COMPITO di ARCHITETTURA DEI CALCOLATORI del 14-11-2019 **MATRICOLA**_____

DA RESTITUIRE INSIEME AGLI ELABORATI e A TUTTI I FOGLI
→ NON USARE FOGLI NON TIMBRATI
→ ANDARE IN BAGNO PRIMA DELL'INIZIO DELLA PROVA
→ NO FOGLI PERSONALI, NO TELEFONI, SMARTPHONE/WATCH, ETC

COGNOME_____

NOME_____

NOTA: per l'esercizio 4 dovranno essere consegnati DUE files: il file del programma VERILOG e il file relativo all'output (screenshot o copy/paste)

1) [12/30] Trovare il codice assembly MIPS corrispondente dei seguenti micro-benchmark (**utilizzando solo e unicamente istruzioni dalla tabella sottostante**), rispettando le convenzioni di utilizzazione dei registri dell'assembly MIPS (riportate qua sotto, per riferimento).

```
<BENCHMARK-1:>
<la funzione e' invocata con fib_it(10)>

int fib_it(int n) {
   int tmp, first = 0, second = 1;

   while (n--) {
      tmp = first+second;
      first = second;
      second = tmp;
   }
   return first;
}
```

```
<BENCHMARK-2:>
<la funzione e' invocata con fib_rc(10)>

int fib_rc(int n) {
   if (n == 0) return (0);
   if (n == 1) return (1);
   else return (fib_rc(n-1)+(fib_rc(n-2));
}
```

Successivamente, calcolare il tempo di esecuzione di ciascuno dei due benchmark ipotizzando che vengano eseguiti su un processore con frequenza di clock pari a 1 GHz, assumendo i seguenti valori per il CPI di ciascuna categoria di istruzioni: aritmetico-logiche-salti 1, branch 3, load-store 10.

