COMPITINO #1 di *CALCOLATORI ELETTRONICI 1* del 26-10-05 **MATRICOLA**_____

(versione riveduta da errori di stampa) **COGNOME**_____

NOME_____

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:>
<M1,M2,MR sono matrici di 'int'>

for (j = 0; j < 100; ++j) {
   for (k = 0; k < 100; ++k) {
      t = 0;
      for (i = 0; i < 100; ++i) {
          t += M1[j][i]*M2[i][k];
      }
      MR[j][k] = t;
    }
}
```

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

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

Successivamente, calcolare il tempo di esecuzione di ciascuno dei due benchmark ipotizzando che tali benchmark vengano eseguiti:

A. su un processore con frequenza di clock pari a 4 GHz, assumendo i seguenti valori per il CPI di ciascuna categoria di istruzioni: aritmetico-logiche-salti 1, branch 3, load-store 10.
B. 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 5.

Infine, si calcolino le medie aritmetica, geometrica e armonica dei tempi di esecuzione sulla macchina A e B, indicando quale delle due macchine (A o B) si ritiene migliore in base ai risultati ottenuti e motivando brevemente la scelta.

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

Una possibile soluzione per il primo micro-benchmark e':

```
#_____________________________________________________________________________________________BB1
       # Preparazione costanti
       addi $t3, $zero, 400          # occupazione di memoria di una riga della matrice
       addi $t4, $zero, 4            # occupazione di memoria di un elemento della matrice
       add  $t0, $zero, $zero        # associo j a t0 e faccio j=0

forj:
#_____________________________________________________________________________________________BB2
       slti $t6, $t0, 100            # se j<100 allora t6=1
       beq  $t6, $zero, endj         # se t6==0 (j>=100) allora salto alla fine del ciclo

#_____________________________________________________________________________________________BB3
       add  $t1, $zero, $zero        # associo k a t1 e faccio k=0

fork:
#_____________________________________________________________________________________________BB4
       slti $t6, $t1, 100            # se k<100 allora t6=1
       beq  $t6, $zero, endk         # se t6==0 (k>=100) allora salto alla fine del ciclo

#_____________________________________________________________________________________________BB5
       add  $a0, $zero, $zero        # associo t a a0 e faccio t=0
       add  $t2, $zero, $zero        # associo i a t2 e faccio i=0

fori:
#_____________________________________________________________________________________________BB6
       slti $t6, $t2, 100            # se i<100 allora t6=1
       beq  $t6, $zero, endi         # se t6==0 (i>=100) allora salto alla fine del ciclo

#_____________________________________________________________________________________________BB7
       # calcolo dell'indirizzo elemento M1[j][i]
       mult $t0, $t3                 # j * 400
       mflo $t6                      # prendo solo LO perche' j e' sempre < 100
       mult $t2, $t4                 # i * 4
       mflo $t7                      # prendo solo LO perche' i e' sempre < 100
       add  $t6, $t6, $t7            # offset elemento: (j * 400) + (i * 4)
       add  $t6, $a1, $t6            # indirizzo elemento: sommo base (a1) e offset (t6)
       lw   $t5, 0($t6)              # carico M1[j][i] in t5

       # calcolo dell'indirizzo elemento M2[i][k]
       mult $t2, $t3                 # i * 400
       mflo $t6                      # prendo solo LO perche' i e' sempre < 100
       mult $t1, $t4                 # k *  4
       mflo $t7                      # prendo solo LO perche' k e' sempre < 100
       add  $t6, $t6, $t7            # offset elemento: (i * 400) + (k * 4)
       add  $t6, $a2, $t6            # indirizzo elemento: sommo base (a2) e offset (t6)
       lw   $t8, 0($t6)              # carico M2[i][k] in t8

       # moltiplicazione
       mult $t5, $t8                 # moltiplicazione M1[j][i] * M2[i][k]
       mflo $t6                      # tronco a 32 bit (sto lavoroando su (int))
       add  $a0, $a0, $t6            # t += (int)(M1[j][i] * M2[i][k])

