COMPITINO #1 di *CALCOLATORI ELETTRONICI 1* del 03-11-09 **MATRICOLA**_____

COGNOME_____

NOME_____

1) [28/40] 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 in calce, 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.

2) [12/40] A) Motivare l'utilizzo dell'istruzione syscall. B) Descrivere in dettaglio tale istruzione, indicando gli elementi architetturali che eventulemente modifica (registri generali, registri speciali o altro). C) Indicare passo-passo cosa avviene *durante* l'esecuzione di tale istruzione ed inoltre le fasi di *preparazione* e *successiva*. D) Discutere una possibile implementazione della syscall modificando il digramma a stati elementare che descrive il comportamento del processore (FETCH, DECODE, EXECUTE, WRITE-BACK).

MIPS instructions

Instruction	Example		Meaning	Comments
add	`add`	`$1,$2,$3`	$1 = $2 + $3	3 operands; exception possible
subtract	`sub`	`$1,$2,$3`	$1 = $2 - $3	3 operands; exception possible
add immediate	`addi`	`$1,$2,100`	$1 = $2 + 100	+ constant; exception possible
subtract immediate	`subi`	`$1,$2,100`	$1 = $2 - 100	- constant; exception possible
multiplication	`mult`	`$1, $2`	Hi,Lo= $1 x $2	64-bit Signed Product ; result in Hi,Lo
division	`div`	`$1, $2`	Hi= $1 % $2, Lo = $1 / $2	Signed division
move from Hi	`mfhi`	`$1`	$1 = Hi	Create copy of Hi
move from Lo	`mflo`	`$1`	$1 = Lo	Create copy of Lo
and	`and`	`$1,$2,$3`	$1 = $2 & $3	3 register operands; Logical AND
or	`or`	`$1,$2,$3`	$1 = $2 | $3	3 register operands; Logical OR
nor	`nor`	`$1,$2,$3`	$1 = !($2 | $3)	3 register operands; Logical NOR
xor	`xor`	`$1,$2,$3`	$1 = $2 ^ $3	3 register operands; Logical XOR
and immediate	`andi`	`$1,$2,100`	$1 = $2 & 100	Logical AND register, constant
or immediate	`ori`	`$1,$2,100`	$1 = $2 | 100	Logical OR register, constant
xor immediate	`xori`	`$1,$2,100`	$1 = $2 ^ 100	Logical XOR register, constant
shift left logical	`sll`	`$1,$2,10`	$1 = $2 << 10	Shift left by constant
shift right logical	`srl`	`$1,$2,10`	$1 = $2 >> 10	Shift right by constant
load word	`lw`	`$1,100($2)`	$1 = Memory[$2+100]	Data from memory to register
load byte	`lb`	`$1,100($2)`	$1 = Memory[$2+100]	Data from memory to register
load byte unsigned	`lbu`	`$1,100($2)`	$1 = Memory[$2+100]	Data from mem. to reg.; no sign extension
store word	`sw`	`$1,100($2)`	Memory[$2+100] = $1	Data from register to memory
store byte	`sb`	`$1,100($2)`	Memory[$2+100] = $1	Data from register to memory
load address	`la`	`$1,var`	$1 = &var	Load variable address
branch on equal	`beq`	`$1,$2,100`	if ($1 = = $2) go to PC+4+100	Equal test; PC relative branch
branch on not equal	`bne`	`$1,$2,100`	if ($1 != $2) go to PC+4+100	Not equal test; PC relative
set on less than	`slt`	`$1,$2,$3`	if ($2 < $3) $1 = 1; else $1 = 0	Compare less than; 2's complement
set on less than immediate	`slti`	`$1,$2,100`	if ($2 < 100) $1 = 1; else $1 = 0	Compare < constant; 2's complement
set on less than unsigned	`sltu`	`$1,$2,$3`	if ($2 < $3) $1 = 1; else $1 = 0	Compare less than; natural number
set on less than imm. unsigned	`sltiu`	`$1,$2,100`	if ($2 < 100) $1 = 1; else $1 = 0	Compare constant; natural number
jump	`j`	`10000`	go to 10000	Jump to target address
jump register	`jr`	`$31`	go to $31	For switch, procedure return
jump and link	`jal`	`10000`	$31 = PC + 4;go to 10000	For procedure call