Instructions

Opcode+Funct (hexadecimal)	Instruction	Example	Meaning	Comments
00+20/00+21	**add**	`add/addu $1,$2,$3`	$1 = $2 + $3	(signed/unsigned) 3 operands; exception possible
00+22/00+23	**subtract**	`sub/subu $1,$2,$3`	$1 = $2 - $3	(signed/unsigned) 3 operands; exception possible
08/09	**add immediate**	`addi/addiu $1,$2,100`	$1 = $2 + 100	(signed/unsigned) + constant ; exception possible
00+18/00+19	**multiplication**	`mult/multu $1, $2`	Hi,Lo= $1 x $2	(signed/unsigned) 64-bit Product ; result in Hi,Lo
00+1A/00+1B	**division**	`div/divu $1, $2`	Hi= $1 % $2, Lo = $1 / $2	(signed/unsigned) division
00+10/00+12	**move from Hi / move from Lo**	`mfhi/mflo $1`	$1 = Hi ($1 = Lo)	Create copy of Hi (Create a copy of Lo)
00+2A/00+2B	**set on less than**	`slt/sltu $1,$2,$3`	if ($2 < $3) $1 = 1; else $1 = 0	(signed/unsigned) compare $2 and $3 (less than)
0A/0B	**set on less than immediate**	`slti/sltiu $1,$2,100`	if ($2 < 100) $1 = 1; else $1 = 0	(signed/unsigned) compare $2 and constant (less than)
00+24/25/26/27	**and / or / xor / nor**	`and/or/xor/nor $1,$2,$3`	$1=$2&$3 / $2|$3 / $2^$3 / !($2|$3)	3 register operands; Logical AND/OR/XOR/NOR
0C/0D/0E	**and /or / xor immediate**	`andi/ori/xori $1,$2,100`	$1 = $2 & 100 / $2 | 100 / $2 ^100	Logical AND/OR/XOR register, constant
00+00	**shift left logical**	`sll $1,$2,10`	$1 = $2 << 10	Shift left by constant
00+02/00+03	**shift right (l=logical,a=arithmetic)**	`srl/sra $1,$2,10`	$1 = $2 >> 10	Shift right by constant (for arithmetic: sign is preserved)
00+04	**shift left logical**	`sllv $1,$2,10`	$1 = $2 << $3	Shift left by variable
00+06/00+07	**shift right (l=logical,a=arithmetic)**	`srlv/srav $1,$2,10`	$1 = $2 >> $3	Shift right by variable (for arithmetic: sign is preserved)
23/20	**load word / load byte**	`lw/lb $1,100($2)`	$1 = Memory[$2+100]	Data from memory to register
24	**load byte unsigned**	`lbu $1,100($2)`	$1 = Memory[$2+100]	Data from mem. To reg.; no sign extension
2B/28	**store word / store byte**	`sw/sb $1,100($2)`	Memory[$2+100] = $1	Data from register to memory
0F	**load upper immediate**	`lui $1,0x1234`	$1=0x1234'0000	load most significant 16 bits
PSEUDOINSTRUCTION	**load address**	`la $1,var`	$1 = &var	Load address of var (lui $1,H16(&var);ori $1, L16(&var)) H16/L16=high/low 16 bits of &var
02	**jump**	`j 10000`	go to 10000	Jump to target address
00+08	**jump register**	`jr $31`	go to $31	For switch, procedure return
03	**jump and link**	`jal 10000`	$31 = PC + 4;go to 10000	For procedure call
04	**branch on equal**	`beq $1,$2,100`	if ($1 = = $2) go to PC+4+100	Equal test; PC relative branch
05	**branch on not equal**	`bne $1,$2,100`	if ($1 != $2) go to PC+4+100	Not equal test; PC relative
00+0C	**syscall**	`syscall`	call OS service $v0	See table of system calls below
10+10,rs=10	**rfe**	`rfe`	shift right (k,e) bits in STATUS reg	Exit Kernel Mode, Enable Interrupts
PSEUDOINSTRUCTION	**branch unconditional**	`b 100`	go to PC+4+100	PC relative branch (e.g., beq $0,$0,100)
PSEUDOINSTRUCTION	**no operation**	`nop`	do nothing	Do nothing (e.g. sll $0,$0,0)
30	**load-linked**	`ll $1,100($2)`	$1=Memory[$2+100]	Read and start to monitor the given memory location
38	**store-conditional**	`sc $1,100($2)`	Memory[$2+100]=$1 or →	return 0 if a coherence action happens since the previous ll ($1 must be different from 0)
11+00 fmt=10/11	**add.s / add.d**	`add.x $f0,$f2,$f4`	$f0=$f2+$f4	Single and double precision add
11+01 fmt=10/11	**sub.s / sub.d**	`sub.x $f0,$f2,$f4`	$f0=$f2-$f4	Single and double precision subtraction
11+02 fmt=10/11	**mul.s / mul.d**	`mul.x $f0,$f2,$f4`	$f0=$f2*$f4	Single and double precision multiplication
11+03 fmt=10/11	**div.s / div.d**	`div.x $f0,$f2,$f4`	$f0=$f2/$f4	Single and double precision division
11+05 fmt=10/11	**abs.s / abs.d**	`abs.x $f0,$f2`	$f0=ABS($f2)	Single and double precision absolute value
11+06 fmt=10/11	**mov.s / mov.d**	`mov.x $f0,$f2`	$f0←$f2	Single and double precision move
11+07 fmt=10/11	**neg.s / neg.d**	`neg.x $f0,$f2`	$f0= – ($f2)	Single and double precision opposite value
11+3C(31,32,3D,3E,3F) fmt=10/11	**c.lt.s / c.lt.d (ne,eq,gt,le,ge)**	`c.lt.x $f0,$f2`	Temp=($f0<$f2)	Single and double: compare $f0 and $f2 <,=,!=,>,<=>=
11+00 fmt=4/0	**move to/from coprocessor 1**	`mtc1/mfc1 $1,$f2`	$f2=$1 / $1=$f2	Move $1 to/from C1reg. $f2 (no conversion)
10+00 fmt=4/0	**move to/from coprocessor 0**	`mtc0/mfc0 $1,$2`	$c2=$1 / $1=$c2	Move $1 to/from C0 reg. $f2 (no conversion)
11+00 fmt=6/2	**move to/from control reg of cop.1**	`ctc1/cfc1 $1,$cf2`	$cf2=$1 / $1=$cf2	Move $1 to/from C1-CONTROL register
11 fmt=8,ft=1/0	**branch on true/false**	`bc1t/bc1f label`	If (Temp = = true/false) go to label	Temp is 'Condition-Code'
31/39	**load/store floating point (32bit)**	`lwc1/swc1 $f0,0($1)`	$f0←Memory[$1] / Memory[$1]←$f0	Data from FP (C1) register to memory
11+21,fmt=10/11+22,fmt=11	**convert from/to single to/from double**	`cvt.d.s/cvt s.d $f0,$f2`	$f0=(double)$f2/$f0=(single)$f2	Type conversion
11+24,fmt=11/11+20	**convert from/to single to/from integer**	`cvt.w.s/cvt s.w $f1,$f0`	$f1=(int)$f0 / $f0=(single)$f1	Type conversion