       # incremento il contatore del ciclo
       addi $t2, $t2, 1              # ++i
       j    fori

endi:
#_____________________________________________________________________________________________BB8
       # calcolo dell'indirizzo elemento MR[j][k]
       mult $t0, $t3                 # j * 400
       mflo $t6                      # prendo solo LO perche' j e' sempre < 100
       mult $t1, $t4                 # k * 4
       mflo $t7                      # prendo solo LO perche' k e' sempre < 100
       add  $t6, $t6, $t7            # offset elemento: (j * 400) + (k * 4)
       add  $t6, $a3, $t6            # indirizzo elemento: sommo base (a3) e offset (t6)
       sw   $a0, 0($t6)              # scrivo MR[j][k] <-- t

       # incremento il contatore del ciclo
       add  $t1, $t1, 1              # ++k
       j    fork
endk:
#_____________________________________________________________________________________________BB9
       # incremento il contatore del ciclo
       addi $t0, $t0, 1              # ++j
       j    forj

endj:
```

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_{CPUa}	cBi_{CPUb}	$Cbi_{CPUa}= cBi_{CPUa}*Ni$	$Cbi_{CPUb}= cBi_{CPUb}*Ni$
BB1	1	3	0	0	3	3	3	3
BB2	101	1	1	0	4	4	404	404
BB3	100	1	0	0	1	1	100	100
BB4	10100	1	1	0	4	4	40400	40400
BB5	10000	2	0	0	2	2	20000	20000
BB6	1010000	1	1	0	4	4	4040000	4040000
BB7	1000000	17	0	2	37	27	37000000	27000000
BB8	10000	8	0	1	18	13	180000	130000
BB9	100	2	0	0	2	2	200	200
Ccpu (in cicli di clock)							**41281107**	**31231107**

Si ha quindi che:

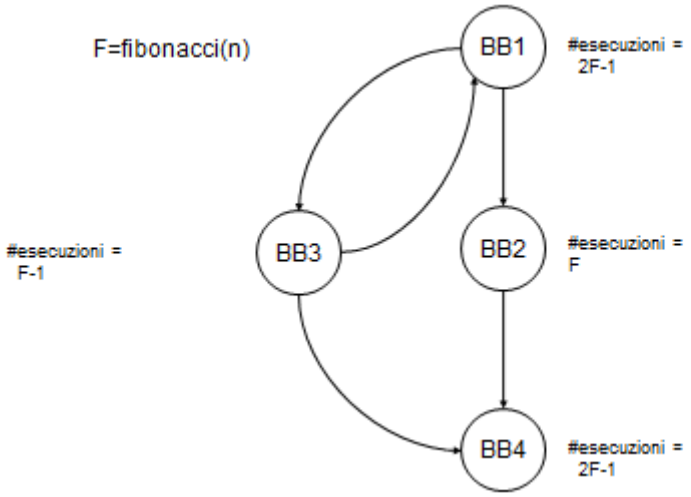
$$T^1_{CPU_A} = \frac{C_{CPU_A}}{f_{CPU_A}} = \frac{41281107}{4 \cdot 10^9} \cong 10.320ms$$

$$T^1_{CPU_B} = \frac{C_{CPU_B}}{f_{CPU_B}} = \frac{31231107}{1 \cdot 10^9} \cong 31.231ms$$

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

```
fibo:
#_____________________________________________________________________________BB1
        addi $sp, $sp, -8                   # riservo due word nello stack
        sw   $ra, 0($sp)                    # salvo l'indirizzo di ritorno nello stack

        addi $t0, $zero, 2                  # metto il valore fisso di confronto 2 in t0 (t0 = 2)
        slt  $t1, $t0, $a0                  # se 2<n allora t1=1, (ovvero t1=0 se n<=2)
        bne  $t1, $zero, return             # se t1!=0 (n>2) allora calcolo iterativamente
                                            # (etichetta 'return')