Register Usage

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

Name	Register 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

Name	Usage
$f0, $f1, …, $f31	Single precision floating point registers
$f0, $f2, …, $f30	Double precision floating point registers

1) Una possibile soluzione per il primo micro-benchmark e' (*la funzione fib_it viene chiamata con input 10: fib_it(10)*):

```
fib_it:
#_____________________________________________________________________________________BB1
        addi $t1, $zero, 0              # first = 0
        addi $t2, $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, $t1, $t2              # tmp = first+second
        add  $t1, $zero, $t2            # first = second
        add  $t2, $zero, $t0            # second = tmp
        j while

end_while:
#_____________________________________________________________________________________BB4
        add  $v0, $t1, $zero            # in t1 ho fib
        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 (B) o una jump incondizionata (J o JR) o RET (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. Sia E_i il numero di esecuzioni del basic-block i-esimo, derivabile dall'analisi del funzionamento programma quando il parametro di ingresso n≡a0 vale 10, come indicato dal testo dell'esercizio. Ricordando $C_{CPU} = N_{CPU} \cdot \overline{CPI} = N_{CPU} \cdot \sum_{k=ALJ,B,LS} \frac{N_{CPU,k}}{N_{CPU}} \cdot CPI_k = \sum_{k=ALJ,B,LS}^{T} N_{CPU,k} \cdot CPI_k$ e applicando questa relazione a ciascun basic-block i-esimo e sommando tutti i contributi i-esimi ai cicli $C_{CPU,i}$ di ogni basic block si ottengono i cicli totali $C_{CPU} = \sum_{i=1..4} C_{CPU,i}$

BB_i	$N_{CPU,ALJ}$ (CPI_{ALJ} = 1)	$N_{CPU,B}$ (CPI_{ALJ} = 3)	$N_{CPU,LS}$ (CPI_{LS} =10)	$(N_{CPU,k} \times CPI_k)_i$	E_i	$C_{CPU,i} = (N_{CPU,k} \times CPI_k)_i * E_i$
BB1	2	0	0	2	1	2 cc
BB2	2	1	0	5	11	55 cc
BB3	4	0	0	4	10	40 cc
BB4	2	0	0	2	1	2 cc
C_{CPU} (in cicli di clock)						**99 cc**

Si ha quindi che:

$$T_{CPU}^{1} = \frac{C_{CPU}^{1}}{f_{CPU}} = \frac{99}{10^9} = 99ns$$

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

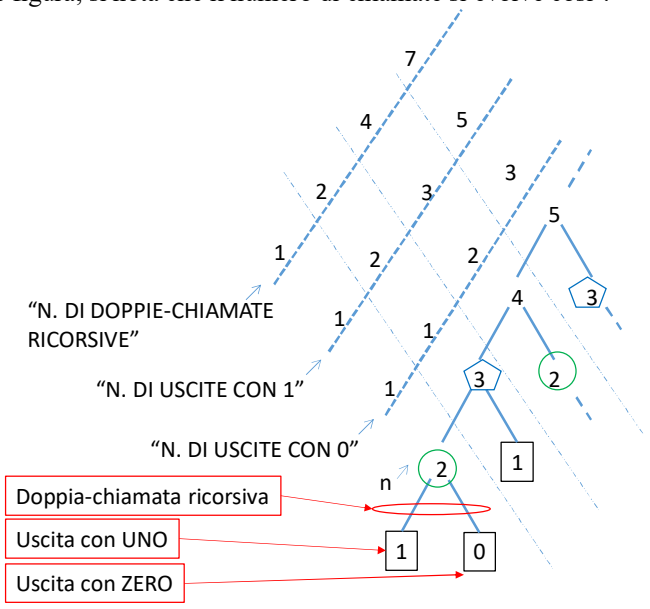
```
fib_rc:
#_____________________________________________________________________________________BB1
        addi $sp, $sp, -8               # riservo due word nello stack
        sw   $ra, 0($sp)                # salvo l'indirizzo di ritorno nella prima word dello stack
        add  $v0, $zero, $zero          # 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
```

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

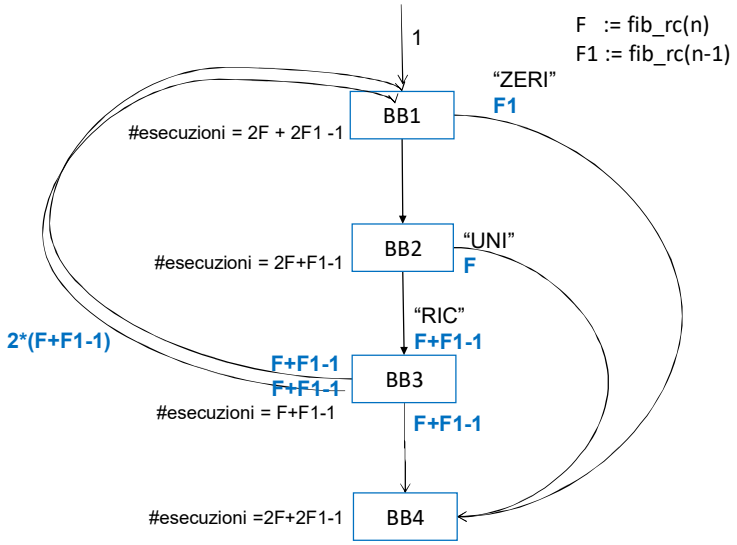


n	Fn
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55

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 significa 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

BB$_i$	$N_{CPU,ALJ}$ (CPI_{ALJ} = 1)	$N_{CPU,B}$ (CPI_{ALJ} = 3)	$N_{CPU,LS}$ (CPI_{LS} =10)	$(N_{CPU,k} \times CPI_k)_i$	E_i	$C_{CPU,i} = (N_{CPU,k} \times CPI_k)_i * E_i$
BB1	2	1	1	15	177	2655 cc
BB2	1	1	0	4	143	572 cc
BB3	5	0	4	45	88	3960 cc
BB4	2	0	1	12	177	2124 cc
				C_{CPU} (in cicli di clock)		**9311 cc**

Si ha quindi che:

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

2A) L'istruzione syscall fornisce al codice che si trova nello spazio di indirizzamento Utente un meccanismo controllato per ottenere accesso alle funzionalita' del Nucleo (o Kernel) del Sistema Operativo. Il codice e i dati che si trovano nello spazio di indirizzamento del Nucleo NON risultano normalmente accessibili al codice che si trova nello spazio di indirizzamento Utente (Figura 1).

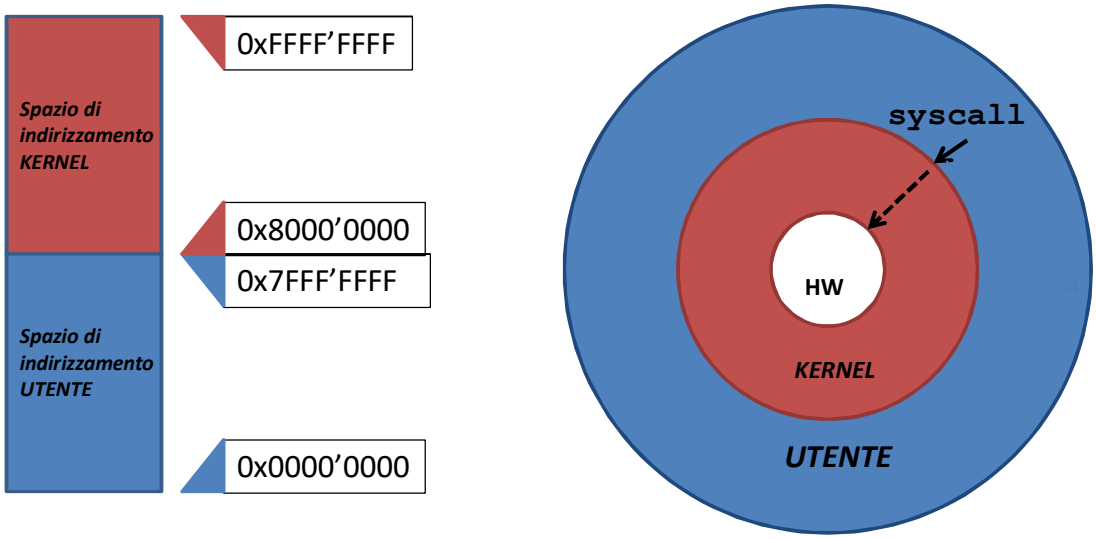


Figura 1. A sinistra, lo spazio di indirizzamento KERNEL e USER nel processore MIPS. A destra, e' rappresentata l'azione della syscall.

2B) Nel momento in cui il processore esegue l'istruzione syscall, vengono modificati alcuni registri speciali dell'architettura MIPS. In particolare: i registri speciali STATUS (bit K e E), CAUSE (bit 5-2), EPC; inoltre, possono essere modificati i registri generali $k0 e $k1.

2C) Passo-passo, cio' che avviene DURANTE l'esecuzione e':
i) i bit 5-0 del registro STATUS vengono traslati verso sinistra di due posizioni ed andranno a costituire le due coppie di bit K e E precedente (bit 3-2) e precedente-precedente (bit 5-4);
ii) i bit 1-0 del registro STATUS vengono inizializzati coi valori 00 indicando rispettivamente che il processore si trova da quel momento in stato Kernel e che gli interrupt sono disabilitati;