Register Usage

Name	Reg. Num.	Usage
$zero	0	The constant value 0
$s0-$s7	16-23	Saved
$t0-$t9	8-15,24-25	Temporaires
$a0-$a3	4-7	Arguments

Name	Reg.Num.	Usage
$v0-$v1	2-3	Results
$fp, $sp	30,29	frame pointer, stack pointer
$ra, $gp	31,28	return address, global pointer
$k0-$k1	26,27	Kernel usage

Reg. Num.	Usage
$f0, $f2	Return values
$f12,$f14	Function arguments
$f20,$f22,$f24,$f26,$f28,$f30	Saved registers
$f4,$f6,$f8,$f10,$f16,$f18	Temporaries registers

System calls

Service Name	Serv.No.($v0)	INPUT Arguments	OUTPUT Args
print_int	1	$a0=integer to print	---
print_float	2	$f12=float to print	---
print_double	3	($f12,$f13)=double to print	---
print_string	4	$a0=address of ASCIIZ string to print	---
read_int	5	---	$v0=integer

Service Name	Serv.No.($v0)	INPUT Arguments	OUTPUT Arguments
read_float	6	---	$f0=float
read_double	7	---	$f0-f1=double
read_string	8	$a0=address of input buffer, $a1=max chars to read	---
sbrk	9	$a0=Number of bytes to be allocated	$v0=pointer to allocated memory
exit	10	---	---

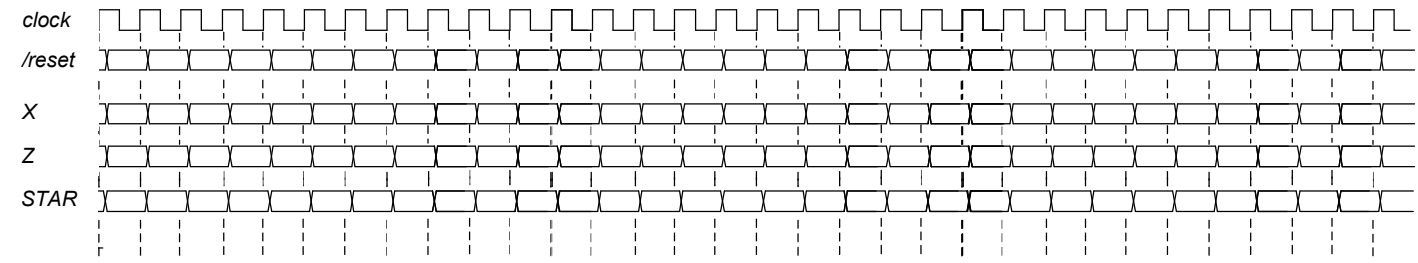
COMPITO di ARCHITETTURA DEI CALCOLATORI del 14-11-2019

2) [5/30] Si consideri una cache di dimensione 128B e a 4 vie di tipo write-back. La dimensione del blocco e' 32 byte, il tempo di accesso alla cache e' 4 ns e la penalita' in caso di miss e' pari a 40 ns, la politica di rimpiazzamento e' LRU. Il processore effettua i seguenti accessi in cache, ad indirizzi al byte: 99, 104, 140, 118, 112, 197, 178, 112, 250, 176, 125, 223, 133, 277, 256, 212, 163, 174, 184. Tali accessi sono alternativamente letture e scritture. Per la sequenza data, ricavare il tempo medio di accesso alla cache, riportare i tag contenuti in cache al termine e la lista dei blocchi (ovvero il loro indirizzo) via via eliminati durante il rimpiazzamento ed inoltre in corrispondenza di quale riferimento il blocco e' eliminato.

