizione al concetto di attività sequenziale.

Immaginiamo di aver bisogno di tradurre un libro dall'inglese all'italiano. Una singola persona può cominciare a tradurre il libro dalla prima pagina e proseguire fino all'ultima pagina. La singola persona che traduce è in questo caso il nostro "processore" e la traduzione avviene sequenzialmente, una pagina dopo l'altra. Assumendo che mediamente la persona impieghi un certo tempo t_p per tradurre una singola pagina e che il libro contenga m pagine, il tempo necessario per la traduzione sarà $m \times t_p$.

Immaginiamo adesso di avere a disposizione n persone in grado di tradurre testo dall'inglese all'italiano, tutte in grado di tradurre una singola pagina in un tempo pari al t_p di prima. Quello che potremmo fare è assegnare ad ognuno degli n traduttori un n-esimo delle pagine da tradurre. Dunque dividiamo il libro in n parti da $\frac{m}{n}$ pagine ed assegnamo ognuna delle parti ad uno dei traduttori. Tutti i traduttori lavoreranno per un tempo pari a $\frac{m}{n} \times t_p$ ed al termine potremmo riunire le diverse parti del libro tradotto a formare il risultato finale "intero libro tradotto".

In questo caso abbiamo utilizzato un insieme di n "processori" che hanno lavorato in parallelo su compiti completamente indipendenti fra di loro. Il risultato netto è stata una chiara diminuzione del tempo necessario per tradurre il nostro libro. Se assumiamo di aver speso un certo t_{split} per dividere il libro in parti uguali ed assegnarle ai traduttori e un certo tempo t_{merge} per ricomporre il libro tradotto, potremmo dire che la traduzione del libro ha richiesto un tempo pari a

$$t_{split} + \frac{m \times t_p}{n} + t_{merge}$$

a fronte di un tempo "sequenziale" pari al valore precedentemente calcolato come

$$m \times t_p$$

Quanto abbiamo guadagnato in questo caso, in termini di tempo necessario per la traduzione? Siamo andati

$$\frac{m \times t_p}{t_{split} + \frac{m \times t_p}{n} + t_{merge}}$$

volte più veloci (rapporto fra il tempo impiegato per tradurre il libro da soli e quello impiegato per tradurre il libro con n traduttori *in parallelo*). Se i tempi per dividere il libro in parti e per ricomporre il risultato finale fossero

trascurabili rispetto al tempo necessario per tradurre $\frac{m}{n}$ pagine, potremmo dire che il guadagno è stato

$$\frac{m \times t_p}{\frac{m \times t_p}{n}} \equiv n$$

ovvero è direttamente proporzonale al numero di traduttori ("processori") utilizzati.

Consideriamo un altro esempio, completamente diverso: costruire una serie di oggetti (m oggetti in tutto) ciascuno dei quali richiede per costruzione una sequenza di operazioni $f_1, \ldots, f_k$ ciascuna delle quali richiede un tempo t_i per essere completata.

Se facessi il lavoro da solo, impiegherei un tempo pari a

$$\sum_{i=1}^{k} t_i$$

tempo per costruire un oggetto e di conseguenza un tempo pari a

$$m \times \sum_{i=1}^{k} t_i$$

per costruirne m.

Se fossimo k costruttori potremmo però organizzarci a "catena di montaggio": il primo costruttore potrebbe compiere l'operazione f_1 e passare il risultato dell'azione al secondo costuttore. Questo potrebbe eseguire l'operazione f_2 e passare il risultato al terzo costruttore, e così via. Alla fine della catena, l'ultimo costruttore riceverebbe il pezzo già lavorato con le operazion $f_1, \ldots, f_{k-1}$ svolgerebbe l'operazione f_k e produrrebbe finalmente il primo oggetto. Tuttavia, mentre il costruttore i esegue l'operazione f_i su un pezzo appena ricevuto dal costruttore $i-1$ quest'ultimo potrebbe eseguire l'operazione f_{i-1} sul pezzo successivo e il costruttore $i+1$ potrebbe eseguire l'operazione f_{i+1} sul pezzo precedente della seguenza di oggetti in costruzione.