#_____________________________________________________________________________BB2
        addi $v0, $zero, 1                  # restituisco 1 (casi n<=2)
        j exit

return:
#_____________________________________________________________________________BB3
        addi $s0, $a0, -2                   # s0 = n-2
        addi $a0, $a0, -1                   # a0 = n-1
        sw   $s0, 4($sp)                    # salva s0 nello stack
        jal  fibo                           # chiama fibo(n-1): risultato in v0
        lw   $a0, 4($sp)                    # ripristino ex-s0 in a0 (n-2)
        sw   $v0, 4($sp)                    # salva v0 nello stack
        jal  fibo                           # chiama fibo(n-2): risultato in v0
        lw   $s0, 4($sp)                    # ripristina ex-v0 in a0 (fibo(n-1))
        add  $v0, $v0, $s0                  # 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
```

Una volta eseguito il partizionamento del programma in basic block può essere utile per calcolare il Ccpu fare uno schema dell'ordine in cui i basic block vengono eseguiti:



	Ni	***ALJ***	***B***	***LS***	cBi_{CPUa}	cBi_{CPUb}	$Cbi_{CPUa}= cBi_{CPUa}*Ni$	$Cbi_{CPUa}= cBi_{CPUa}*Ni$
BB1	109	3	1	1	16	11	1744	1199
BB2	55	2	0	0	2	2	110	110
BB3	54	5	0	4	45	25	2430	1350
BB4	109	2	0	1	12	7	1308	763
Ccpu (in cicli di clock)							**5592**	**3422**

$$T^2_{CPU_A} = \frac{C_{CPU_A}}{f_{CPU_A}} = \frac{5592}{4 \cdot 10^9} \cong 1.398 \mu s$$

$$T^2_{CPU_B} = \frac{C_{CPU_B}}{f_{CPU_B}} = \frac{3422}{1 \cdot 10^9} \cong 3.422 \mu s$$

A questo punto e' possibile procedere al calcolo delle tre medie richieste:

$$AMT_{CPU_A} = \frac{1}{2} \cdot (T^1_{CPU_A} + T^2_{CPU_A}) = 5.161 ms$$

$$AMT_{CPU_B} = \frac{1}{2} \cdot (T^1_{CPU_B} + T^2_{CPU_B}) = 15.617 ms$$

$$HMT_{CPU_A} = \left(\frac{1}{2} \cdot (\frac{1}{T^1_{CPU_A}} + \frac{1}{T^2_{CPU_A}}) \right)^{-1} = 2.796 \mu s$$

$$HMT_{CPU_B} = \left(\frac{1}{2} \cdot (\frac{1}{T^1_{CPU_B}} + \frac{1}{T^2_{CPU_B}}) \right)^{-1} = 6.843 \mu s$$

$$GMT_{CPU_A} = \left(T^1_{CPU_A} \cdot T^2_{CPU_A} \right)^{0.5} = 120.11 \mu s$$

$$GMT_{CPU_B} = \left(T^1_{CPU_B} \cdot T^2_{CPU_B} \right)^{0.5} = 326.91 \mu s$$

Dai risultati ottenuti la macchina A risulta piu' veloce della macchina B, con tutte e tre le metriche considerate.

Soluzione RISCV

RISCV Instructions (RV64IMFD)
v191222

Instruction coding (hexadecimal) opcode+funct3+[funct7,imm]	Instruction	Example		Meaning	Comments (** instructions available only in RV64, i.e. 64-bit case)
33+0+00/3b+0+00	**add**	`add/addw`	`x5,x6,x7`	x5 ← x6 + x7	Add two operands; exception possible (addw**)
33+0+20/3b+0+20	**subtract**	`sub/subw`	`x5,x6,x7`	x5 ← x6 – x7	Subtracts two operands; exception possible (subw**)
13+0+imm/1b+0+imm	**add immediate**	`addi/addiw`	`x5,x6,100`	x5 ← x6 + 100	Add a constant ; exception possible (addiw**)
33+0+01/3b+0+01	**multiply**	`mul/mulw`	`x5,x6, x7`	x5 ← x6 * x7	(signed/word) Lower 64 bits of 128-bits product (mulw**)