iii) i bit 5-2 del registro CAUSE vengono posti al valore 0x8 (o 1000) per indicare che si sta servendo una eccezione del tipo "trap" ovvero la syscall appunto;
iv) il registro EPC viene inizializzato con il PC dell'istruzione syscall (nota: il registro BADVADDR NON VIENE modificato); (nota: il registro $ra NON deve essere modificato perche' potrebbe essere gia' in uso);
v) il PC viene inizializzato con il valore (fisso) 0x8000'0080 e quindi l'esecuzione procede da tale punto del codice Kernel
vi) i registri generali $k0 e $k1 sono a questo punto di pieno dominio del codice Kernel che ne disporra' a suo piacimento, eventualmente modificandoli;
vii) si esegue la routine di servizio che gestisce la syscall, utilizzando eventualmente altri registri generali e in particolare $v0 tipicamente contiene il "numero" o "nome" del servizio richiesto al Sistema Operativo, predisposti PRIMA della chiamata alla syscall, come pure altri registri quali ad es. $a0 possono contenere ulteriori parametri di ingresso; DOPO la chiamata alla syscall, altri registri come $v0 stesso possono contenere dei risultati;
viii) deve infine essere eseguita la istruzione rfe (Return From Exception) che non fa altro che traslare verso destra i bit 5-0 del registro STATUS, in particolare disponendo nei bit K ed E i valori precenti al servizio della system call; dato che la system call e' chiamata dal codice Utente, ora il bit 1 (bit K attuale) varra' di nuovo 0.
ix) il controllo ritorna all'istruzione puntata da EPC+4 predisponendo in un registro di appoggio (es. $k1) il valore del EPC+4 ed effettuando (es.) "jr $k1" al termine della routine generale di gestione dell'eccezione.

2D) Nella figura 2 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.

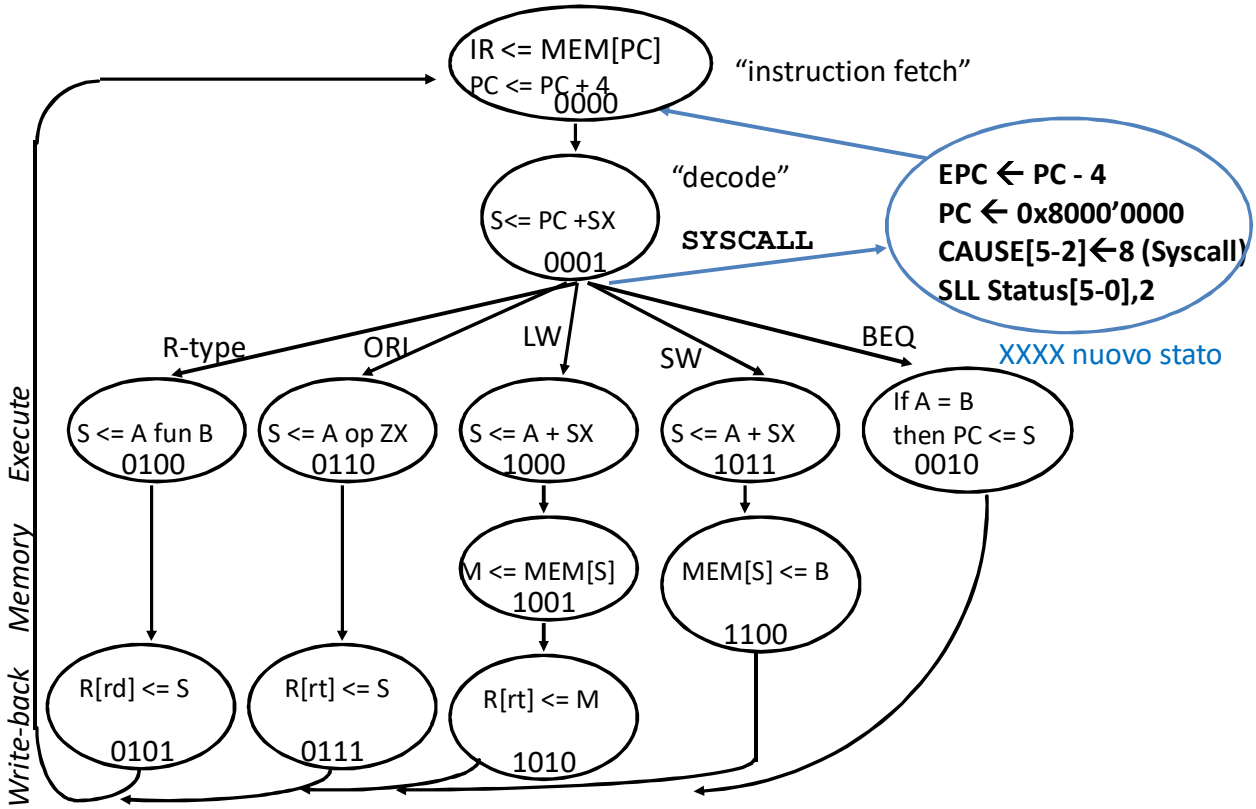


Figura 2. Modifica al ciclo Fetch-Decode-Execute-Write_back per implementare la syscall.

Soluzione RISC-V

RISCV Instructions

Instruction coding (hexadecimal) opcode+funct3+{funct7,imm}	Instruction	Example		Meaning	Comments
33+0+00/3b+0+00	**add**	`add/addw`	`x5,x6,x7`	x5 = x6 + x7	add (addw->64bits) 3 operands; exception possible