3) [5/30]] Disegnare un possibile schema diagramma a stati del comportamento del processore MIPS (FETCH, DECODE, EXECUTE, WRITE-BACK) ed illustrare una possibile implementazione della syscall modificando tale digramma a stati .

4) [8/30] Descrivere e sintetizzare in Verilog una rete sequenziale basata sul modello di Mealy-Ritardato con un ingresso X su un bit e una uscita Z su due bit che funziona nel seguente modo: devono essere riconosciute le sequenze non interallacciate 1,1,0,0 e 0,1,0,1; l'uscita Z[1] va a 1 se si presenta una delle due sequenze mentre Z[0] dice quale sequenza si e' presentata (Z[0]=1 se si presenta 1,1,0,0; Z[0]=0 altrimenti). Rappresentare la macchina a stati finiti per tale rete di Mealy-Ritardato e t**racciare il diagramma di temporizzazione** come verifica della correttezza dell'unità. Nota: si puo' svolgere l'esercizio su carta oppure con ausilio del simulatore salvando una copia dell'output (diagramma temporale) e del programma Verilog su USB-drive del docente.

```
module TopLevel;
  reg reset_;initial begin reset_=0; #22 reset_=1; #300; $stop; end
  reg clock ;initial clock=0; always #5 clock <=(!clock);
  reg X;
  wire [1:0] Z;
  wire [2:0] STAR=Xxx.STAR;
  wire Z1=Xxx.z[1];
  wire Z0=Xxx.z[0];
  initial begin X=0;
  wait(reset_==1); #5
  @(posedge clock); X<=1; @(posedge clock); X<=1; @(posedge clock); X<=0; @(posedge clock); X<=0;
  @(posedge clock); X<=0; @(posedge clock); X<=0; @(posedge clock); X<=0; @(posedge clock); X<=0;
  @(posedge clock); X<=1; @(posedge clock); X<=1; @(posedge clock); X<=0; @(posedge clock); X<=0;
  @(posedge clock); X<=0; @(posedge clock); X<=1; @(posedge clock); X<=0; @(posedge clock); X<=1;
  @(posedge clock); X<=0; @(posedge clock); X<=0; @(posedge clock); X<=0; @(posedge clock); X<=0;
  $finish;
  end
  XXX Xxx(X,Z,clock,reset_);
endmodule
```



COMPITO di ARCHITETTURA DEI CALCOLATORI del 14-11-2019

SOLUZIONE

ESERCIZIO 1

1) Una possibile soluzione per il primo micro-benchmark e' (*si suppone che la funzione fib_it venga richiamata con input 10: fib_it(10)*)::

```
fib_it:
#_____________________________________________________________________________________BB1
       addi $v0, $zero, 0           # first = 0
       addi $v1, $zero, 1           # second = 0

while:
#_____________________________________________________________________________________BB2
       add  $t0, $zero, $a0         # (n != 0) ?
       addi $a0, $a0, -1            # n = n - 1
       beq  $t0, $zero, end_while   # se n==0 allora termino il ciclo

#_____________________________________________________________________________________BB3
       add $t0, $v0, $v1            # tmp = first+second
       add $v0, $zero, $v1          # first = second
       add $v1, $zero, $t0          # second = tmp
       j while

end_while:
#_____________________________________________________________________________________BB4
       jr   $ra                     # passo il controllo al chiamante
```

Il partizionamento del programma in basic block e' fatto prendendo sequenze di una o piu' istruzioni consecutive che o terminano con una istruzione di branch condizionale o una jump incondizionata (ma non una jal), oppure terminano subito prima di un'etichetta di salto. Sono state trascurate tutte quelle parti di codice non richieste espressamente dalla traccia.