Questo modo di procedere fa avanzare i pezzi da un costruttore all'altro con un ritmo dettato dal costruttore più lento. Nell'ipotesi che tutte le operazioni richiedano lo stesso tempo t_o per essere eseguite, questo significa che ogni t_o ciascun costruttore passa al costruttore successivo il pezzo appena lavorato e comincia ad eseguire l'operazione f_i sul pezzo successivo. Se invece ognuna delle operazioni richiedesse un tempo t_i diverso, dovremmo considerare il ritmo di avanzamento degli oggetti come $max\{t_1, \ldots, t_k\}$, visto che l'operazione più lenta bloccherebbe lo scorrimento finchè non venisse completata indipendentemente dal fatto che le altre operazioni potrebbero essere già concluse.

In queste condizioni, il tempo necessario per produrre m oggetti potrebbe essere visto come somma di due tempi:

- il tempo necessario al primo oggetto ad uscire dalla catena di montaggio (riempimento della catena: dopo questo tempo tutti i costruttori lavorano su un pezzo diverso)
- il tempo necessario all'ultimo costruttore a completare gli altri $m-1$ pezzi (ovvero $(m-1)t_o$ oppure $(m-1) \times max\{t_1, \ldots, t_k\}$ a seconda dei due casi).

Con queste considerazioni il tempo per produrre m pezzi sarebbe quindi

$$\sum_{i=1}^{k} t_i + (m-1) \times max\{t_1, \ldots, t_k\}$$

Nell'ipotesi che $m >> k$ (cioè che dobbiamo produrre molti più pezzi del numero dei costruttori) possiamo trascurare sia la sommatoria iniziale che il -1 nel moltiplicatore del massimo e dire che il tempo per produrre gli m oggetti è circa

$$m \times max\{t_1, \ldots, t_k\}$$

Il guadagno rispetto al tempo impiegato da un singolo costruttore sarebbe dunque di

$$\frac{m \times \sum_{i=1}^{k} t_i}{m \times max\{t_1, \ldots, t_k\}} \cong \frac{m \times k \times t_o}{m \times t_o} = k$$

volte.

Questi due esempi mostrano due istanze di parallelismo "spaziale" e "temporale", secondo la terminologia adottata nel libro di testo. Nel primo caso, si tratta di parallelismo spaziale perchè l'elaborazione avviene su dati diversi in "luoghi" diversi. Nel secondo caso è parallelismo "temporale" perchè in qualche modo l'elaborazione di un dato avviene in "luoghi" diversi in tempi diversi.

11.2 Misure

In questa sezione, definiamo formalmente un certo numero di misure che serviranno per valutare gli effetti dell'introduzione di forme di parallelismo.

- **Latenza**
  La latenza misura il tempo fra l'inizio di un calcolo e la produzione del relativo risultato. La latenza si indica normalmente con L.
- **Tempo di completamento**
  Misura relativa all'esecuzione di un certo numero m di calcoli (normalmente calcolo di una stessa funzione f) su un insieme di dati in ingresso $x_1, \ldots, x_m$. Il tempo di completamento (T_C) rappresenta l'intervallo fra il tempo in cui inizia il primo calcolo e quello in cui termina l'ultimo calcolo.
- **Tempo di servizio**
  Misura relativa all'esecuzione di un certo numero m di calcoli (normalmente calcolo di una stessa funzione f) su un insieme di dati in ingresso $x_1, \ldots, x_m$. Il tempo di servizio (T_S) è il tempo che intercorre fra la produzione di due risultati consecutivi o fra l'inizio di due calcoli consecutivi.
- **Throughput**
  E' l'inverso del tempo di servizio, ovvero il numero di calcoli completati per unità di tempo. Spesso viene detto anche "banda di elaborazione" (B).

Dalla definizione segue che diminuire la latenza significa rendere eseguire una singola computazione più velocemente, mentre diminure il tempo di servizio (aumentare il throughput) significa produrre più risultati nella stessa quantità di tempo.

