Architettura degli Elaboratori—Corso A—9 gennaio 2020

Riportare su tutti i fogli consegnati in alto a destra Nome, Cognome, Matricola.
I risultati saranno resi pubblici via WEB appena disponibili assieme al calendario degli orali.

Domanda 1

Si fornisca il codice il codice di un componente (*module* o *primitive*) Verilog che calcola la funzione "maggiore o uguale" di due numeri da 2 bit. Il risultato deve essere restituito tramite l'unico parametro *output* del componente. Il componente deve essere implementato o utilizzando una tabella di verità (in questo caso sarà un componente di tipo *primitive*) o utilizzando espressioni dell'algebra booleana (in questo caso sarà un *module*). **Non** è permesso utilizzare operatori di confronto e implementazioni di tipo behavioural.
Suggerimento: potete utilizzare una mappa di Karnaugh per calcolare quanto va incluso nel *module* o nella *primitive*.

Domanda 2

Si fornisca il codice assembler ARM derivato dalla compilazione dello pseudocodice:

```
for(k=0;k<N;k++) {
  U[k,k] = A[k,k];
  for(int i=k+1; i<N; i++) {
    L[i,k]=A[i,k]/U[k,k];
    U[k,i] = A[k,i];
  }
}
```

assumendo che:

- le matrici siano di interi e tutti i calcoli siano su interi
- le matrici siano memorizzate per riga (prima riga a partire dall'indirizzo base, seconda riga a partire da indirizzo base + N, etc.)
- gli indirizzi base delle matrici U, L ed A siano contenuti rispettivamente nei registri R4, R5 ed R6 e che il registro R7 contenga invece il valore N
- la divisione fra interi (per la quale non è prevista nell'assembler ARM una specifica istruzione) sia calcolata chiamando la funzione **fdiv**, con il dividendo in R0 e il divisore in R1. Si assuma anche che la funzione restituisca in R0 il risultato e in R1 il resto della divisione.

Domanda 3

Si discutano sinteticamente le cause di degrado delle prestazioni in un processore ARM pipeline come quello descritto nel libro di testo. Successivamente, si consideri il codice relativo alla compilazione del ciclo più interno nel codice assembler prodotto per rispondere alla Domanda 2 e si indichino quali parti di tale codice causano degrado delle prestazioni.

Traccia di soluzione

Domanda 1

La mappa di Karnaugh per gli ingressi X e Y di due bit ciascuno (x1,x2 e y1,y2 con x1 e y1 bit più significativi) è la seguente:

	y1 y2			
x1 x2	00	01	11	10
00	1	0	0	0
01	1	1	0	0
11	1	1	1	1
10	1	1	0	1

Il primo termine è dato dalla prima colonna (~y1 & ~y2), un secondo termine copre la terza riga (x1 & x2), due termini coprono il quadrato sulle prime due colonne rispettivamente seconda e terza riga (x2 & ~y1) e terza e quarta riga (x1 & ~y1) ed infine o il termine x1 & ~x2 & ~y2 (o equivalentemente il termine x1 & y1 & ~y2) copre l'ultimo 1 in basso a destra. Dunque, l'espressione booleana che definisce la funzione è

XMaggioreDiY = (~y1 & ~y2) | (x1 & x2) | (x2 & ~y1) | (x1 & ~y1) | (x1 & y1 & ~y2)

Quindi il componente *primitive* potrebbe essere scritto come

```
primitive magg(output XMaggioreDiY,
               input x1, input x2,
               input y1, input y2);
   table
      // codifica degli implicanti a 1
      ? ? 0 0 : 1;
      1 1 ? ? : 1;
      ? 1 0 ? : 1;
      1 ? 0 ? : 1;
      1 ? 1 ? : 1;
      // codifica degli implicanti a zero
      0 0 ? 1 : 0;
      0 ? 1 1 : 0;
      0 ? 1 0 : 0;
      ? 0 1 1 : 0;
   endtable

endprimitive
```

Naturalmente, potrebbe essere anche scritto fornendo tutte le righe rappresentate nella mappa di Karnaugh:

```
primitive magg(output XMaggioreDiY, input [1:2]x, input [1:2]y);
  table
     0 0 0 0 : 1;
     0 1 0 0 : 1;
     1 1 0 0 : 1;
```

```
    1 0 0 0 : 1;
    0 1 0 1 : 1;
    1 1 0 1 : 1;
    1 1 1 1 : 1;
    1 1 1 0 : 1;
    1 0 0 1 : 1;
    1 0 1 0 : 1;
    0 0 0 1 : 0;
    0 0 1 0 : 0;
    0 0 1 1 : 0;
    0 0 1 0 : 0;
    0 1 1 1 : 0;
    0 1 1 0 : 0;
    1 0 1 1 : 0;
  endtable
endprimitive
```