	Ni	***ALJ***	***B***	***LS***	***cBi$_{CPU}$***	***Cbi$_{CPU}$= cBi$_{CPU}$*Ni***
BB1	1	2	0	0	2	2
BB2	11	2	1	0	5	55
BB3	10	4	0	0	4	40
BB4	1	1	0	0	1	1
Ccpu (in cicli di clock)						**98**

Si ha quindi che:

$$T_{CPU}^{1} = \frac{C_{CPU}}{f_{CPU}} = \frac{98}{10^{9}} = 98ns$$

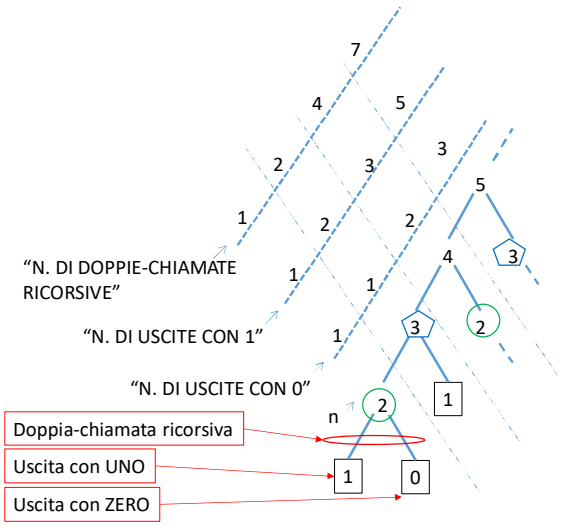
Una possibile soluzione per il secondo micro-benchmark e' (*si suppone che la funzione fib_rc venga richiamata con input 10:fib_rc(10)*):

```
fib_rc:
#_____________________________________________________________________________________BB1
       addi $sp, $sp, -8            # riservo due word nello stack
       sw   $ra, 0($sp)             # salvo l'indirizzo di ritorno nella prima word dello stack
       add  $v0, $zero, $zero       # metto il valore fisso di confronto 0 in t0
       beq  $a0, $v0, exit          # se t1==0 allora ho finito

#_____________________________________________________________________________________BB2
       addi $v0, $zero, 1           # metto il valore fisso di confronto 1 in t0
       beq  $a0, $v0, exit          # se t1==1 allora ho finito

#_____________________________________________________________________________________BB3
       addi $t0, $a0, -2            # t0 = n-2
       addi $a0, $a0, -1            # a0 = n-1
       sw   $t0, 4($sp)             # salva t0 nello stack
       jal  fib_rc                  # chiama fibo(n-1): risultato in v0
       lw   $a0, 4($sp)             # ripristino ex-t0 in a0 (n-2)
       sw   $v0, 4($sp)             # salva v0 nello stack
       jal  fib_rc                  # chiama fibo(n-2): risultato in v0
       lw   $t0, 4($sp)             # ripristina ex-v0 in a0 (fibo(n-1))
       add  $v0, $v0, $t0           # somma fibo(n-2) e fibo(n-1)

exit:
#_____________________________________________________________________________________BB4
       lw   $ra, 0($sp)             # recupero l'indirizzo di ritorno dallo stack
       addi $sp, $sp, 8             # incremento lo stack
       jr   $ra                     # passo il controllo al chiamante
```

COMPITO di ARCHITETTURA DEI CALCOLATORI del 14-11-2019

SOLUZIONE

Osservando la seguente figura, si nota che il numero di chiamate si evolve cosi':



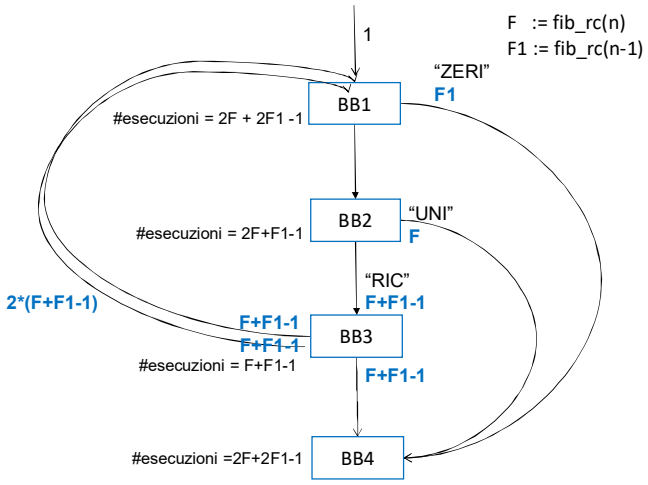
n	Fn
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55