11.3 Misure derivate

Utilizzando le misure (primitive) appena definite possiamo definire alcune misure, tutte definite in funzione del grado di parallelismo n:

- **Speedup**
  E' rapporto fra il miglior tempo sequenziale e il tempo utilizzato in parallelo con grado di parallelismo pari a n:
  $$\text{Speedup}(n) = \frac{T_{seq}}{T_{par}(n)}$$
- **Scalabilità**
  E' il rapporto fra il tempo impiegato a calcolare con grado di parallelismo 1 e quello impiegato con grado di parallelismo n:
  $$\text{Scalab}(n) = \frac{T_{par}(1)}{T_{par}(n)}$$
- **Tempo ideale**
  E' il rapporto fra il tempo sequenziale e il grado di parallelismo n:
  $$T_{id} = \frac{T_{seq}}{n}$$

- **Efficienza**
  E' il rapporto fra il tempo ideale e quello ottenuto con grado di parallelismo n:

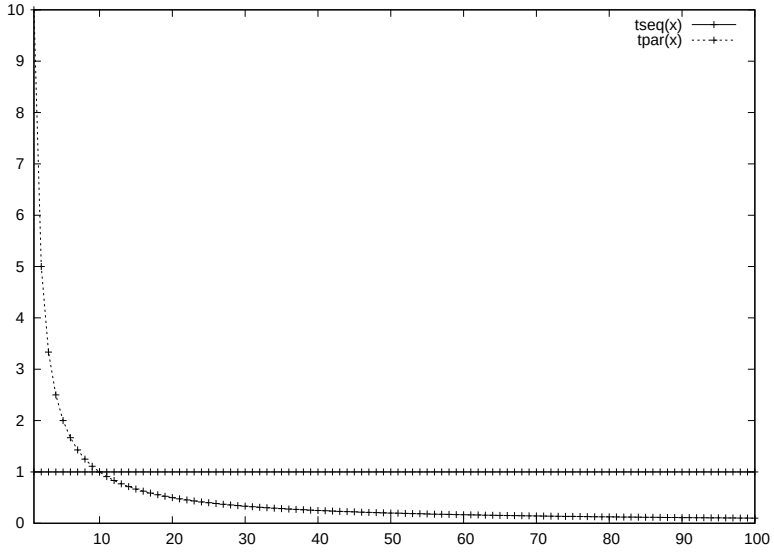
$$\text{eff}(n) = \frac{T_{id}}{T_{par}(n)} = \frac{\frac{T_{seq}}{n}}{T_{par}(n)} = \frac{T_{seq} n}{n \times T_{par}(n)} = \frac{\text{speedup}(n)}{n}$$

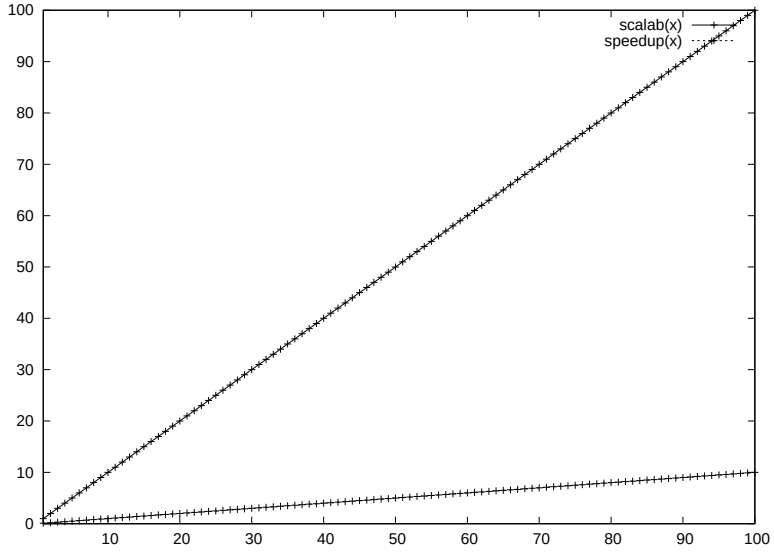
Lo speedup misura di quanto miglioriamo il tempo dell'esecuzione sequenziale ed è normalmente un valore che è compreso fra la retta $y = x$ (bisettrice del primo quadrante) e l'asse delle ascisse, ovvero vale che