33+0+20/3b+0+20	**subtract**	`sub/subw`	`x5,x6,x7`	x5 = x6 – x7	sub (subw->64bits) 3 operands; exception possible
13+0+imm/1b+0+imm	**add immediate**	`addi/addiw`	`x5,x6,100`	x5 = x6 + 100	addi(addiw->64bits) + constant ; exception possible
33+0+01/3b+0+01	**multiply**	`mul/mulw`	`x5,x6, x7`	x5= x6 * x7	(signed/word) Lower 64 bits of 128-bits product
33+01+01	**multiply high**	`mulh`	`x5,x6,x7`	x5=x6 * x7	Higher 64bits of 128-bits product
33+4+01/3b+4+01	**division**	`div/divw`	`x5,x6,x7`	x5 = x6/x7	(signed/unsigned) division
33+6+01/3b+6+01	**reminder**	`rem/remw`	`x5,x6,x7`	x5 = x6 % x7	Create copy of Hi (Create a copy of Lo)
33+2+0/33+3+0	**set on less than**	`slt/sltu`	`x5,x6,x7`	if (x5 < x6) x5 = 1; else x5 = 0	(signed/unsigned) compare x5 and x6 (less than)
13+2+imm/13+3+imm	**set on less than immediate**	`slti/sltiu`	`x5,x6, x7`	if (x5 < x6) x5= 1; else x5 = 0	(signed/unsigned) compare x5 and x6 (less than)
33+7+0/33+6+0/33+4+0	**and / or / xor**	`and/or/xor`	`x5,x6,x7`	x5=x6&x7 / x6|x7 / x6^x7	3 register operands; Logical AND/OR/XOR
13+7+imm/13+6+imm/13+4+imm	**and /or / xor immediate**	`andi/ori/xori`	`x5,x6,100`	x5 = x6 & 100 / x6 | 100 / x6 ^100	Logical AND/OR/XOR register, constant
33+1+0/3b+1+0	**shift left logical**	`sll/sllw`	`x5,x6,x7`	x5 = x6 << x7	Shift left by register (sllw->64bits)
13+1+imm/1b+1+imm	**shift left logical immediate**	`slli/slliw`	`x5,x6,10`	x5 = x6 << 10	Shift left by the immediate value
33+5+0/3b+5+0	**shift right logical**	`srl/srlw`	`x5,x6,x7`	x5 = x6 >> x7	Shift right by register (for arithmetic: sign is preserved)
13+5+imm/1b+5+imm	**shift right logical immediate**	`srli/srliw`	`x5,x6,10`	x5 = x6 >> 10	Shift left by variable
33+5+20/3b+5+20	**shift right arithmetic**	`sra/sraw`	`x5,x6,x7`	x5 = x6 >> x7 (arith.)	(64bits) Shift right by variable (sign is preserved)
13+5+imm/1b+5+imm	**shift right arithmetic immediate**	`srai/sraiw`	`x5,x6,10`	x5 = x6 >> 10 (arith.)	Shift right by immediate value (sraiw->64bits)
03+3+imm/03+2+imm/03+0+imm	**load dword / word / byte**	`ld/lw/lb`	`x5,100(x6)`	x5 = MEM[x6+100]	Data from memory to register
03+6+imm/03+4+imm	**load word / byte unsigned**	`lwu/lbu`	`x5,100(x6)`	x5 = MEM[x6+100]	Data from mem. To reg.; no sign extension (lwu->64bits)