i) Per n=2 (cerchietto verde in basso) →1, per n=3 (pentagono blu in basso) →1, per n=4 sono quelle nel ramo di n=3 + quelle nel ramo di n=2, quindi si ottiene una successione di Fibonacci con un n traslato, infatti per n=4 →2 (ovvero F_3), per n=5 → 3 (ovvero F_4), quindi tali chiamate seguono una successione di Fibonacci, dove **il numero di chiamate "zero" al livello n e' pari a F_{n-1}**. (cf. tabellina della successione di Fibonacci nella stessa figura).

ii) Ragionando in modo analogo, sempre osservando la precedente figura, il numero di chiamate che si risolve restituendo direttamente "uno" si evolve cosi': per n=2 →1, per n=3 →2, per n=4, come nel caso precedente sono quelle nel ramo di n=3 + quelle nel ramo di n=2, quindi si parla sempre di una successione di Fibonacci, ma stavolta per n=2,3,4 ottengo 1,2,3 quindi **il numero di chiamate "uno" e' proprio F_n**.

iii) Infine, se la funzione non restituisce subito "zero" o "uno" significa che effettua la doppia chiamata ricorsiva a fib_rc(n-1) e fib_rc(n-2) e contando dalla stessa figura il numero di tali doppie chiamate si trova: per n=2 →1, per n=3 →2, per n=4 una doppia-chiamata che e' quella del n corrente + le chiamate sul ramo di n=3 + le chiamate sul ramo di n=2, quindi n=4→1+1+2=4, per n=5→1+2+4=7 e cosi' via; quindi, nel caso generale ho 1+a+b dove per n=2,3,4,5,6,... a si evolve secondo 0,0,1,2,4,... e b secondo 0,1,2,4,7,... ovvero a=(F_{n-1}-1) e b=(F_n-1); pertanto **il numero di doppie-chiamate ricorsive (quelle all'interno di BB3) si evolve secondo 1+(F_{n-1}-1) +(F_n-1) = F_n+F_{n-1}-1**.

Sulla base delle precedenti considerazioni possiamo quindi riportare tali valori nel seguente diagramma di flusso che coinvolge i quattro basic-block, sui tre archi etichettati con "ZERI", "UNI" e "RIC" e dedurre quindi il numero di volte in qui transito su ognuno degli archi rimanenti.



Infatti:

i) **BB3 viene invocato F_n + F_{n-1} -1** volte come gia' dedotto e dato che al suo interno chiama due volte nuovamente la funzione fib_rc signifi che l'arco che richiama da BB3 il BB1 e' percorso 2 (F_n+F_{n-1}-1) volte.

ii) **BB1 viene quindi invocato 2 (F_n+F_{n-1}-1)** volte piu' la volta iniziale e quindi 2 F_n + 2F_{n-1} -1 volte.

iii) **BB2 viene invocato** (2 F_n + 2F_{n-1} -1) volte meno il numero di uscite dall'arco "ZERI", ovvero F_{n-1} quindi **2 F_n + F_{n-1} -1 volte**.

iv) **BB4 viene invocato** dall'arco "RIC", dall'arco "ZERI" e dall'arco "UNI" quindi (F_n+F_{n-1}-1) + (F_n) + (F_{n-1}) = 2 **F_n + 2F_{n-1} -1 volte**.

Ricapitolando:

BB1: 2 F_n + 2F_{n-1} -1→ per n=10 vale 177

BB2: 2 F_n + F_{n-1} -1→ per n=10 vale 143

BB3: F_n + F_{n-1} -1→ per n=10 vale 88

BB4: 2 F_n + 2F_{n-1} -1 → per n=10 vale 177

COMPITO di ARCHITETTURA DEI CALCOLATORI del 14-11-2019

SOLUZIONE

quindi:

	Ni	*ALJ*	*B*	*LS*	cBi_{CPU}	$Cbi_{CPU}= cBi_{CPU}*Ni$
BB1	177	2	1	1	15	2655
BB2	143	1	1	0	4	572
BB3	88	5	0	4	45	3960
BB4	177	2	0	1	12	2124
Ccpu (in cicli di clock)						**9311**

$$T^2_{CPU} = \frac{C_{CPU}}{f_{CPU}} = \frac{9311}{10^9} = 9311ns$$

ESERCIZIO 2

Sia X il generico riferimento, A=associativita', B=dimensione del blocco, C=capacita' della cache.
Si ricava S=C/B/A=# di set della cache=128/32/4, XM=X/B, XS=XM%S, XT=XM/S.
A=4, B=32, C=128, RP=LRU, Thit=4, Tpen=40, 19 references:

```
=== T    X   XM  XT  XS  XB  H [SET]:USAGE [SET]:MODIF  [SET]:TAG
=== R   99    3   3   0   3  0 [0]:3,0,0,0 [0]:0,0,0,0  [0]:3,-,-,-
=== W  104    3   3   0   8  1 [0]:3,0,0,0 [0]:1,0,0,0  [0]:3,-,-,-
=== R  140    4   4   0  12  0 [0]:2,3,0,0 [0]:1,0,0,0  [0]:3,4,-,-
=== W  118    3   3   0  22  1 [0]:3,2,0,0 [0]:1,0,0,0  [0]:3,4,-,-
=== R  112    3   3   0  16  1 [0]:3,2,0,0 [0]:1,0,0,0  [0]:3,4,-,-
=== W  197    6   6   0   5  0 [0]:2,1,3,0 [0]:1,0,0,0  [0]:3,4,6,-
=== R  178    5   5   0  18  0 [0]:1,0,2,3 [0]:1,0,0,0  [0]:3,4,6,5
=== W  112    3   3   0  16  1 [0]:3,0,1,2 [0]:1,0,0,0  [0]:3,4,6,5
=== R  250    7   7   0  26  0 [0]:2,3,0,1 [0]:1,0,0,0  [0]:3,7,6,5
=== W  176    5   5   0  16  1 [0]:1,2,0,3 [0]:1,0,0,1  [0]:3,7,6,5
=== R  125    3   3   0  29  1 [0]:3,1,0,2 [0]:1,0,0,1  [0]:3,7,6,5
=== W  223    6   6   0  31  1 [0]:2,0,3,1 [0]:1,0,1,1  [0]:3,7,6,5
=== R  133    4   4   0   5  0 [0]:1,3,2,0 [0]:1,0,1,1  [0]:3,4,6,5
=== W  277    8   8   0  21  0 [0]:0,2,1,3 [0]:1,0,1,0  [0]:3,4,6,8
=== R  256    8   8   0   0  1 [0]:0,2,1,3 [0]:1,0,1,0  [0]:3,4,6,8
=== W  212    6   6   0  20  1 [0]:0,1,3,2 [0]:1,0,1,0  [0]:3,4,6,8
=== R  163    5   5   0   3  0 [0]:3,0,2,1 [0]:0,0,1,0  [0]:5,4,6,8
=== W  174    5   5   0  14  1 [0]:3,0,2,1 [0]:1,0,1,0  [0]:5,4,6,8
=== R  184    5   5   0  24  1 [0]:3,0,2,1 [0]:1,0,1,0  [0]:5,4,6,8
-------------------------------------
P1 Nmiss=8   Nhit=11  Nref=19  mrate=0.421053   AMAT=th+mrate*tpen=20.8421
```