$$\text{speedup}(n) \leq n$$

La scalabilità misura invece quanto, data una soluzione parallela, questa soluzione riesce ad adattarsi a gradi di parallelismo diversi. Dal momento che ci confrontiamo con il caso "soluzione parallela con grado di parallelismo 1" sostanzialmente misuriamo quanto aumentando il grado di parallelismo aumenti anche il guadagno. Questo però non dice nulla relativamente allo speedup. Potremmo avere una soluzione sequenziale che calcola il risultato in $T_{seq} < T_{par}(1)$ e quindi ci potremmo trovare nella condizione che il tempo parallelo diventa migliore del tempo sequenziale solo dopo un certo grado di paralleliesmo, pur in presenza una scalabilità lineare.

La situazione è riassunta dai due grafici che seguono, che rappresentano una situazione abbastanza tipica e non ideale: il primo rappresenta il tempo impiegato per calcolare quanto dobbiamo calcolare in funzione del grado di parallelismo (asse delle ascisse). La retta rappresenta il tempo sequenziale. La curva rappresenta il tempo parallelo. Il secondo rappresenta la scalabilità (retta più alta) e lo speedup (retta più bassa). Notate che la scalabilità è ideale (per grado di parallelismo n abbiamo una scalabilità di n mentre per lo speedup abbiamo uno speedup di $\frac{n}{10}$ per grado di parallelismo n.





L'efficienza ci dice quanto riusciamo a sfruttare le risorse a disposizione ed è normalmente un valore che è compreso fra la retta $y = 1$ e l'asse delle ascisse, ovvero vale che

$$\text{eff}(n) \leq 1$$

Nel caso dei grafici precedenti, l'efficienza sarebbe sempre pari a 0.1, ovvero molto bassa rispetto all'ideale, cioè 1.

11.4 Forme di parallelismo

Descriviamo in maniera un po' più formale le forme di parallelismo che utilizzeremo poi per la realizzazione del processore.

11.4.1 Pipeline

Un pipeline è costituito da un certo numero di *stadi* $S_1, \ldots, S_k$ ognuno dei quali calcola una certa funzione (assumiamo che S_i calcoli f_i. L'uscita dello stadio i è ingresso dello stadio $i+1$. Il primo stadio riceve dall'esterno un certo numero di task $x_1, \ldots, x_m$. I task arrivano in istanti distinti, tipicamente ogni T_a. L'ultimo stadio produce un numero pari a m di valori in uscita, diretti all'esterno del pipeline. Il valore $j-$esimo y_j è il risultato del calcolo di $f_m(f_{m-1}(\ldots f_2(f_1(x_j))\ldots))$.

Per il pipeline con k stadi che processa uno stream di m elementi, detti rispettivamente L_i e T_i la latenza e il tempo di servizio dello stadio i, sotto condizione che $T_a < max\{T_1, \ldots, T_k\}$ abbiamo che:

$$T_S(k) = max\{T_1, \ldots, T_k\}$$

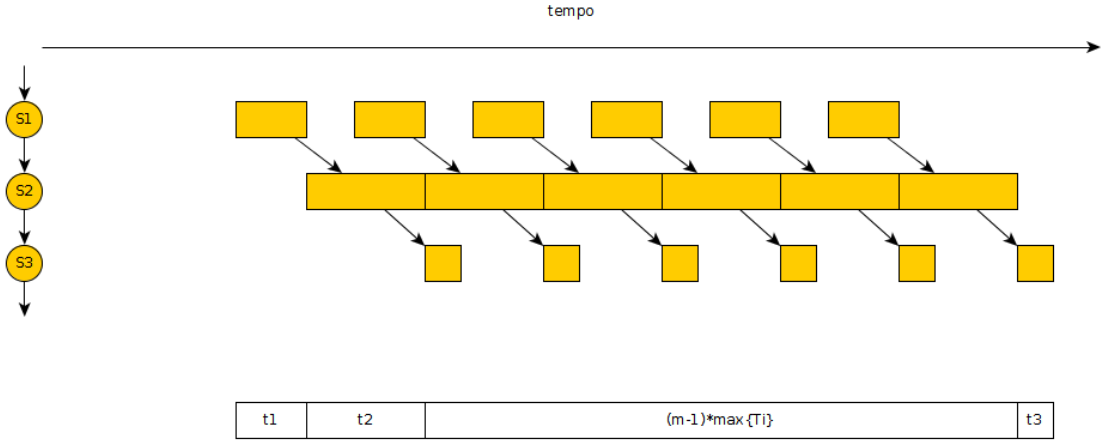
$$T_C(k, m) = \sum_{i=1}^{k} T_i + (m-1) * T_S$$