23+3+imm/23+2+imm/23+0+imm	**store dword / word / byte**	`sd/sw/sb`	`x5,100(x6)`	MEM[x6+100] = x5	Data from register to memory
37+imm[31:12] (no funct3)	**load upper immediate**	`lui`	`x5,0x1234`	x5=0x1234'5000	load most significant 20 bits
PSEUDOINSTRUCTION	**load address**	`la`	`x5,var`	x5 = &var	Load address of var (lui x5,H20(&var);ori x12, L12(&var)) H20=high 20 bit of &var; L12=low 12 bits of &var
PSEUDOINSTRUCTION	**jump**	`j/b`	`1000`	go to 1000	`(PSEUDO) INSTR. IS: jal x0,offset`
PSEUDOINSTRUCTION	**jump and link (offset)**	`jal`	`100`	x1=(PC+4): go to PC+100	`(PSEUDO) INSTR. IS: jal x1,offset`
PSEUDOINSTRUCTION	**return from procedure**	`ret`		PC=x1	`(PSEUDO) INSTR. IS: jalr x0,0(x1)`
67+0+imm	**jump and link register**	`jalr`	`x1, 100(x5)`	x1 = (PC + 4);go to x5+100	Procedure return; indirect call
63+0+(imm ÷ 2)	**branch on equal**	`beq`	`x5,x6,100`	if (x5 = = x6) go to PC+100	Equal test; PC relative branch
63+1+(imm ÷ 2)	**branch on not equal**	`bne`	`x5,x6,100`	if (x5 != x6) go to PC+100	Not equal test; PC relative
73+0+0 (rs1=0,rs2=0,rd=0)	**ecall**	`ecall`		call OS service number in a7	See table of system calls below
73+0+8 (rs1=0,rs2=2,rd=0)	**sret**	`sret`		privileged level is set to user mode	Exit Kernel Mode
PSEUDOINSTRUCTION	**move**	`mv`	`x5,x6`	x5 = x6	`(PSEUDO) INSTR. IS: add x5,x0,x6`
PSEUDOINSTRUCTION	**load immediate**	`li`	`x5,100`	x5 - 100	`(PSEUDO) INSTR. IS: addi x5,x0,100`
PSEUDOINSTRUCTION	**no operation (nop)**	`nop`		do nothing	`(PSEUDO) INSTR. IS: addi x0,x0,0`
53+0+{0,1}	**floating point add (z=s or d)**	`fadd.{s,d}`	`f0,f1,f2`	f0=f1+f2	Single or double precision add
53+0+{4,5}	**floating point subtract**	`fsub.{s,d}`	`f0,f1,f2`	f0=f1-f2	Single or double precision subtraction
53+0+{8,9}	**floating point multiplication**	`fmul.{s,d}`	`f0,f1,f2`	f0=f1*f2	Single or double precision multiplication
53+0+{c,d}	**floating point division**	`fdiv.{s,d}`	`f0,f1,f2`	f0=f1/f2	Single or double precision division
53+2+{10,11}	**floating point absolute value**	`fabs.{s,d}`	`f0,f1`	f0=ABS(f1)	`(PSEUDO) INSTR. IS: fsgnjx.{s,d} f0,f1`
53+0+{10,11}	**floating point move between f-regs**	`fmv.{s,d}`	`f0,f1`	f0←f1	`(PSEUDO) INSTR. IS: fsgnj.{s,d} f0,f1`