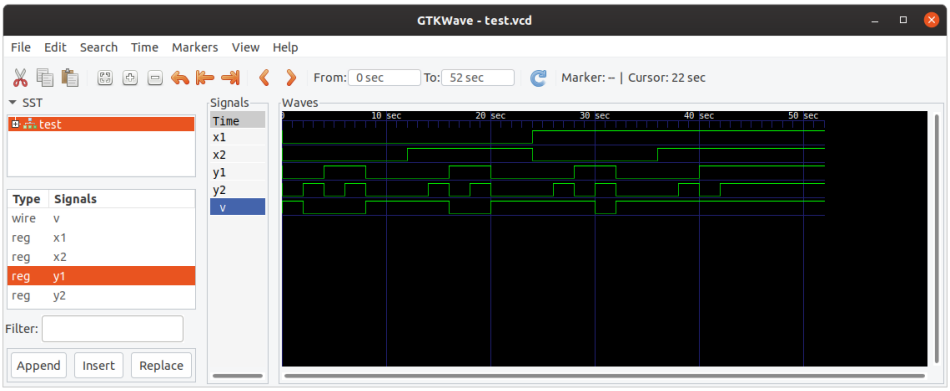
Il modulo implementato mediante espressioni dell'algebra booleana avrebbe invece potuto essere scritto come segue:

```
module magg(output XMaggioreDiY,
            input x1, input x2,
            input y1, input y2);

  assign XMaggioreDiY =
    (~y1 & ~y2) | (x1 & x2) | (x2 & ~y1)
  | (x1 & ~y1) | (x1 & y1 & ~y2);

endmodule
```

Utilizzando un programma di test che esegue due cicli annidati per x e y otteniamo i risultati attesi con ciascuna delle due implementazioni, come evidenziato nella figura che segue.



Domanda 2

Con le solite convenzioni per l'utilizzo dei registri ARM, immaginiamo di utilizzare i registri come segue:

Valore	Base U	Base L	Base A	N	k	i	Off(k,k)	Off(i,k)	Off(k,i)
RegNo	4	5	6	7	8	9	10	11	12

Inoltre, utilizzeremo i registri r0-r3 per valori temporanei avendo cura di salvarne i valori sullo stack quando chiameremo la funzione **fdiv** che li potrà a sua volta utilizzare senza salvarli per i propri calcoli interni (vedi convenzioni utilizzo registri ARM).

Il codice assembler derivante dalla compilazione dello pseudocodice è di conseguenza il seguente:

```
 1            push {r8-r12,lr}
 2            mov r8, #0                    @ R0 <- k
 3    fork:   cmp r8, r7                    @ controlla che i != N
 4            beq finefork
 5            mul r10, r8, r7               @ N * k -> r10
 6            add r10, r10, r8              @ r10 <- off(k,k)
 7            ldr r0, [r6,r10,lsl #2]       @ a[k,k]
 8            str r0, [r4,r10,lsl #2]       @ u[k,k] = a[k,k]
 9            mov r9, r8                    @ i = k
10            add r9, r9, #1                @ i = k+1
11    fori:   cmp r9, r7
12            beq finefori
13            mul r2, r9, r7                @ i*n
14            add r2, r2, r8                @ r2 <- off(i,k)
15            mov r1, r0                    @ u[k,k]        divisore  (appena calcolato)
16            ldr r0, [r6,r2,lsl #2]        @ a[i,k]        dividendo
17            push {r2}                     @ salvato per dopo
18            bl fdiv                       @ restituisce in r0 il risultato della divisione
19            pop {r2}
20            str r0, [r5, r2,lsl #2]       @ l[i,k] = a[i.k] / u[k,k]
21            mul r3, r8, r7                @ k*n
22            add r3, r3, r9                @ r3 <- off(k,i)
23            ldr r0, [r6,r3,lsl #2]        @ a[k,i]
24            str r0, [r4,r3,lsl #2]        @ u[k,i] = a[k,i]
25            @ fine body fori
26            add r9, r9, #1                @ i++
27            b fori                        @ nuova iterazione for(i
28    finefori:
29            add r8, r8, #1                @ k++
30            b fork                        @ nuova iterazione for(k
31    finefork:
32            pop {r8-r12,lr}
```