e il massimo speedup che possiamo ottenere rispetto al caso sequenziale è pari a k, numero degli stadi. In realtà, senza alcuna assunzione su T_a vale che

$$T_S = max\{T_a, T_1, \ldots, T_k\}$$

dal momento che indipendentemente dalla velocità di esecuzione dei task nel pipeline non possiamo andare più veloci del tempo impiegato dai task per arrivare al pipeline stesso.

Il grafico che segue fa vedere come vengono calcolati diversi task nel tempo (tempo che va da sinistra versa destra) nei tre stadi (dall'alto verso il basso). La parte in basso fa vedere il tempo di completamento come sommatoria dei tempi dei vari stadi (due all'inizio e uno alla fine, visto che lo stadio più lungo è quello intermedio) più numero dei task meno uno volte il tempo dello stadio più lento.



Rispetto alla terminologia del libro, il pipeline esprime parallelismo "temporale".

Un pipeline permette di aumentare il throughput ma non di diminuire la latenza di una computazione sequenziale.

Farm

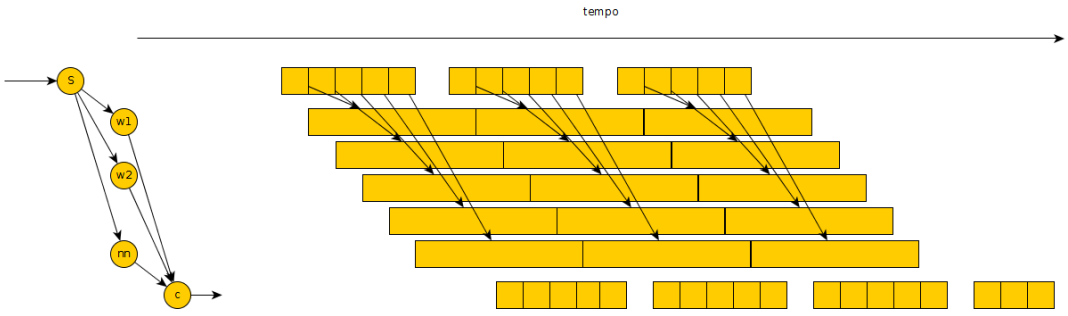
Un farm è costituito da un certo numero n_w di worker, ognuno dei quali riceve in ingresso un dato x, calcola la stessa funzione f e produce in uscita un risultato $f(x)$. Il farm riceve dall'esterno un certo numero di task $x_1, \ldots, x_m$ e produce come risultati un numero pari a m di valori in uscita, diretti all'esterno del farm, di valore $f(x_1), \ldots, f_{x_m}$. Come per il pipeline, i task in ingresso arrivano in istanti di tempo diversi, ogni T_a.

Per il farm con n_w worker che processa m task in ingresso che arrivano ogni T_a, assumento di impiegare T_{sched} per assegnare un task in ingresso ad un worker, T_{coll} per raccogliere un valore calcolato da un worker e spedirlo in uscita al farm e T_w per calcolare una singola $f(x)$ in uno dei worker, vale che:

$$T_S(n_w) = max\{T_a, T_{sched}, T_{coll}, \frac{T_w}{n_W}\}$$

$$T_C(n_w, m) = T_{sched} + \frac{m}{n_W} T_w + T_{coll} \cong m \times T_S$$

Il grafico che segue fa vedere come si svolge la computazione di un certo numero di task nel tempo (da sinistra verso destra) sui vari worker (dall'alto verso il basso).