53+1+{10,11}	**floating point negate**	`fneg.{s,d}`	`f0,f1`	f0= – (f1)	`(PSEUDO) INSTR. IS: fsgnjn.{s,d} f0,f1`
53+0/1/2+{50,51}	**floating point compare**	`fle/flt/feq.{s,d}`	`x5,f0,f1`	x5=(f0<f1)	Single and double: compare f0 and f1 <=,<,==
53+0+{70,71}	**move between x (integer) and f regs**	`fmv.x.{s,d}`	`x5,f0`	x5=f0 (no conversion)	Copy (without conversion)
53+0+{78,79}	**move between f and x regs**	`fmv.{s,d}.x`	`f0,x5`	f0=x5 (no conversion)	Copy (without conversion)
7+2+imm/27+2+imm	**load/store floating point (32bit)**	`flw/fsw`	`f0,0(f1)`	f0←MEM[f1] / MEM[f1]←f0	Data from FP register to memory
7+3+imm/27+3+imm	**load/store floating point (64bit)**	`fld/fsd`	`f0,0(f1)`	f0←MEM[f1] / MEM[f1]←f0	Data from FP register to memory
53+7+21(rs2=0)/53+7+20 (rs2=1)	**convert from/to single to/from double**	`fcvt.d.s/fcvt.s.d`	`f0,f1`	f0=(double)f1/f0=(single)f1	Type conversion
53+7+{60,61}	**convert from {single,double} to integer**	`fcvt.w.{s,d}`	`x5,f0`	x5=(int)f0	Type conversion
53+7+{68,69}	**convert to {single,double} from integer**	`fcvt.{s,d}.w`	`f0,x5`	f0=({single,double})x5	Type conversion

Register Usage

Register	ABI Name	Usage
x0	**zero**	The constant value 0
x18-27	**s2-11**	Saved
x5-7, x28-31	**t0-2, t3-6**	Temporaries
x12-17	**a2-7**	Arguments

Register	ABI Name	Usage
x10-11	**a0-1**	Arguments and Results
x8,x2	**s0/fp,sp**	frame pointer, stack pointer
x1,x3	**ra, gp**	return address, global pointer

Register	ABI Name	Usage
f10,f11	**fa0-1**	Argument and Return values
f12-17	**fa2-7**	Function arguments
f8-f9, f18-27	**fs2-11**	Saved registers
f0 – f7, f28-31	**ft8-11**	Temporaries registers

System calls

Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Args
print_int	1	a0=integer to print	---
print_float	2	fa0=float to print	---
print_double	3	(fa0,fa1)=double to print	---
print_string	4	a0=address of ASCIIZ string to print	---
read_int	5	---	a0=integer

Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Arguments
read_float	6	---	fa0=float
read_double	7	---	fa0-fa1=double
read_string	8	a0=address of input buffer, a1=max chars to read	---
sbrk	9	a0=Number of bytes to be allocated	a1=pointer to allocated memory
exit	10	---	---