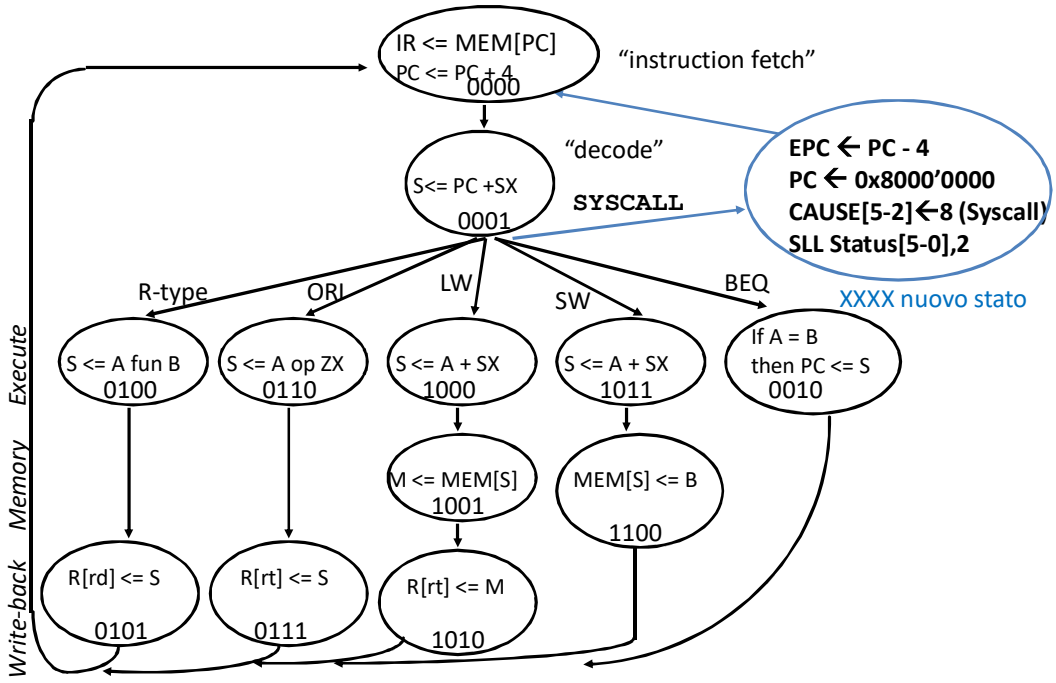
LISTA BLOCCHI USCENTI:

```
(out: XM=4  XT=4  XS=0 )

(out: XM=7  XT=7  XS=0 )
(out: XM=5  XT=5  XS=0 )

(out: XM=3  XT=3  XS=0 )
```

CONTENTUTI dell'unico SET al termine:

ESERCIZIO 3

Nella figura seguente e' rappresentato come potrebbe essere modificato il ciclo di Fetch-Decode-Execute-Write_back (in maniera simile a cio' che avviene per una eccezione) al fine di gestire la syscall.

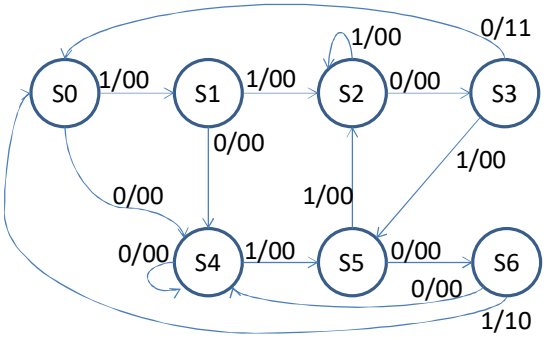


COMPITO di ARCHITETTURA DEI CALCOLATORI del 14-11-2019

SOLUZIONE

ESERCIZIO 4

Diagramma degli stati:



Codice Verilog del modulo da realizzare (possibile soluzione con Mealy-Ritardato):

```
module XXX(x,z,clock,reset_);
input        clock,reset_,x;
output [1:0] z;
reg    [1:0] OUTR;
reg    [2:0] STAR;
parameter S0='B000,S1='B001,S2='B010,S3='B011,
   S4='B100,S5='B101,S6='B110;
always @(reset_==0) #1 begin STAR<=S0; OUTR<=0; end
assign z=OUTR;

always @(posedge clock) if(reset_==1) #3
casex(STAR)
   S0:begin STAR<=(x==0)?S4:S1; OUTR<=0; end
   S1:begin STAR<=(x==0)?S4:S2; OUTR<=0; end
   S2:begin STAR<=(x==0)?S3:S2; OUTR<=0; end
   S3:begin STAR<=(x==0)?S0:S5; OUTR<=(x==0)?'B11:'B00; end
   S4:begin STAR<=(x==0)?S4:S5; OUTR<=0; end
   S5:begin STAR<=(x==0)?S6:S2; OUTR<=0; end
   S6:begin STAR<=(x==0)?S4:S0; OUTR<=(x==1)?'B10:'B00; end
endcase
endmodule
```

Diagramma di Temporizzazione: (template)