Il massimo speedup ottenibile con un farm è pari a n_w. Come il pipeline, il farm permette di aumentare il throughput ma non di diminuire la latenza di una computazione seqeunziale.

Map

Una map processa un certo insieme di dati $\{x_1, \ldots, x_m\}$ utilizzando un certo numero n_w di worker. Ciascuno dei worker è in grado di calcolare una funzione f su un qualunque x_i producendo un valore $f(x_i)$. La map riceve l'insieme in input, impiega un certo tempo T_{split} per dividere l'insieme in n_w sottoinsiemi e assegnarli ai worker, i worker calcolano il loro sottoinsieme ed infine la map impiega un certo tempo T_{merge} per ricostruire il risultato finale a partire dai risultati parziali prodotti dai worker, ovvero a partire dall'insieme degli $f(x_i)$.

Per una map con n_W worker che processa un insieme di m elementi computado su ciascun elemento una funzione f che impiega un tempo T_f vale che:

$$L(n_w) = T_C(n_w, 1) = T_{split} + \frac{T_{seq}}{n_W} * T_{coll} = T_{split} + \frac{m * T_f}{n_W} * T_{coll}$$

Il massimo speedup rispetto alla computazione sequenziale è n_W. Qualora si utilizzi la map per calcolare uno stream di insiemi in input, allora il tempo di servizio andrebbe calcolato come:

$$T_s(n_w) = L(n_w)$$

La map dunque permette di dimunire la latenza e di incrementare in throughput di una computazione sequenziale.

La map rapprensenta un pattern di parallelismo "spaziale" secondo la notazione del libro di testo.

11.5 Composizione di forme di parallelismo

Pipeline e farm lavorano su *stream* (sequenze i cui valori esistono in istanti diversi) di dati e sono dette forme di parallelismo stream (stream parallelism). La map lavora su un singolo insieme di dati, scomponendolo in sottoinsiemi e calcolando il risultato finale come funzione dei risultati calcolati in parallelo sui sottoinsiemi ed è detta forma di parallelismo sui dati (data parallelism).

Forme di parallelismo su stream possono a loro volta avere worker o stadi paralleli di tipo:

- sequenziale
- stream parallel
- data parallel

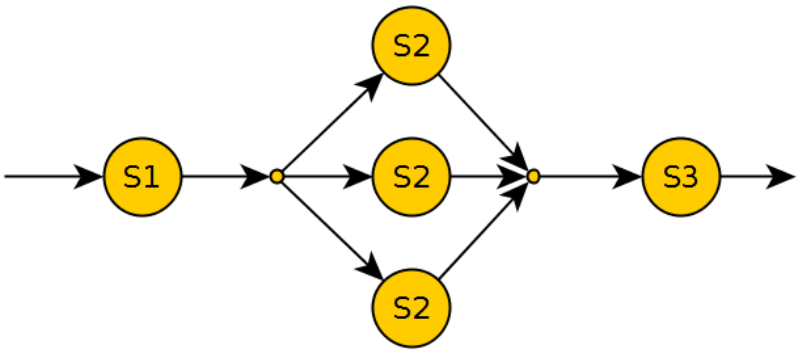
Forme di parallelismo sui dati possono a loro volta avere worker paralleli di tipo:

- sequenziali
- data parallel

ma non stream parallel.

I modelli di prestazioni (T_S e T_C si compongono di conseguenza.

Per esempio, supponiamo di avere un pipeline con tre stadi: il primo e l'ultimo sequenziali con $L1 = t_1$ e $L_3 = t_3$ e il secondo di tipo farm, con n_W worker e $T_w = t_2$.