Prima di cominciare salviamo tutti i registri non temporanei (diversi da r0-r3) sullo stack (riga 1). Per calcolare l'indirizzo delle posizioni [k,k], [k,i] e [i,k] moltiplichiamo l'indice di riga per N e gli sommiamo l'indice di colonna (righe 5,6 per [k,k], 13,14 per [i,k] e 21,22 per [k,i]). Prima di chiamare la fdiv (alla riga 18) salviamo il valore dell'offset [i,k] (riga 17) perché servirà successivamente e perché si trova in un registro temporaneo che potrebbe essere sovrascritto senza problemi dalla fdiv. Il contenuto del registro viene ripristinato immediatamente dopo la chiamata della funzione (riga 19). Notare che quando si salvano i registri con la prima push salviamo anche il link register che verrà utilizzato (e dunque sovrascritto) per chiamare la fdiv.

Domanda 3

Il processore pipeline dovrebbe essere in grado, a regime, di completare una istruzione ogni singolo ciclo di clock. Tuttavia, possiamo avere dipendenze di due tipi nel codice, che fanno si che alcune istruzioni richiedano un numero di cicli di clock maggiore per essere completate:

- Dipendenze sul controllo. Sono quelle legate al calcolo del target di una istruzione di salto. Occorre attendere il calcolo del nuovo valore del PC da parte dello stadio EXEC (quello con la ALU) e questo comporta (utilizzando comunque tecniche di forwarding del risultato della ALU) uno stallo di due cicli di clock.
- Dipendenze sui dati. Sono quelle che si verificano quando l'istruzione i+k-esima deve leggere un registro che deve essere ancora scritto dall'istruzione i-esima per via del modello di esecuzione a pipeline. Le dipendenze fra istruzioni operative sono risolte utilizzando il forwarding. Quelle generate da una istruzione di LDR richiedono uno stallo di 1 ciclo di clock dal momento che il valore da leggere è disponibile solo dopo la lettura nella memoria dati, che non può essere eliminata completamente utilizzando tecniche di forwarding.

Il codice relativo al ciclo più interno è il seguente:

```
11    fori:   cmp r9, r7
12            beq finefori
13            mul r2, r9, r7                @ i*n
14            add r2, r2, r8                @ r2 <- off(i,k)
15            mov r1, r0                    @ u[k,k]        divisore  (appena calcolato)
16            ldr r0, [r6,r2,lsl #2]        @ a[i,k]        dividendo
17            push {r2}                     @ salvato per dopo
18            bl fdiv                       @ restituisce in r0 il risultato della divisione
19            pop {r2}
20            str r0, [r5, r2,lsl #2]       @ l[i,k] = a[i.k] / u[k,k]
21            mul r3, r8, r7                @ k*n
22            add r3, r3, r9                @ r3 <- off(k,i)
23            ldr r0, [r6,r3,lsl #2]        @ a[k,i]
24            str r0, [r4,r3,lsl #2]        @ u[k,i] = a[k,i]
25            @ fine body fori
26            add r9, r9, #1                @ i++
27            b fori                        @ nuova iterazione for(i
```

Sono evidenziate le istruzioni di salto e l'unica istruzione di load seguita da una istruzione che legge il registro appena caricato. Per quanto detto avremmo una bolla da 1 ciclo di clock nell'esecuzione della coppia di istruzioni 23-24 e una bolla da due cicli di clock per l'esecuzione dei salti alla linea 27 e 12. Per eseguire 16 istruzioni si impiegheranno quindi 22 cicli di clock.

Si noti che la dipendenza indotta dalla LDR può essere eliminata. STR non legge il registro r9, che è letto e scritto dalla add alla linea 26. Pertanto, le due istruzioni possono essere eseguite in qualunque ordine. Possiamo spostare la add prima della str:

```
23            ldr r0, [r6,r3,lsl #2]        @ a[k,i]
24            add r9, r9, #1                @ i++
25            str r0, [r4,r3,lsl #2]        @ u[k,i] = a[k,i]
```

Questo fa' sì che la lettura di r0 nella str possa avvenire senza ritardi grazie al forwarding.