```
fib_it:
#_____________________________________________________BB1
   li   a4, 0            # first = 0
   li   a5, 1            # second = 1

while:
#_____________________________________________________BB2
   mv   t0, a0           # metto un attimo da parte a0
   addi a0, a0, -1       # n = n - 1
   beq  t0, x0,end_while # se n==?0 termino il ciclo

#_____________________________________________________BB3
   add  t0, a4, a5       # tmp = first+second
   mv   a4, a5           # first = second
   mv   a5, t0           # second = tmp
   j while

end_while:
#_____________________________________________________BB4
   mv   a0,a4            # in a5 ho fib
   ret                   # passo il controllo al chiamante
```

```
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
   li   t0, 0       # metto il valore fisso di confronto 0 in t0
   beq  a0, t0,exit # se t0==0 allora ho finito

#_____________________________________________________________________________________________BB2
   li   t0, 1       # metto il valore fisso di confronto 1 in t0
   beq  a0, t0,exit # se t1==1 allora ho finito

#_____________________________________________________________________________________________BB3

   addi a0, a0, -1  # a0 = n-1
   sw   a0, 4(sp)   # salva (n-1) nello stack
   jal  fib_rc      # chiama fib_rc(n-1): risultato in a0
   lw   t0, 4(sp)   # ripristino ex-a0 in t0 (n-1)
   sw   a0, 4(sp)   # salva fib_rc(n-1) nello stack
   addi a0, t0, -1  # a0=n-2   (era t0=n-1)
   jal  fib_rc      # chiama fib_rc(n-2): risultato in a0c
   lw   t0, 4(sp)   # ripristina ex-a0 in t0 (fib_rc(n-1))
   add  a0, a0, t0  # somma fib_rc(n-2) e fib_rc(n-1)

exit:
#_____________________________________________________________________________________________BB4
   lw   ra, 0(sp)   # recupero l'indirizzo di ritorno dallo stack
   addi sp, sp, 8   # incremento lo stack
   ret              # passo il controllo al chiamante
```