Per calcolare il tempo di servizio del pipeline:

$$T_{S_{pipe}}(3) = max\{t_1, T_{S_{farm}}, t_3\} \tag{11.1}$$

dal momento che per gli stadi sequenziali vali chiaramente che $T_S = L$. Considerando il tempo di servizio del farm:

$$T_S(n_W) = max\{T_{sched}, \frac{T_w}{n_W}, T_{coll}\}$$

e sostituendolo nella Eq. 11.1:

$$T_{S_{pipe}}(3) = max\{t_1, max\{T_{sched}, \frac{T_w}{n_W}, T_{coll}\}, t_3\}$$

ovvero

$$T_{S_{pipe}}(3) = max\{t_1, T_{sched}, \frac{T_w}{n_W}, T_{coll}, t_3\}$$

11.5.1 Ottimizzazione delle computazioni parallele

Per migliorare la performance di una computazione parallela:

- si individua la misura di interesse (e.g. T_S o T_C)
- si individuano eventuali colli di bottiglia (e.g. stadi lenti in un pipeline)
- si cerca di individuare una migliore decomposizione parallela che rimuova il collo di bottiglia (e.g. si trasforma lo stadio lento del pipeline sequenziale in un farm)

Per esempio, consideriamo un pipeline di tre stadi, ognuno dei quali sia inzialmente sequenziale. Ognuno degli stadi calcola sequenzialmente una `map`. Una rappresentazione dell'algoritmo sequenziale in pseudo codice c-like potrebbe essere la seguente:

```
1  /* stadio 1 */
2  for(i=0; i<N; i++)
3    a[i] = f(b[i]);
4  /* stadio 2 */
5  for(i=0; i<N; i++)
6    c[i] = a[i]+b[i];
7  /* stadio 3 */
8  for(i=0; i<N; i++)
9    d[i] = g(c[i]);
```

Supponiamo che il tempo impiegato per calcolare le funzioni `f` e `g` siano rispettivamente t_f e t_g e il tempo necessario a calcolare la somma di due elementi dei vettori (nel secondo stadio) sia t_+ Il tempo necessario a calcolare questi tre stadi su un singolo elemento in input (vettore `b` di `N` elementi) sarà

$$N \times t_f + N \times t_+ + N \times t_g = N(t_f + t_+ + t_g)$$

Il tempo necessario per calcolare sequenzialmente uno stream di m vettori b sarà quindi

$$mN(t_f + t_+ + t_g)$$

Supponiamo che $t_f << t_g$ e che t_f sia dello stesso ordine di grandezza di t_+. Chiediamoci quanto tempo spendiamo calcolando i nostri risultati utilizzando un pipeline di tre stadi sequenziali. Il tempo sarà approssimato da

$$m \times max\{t_f, t_+, t_g\} \cong m \times t_g$$

Chiaramente è limitato dal fatto che il terzo stadio sia quello più lento. Sfruttando il fatto che sappiamo che potenzialmente tutti e tre gli stadi sono `map` potremmo pensare di parallelizzare il terzo stadio come `map` utilizzando tanti worker quanti necessari per far diventare il tempo di calcolo simile a quello degli altri stadi, ovvero

$$n_W \cong \frac{t_g}{t_f}$$

Questo è chiaramente possibile se e solo se $\frac{t_g}{t_f}$ rimane maggiore del t_{split} e del t_{merge} spesi per distribuire sottoinsiemi del dato in ingresso ai worker a per collezionare i risultati parziale e ricostruire il risultato finale, diversamente il tempo della map rimarrebbe limitato a $max\{t_{split}, t_{merge}\}$.

Nella migliore delle ipotesi, settando il grado di parallelismo $n_w = \lceil \frac{t_g}{t_f} \rceil$ il tempo di completamento scenderebbe a

$$m \times max\{t_f, t_+, \frac{t_g}{\lceil \frac{t_g}{t_f} \rceil}\} \cong m \times t_f$$

Avremmo anche potuto fare di più. In effetti avremmo implementare delle map in tutti e tre gli stadi. In teoria questo avrebbe permesso di far scendere il tempo di calcolo di tutti e tre gli stadi a $max\{t_{split}, t_{merge}\}$ e quindi avrebbe potuto far scendere il tempo necessario a calcolare m task a

$$m \times max\{t_{split}, t_{merge}\}$$

Va comunque tenuto presente che in questo caso siamo partiti da una computazione che di fatto era un `pipeline` di `map` momentaneamente implementate come sequenziali. Non sempre è così. Se gli stadi non avessero potuto essere trasformati in `map`, avremmo potuto comunque migliorare le prestazioni del pipeline in termini di throughput invece che di tempo di completamento. Se avessimo avuto comunque un massimo nel tempo di calcolo del terzo stadio e questo non potesse essere parallelizzato con una `map`, avremmo potuto trasformare lo stadio in un `farm`. Questo avrebbe ridotto il tempo di servizio del pipeline, riducendo il tempo di servizio del terzo stadio e, di conseguenza, anche il tempo di completamento, pur non riducendo il tempo necessario per il calcolo di un singolo task.

11.6 Esempi

In questa sezione discutiamo alcuni esempi di utilizzo del parallelismo nella microarchitettura. Alcune di queste cose le trovate anche nel libro di testo, nella parte che descrive le caratteristiche avanzate dei processori, ma le ripresentiamo con la terminologia discussa in queste note.

11.6.1 Processore pipeline

Il processore pipeline visto nel capitolo della microarchitettura sfrutta il parallelismo pipeline. Gli stadi si occupano delle varie fasi relative all'esecuzione di una istruzione:

- **fecth**: prelievo dell'istruzione (accesso alla IM all'indirizzo PC)
- **decode**: accesso al register file e/o estrazione delle costanti dalla parola dell'istruzione
- **esecuzione**: utilizzo della ALU per calcolo dei risultati e/o degli eventuali indirizzi per gli accessi in memoria data

- **memoria**: accesso eventuale alla memoria dati per load e store
- **write back**: scrittura nel register file (o nel PC, in caso di salti) dei risultati dell'esecuzione dell'istruzione

Il fatto che la forma di parallelismo si pipeline ha diverse implicazioni:

- il tempo di servizio è dato dal massimo dei tempi di servizio degli stadi (leggi: la lunghezza del ciclo di clock va dimensionata sullo stadio più lento)
- il tempo di completamento per lunghe sequenze di calcoli (ovvero di istruzioni) diventa proporzionale al tempo di servizio dello stadio più lento (m istruzioni impiegano circa $m\tau$ per essere calcolate, con τ lunghezza del ciclo di clock, ovvero tempo di servizio dello stadio più lento)
- se avessimo uno stadio molto lento potremmo cercare di limitarne le consequenze replicandolo (ovvero trasformandolo in un `farm`). La cosa si può applicare ai soli stadi senza stato (per esempio la ALU) e ha degli impatti sulla performance in caso di dipendenze logiche.
- per abbassare il tempo di servizio potremmo spezzare lo stadio lento in più sotto stadi. Questo si puà fare in alcuni casi, non sempre e ha comunque degli impatti sulla performance in caso di dipendenze logiche, come nel caso precedente.

11.6.2 Unità vettoriale

Quando ci troviamo nella condizione di dover eseguire molte istruzioni su dati diversi di una struttura dati tipo vettore, possiamo pensare di implementare una map:

- carichiamo più dati in un registro possibilmente senza spendere più tempi di accesso in memoria (e.g. utilizzando memorie interallaciate e collegamenti di ampiezza maggiore della singola parola), e
- utilizziamo più ALU in parallelo per operare sulle diversi porzioni dei registri

Di fatto stiamo utilizzando una `map` e quindi:

- riduciamo il tempo (latenza) necessario per calcolare operazioni su tutti gli elementi del vettore di un fattore che può arrivare al massimo a $\frac{N}{n_{alu}}$ (con N numero di elementi nel vettore e n_{alu} numero delle ALU nell'unità vettoriale

Le istruzioni vettoriali (estensioni SSE, AVX (mondo Intel) o NEON (mondo ARM) dell'insieme di istruzioni scalari) permettono di calcolare operazioni fra registri vettoriali visti come vettori di registri scalari. In pratica, permettono di eseguire molto velocemente operazioni tipo

$$\forall i : a[i] = b[i]\ OP\ c[i]$$

con OP operazione aritmetica tipo ADD, SUB, MUL ... o logica tipo AND, OR, ...