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/word) division (divw**)
33+6+01/3b+6+01	**reminder**	`rem/remw`	`x5,x6,x7`	x5 ← x6 % x7	Reminder of the division (remw**)
33+2+0/33+3+0	**set on less than**	`slt/sltu`	`x5,x6,x7`	if (x6 < x7) x5 ← 1; else x5 ← 0	(signed/unsigned) compare x6 and x7 (less than)
13+2+imm/13+3+imm	**set on less than immediate**	`slti/sltiu`	`x5,x6,100`	if (x6 < 100) x5← 1; else x5 ← 0	(signed/unsigned) compare x6 and 100 (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	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**)
13+1+imm/1b+1+imm	**shift left logical immediate**	`slli/slliw`	`x5,x6,10`	x5 ← x6 << 10	Shift left by the immediate value (slliw**)
33+5+0/3b+5+0	**shift right logical**	`srl/srlw`	`x5,x6,x7`	x5 ← x6 >> x7	Shift right by register (srlw**)
13+5+imm/1b+5+imm	**shift right logical immediate**	`srli/srliw`	`x5,x6,10`	x5 ← x6 >> 10	Shift left by immediate value (srliw**)
33+5+20/3b+5+20	**shift right arithmetic**	`sra/sraw`	`x5,x6,x7`	x5 ← x6 >> x7 (arith.)	Shift right by register (sign is preserved) (sraw**)
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**)
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**)
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 (sw**)
37+imm[31:12] (no funct3)	**load upper immediate**	`lui`	`x5,0x12345`	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/beq x0,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)/63+1+(imm÷2)	**branch on equal / not-equal**	`beq/bne`	`x5,x6,100`	if (x5 = =/!= x6) PC=PC+100	Equal / Not-equal test; PC relative branch
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`		Exit Supervisor 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}/53+0+{4,5}	**floating point add/sub**	`fadd.{s,d}/fsub.{s,d}`	`f0,f1,f2`	f0←f1+f2 / f0←f1-f2	Single or double precision add / subtract
53+0+{8,9}/53+0+{c,d}	**floating point multiplication/division**	`fmul.{s,d}/fdiv.{s,d}`	`f0,f1,f2`	f0←f1*f2 / f0←f1/f2	Single or double precision multiplication / division
53+2+{10,11}	**floating point absolute value**	`fabs.{s,d}`	`f0,f1`	f0← | 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 (no conversion)
53+0+{78,79}	**move between f and x regs**	`fmv.{s,d}.x`	`f0,x5`	f0←x5 (no conversion)	Copy (no conversion)
7+2+imm/27+2+imm	**load/store floating point (32bit)**	`flw/fsw`	`f0,0(x5)`	f0←MEM[x5] / MEM[x5]←f0	Data from FP register to memory
7+3+imm/27+3+imm	**load/store floating point (64bit)**	`fld/fsd`	`f0,0(x5)`	f0←MEM[x5] / MEM[x5]←f0	Data from FP register to memory
53+7+21(rs2=0)/53+7+20 (rs2=1)	**convert to/from double from/to single**	`fcvt.d.s/fcvt.s.d`	`f0,f1`	f0← (double)f1 / f0← (single)f1	Type conversion
53+7+{60,61}	**convert to integer from {single,double}**	`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
x10-x11	**a0-a1**	arguments and results
x9, x18-x27	**s1, s2-s11**	Saved
x5-7, x28-x31	**t0-t2, t3-t6**	Temporaries
x12-x17	**a2-a7**	Arguments

Register	ABI Name	Usage
x0	**zero**	The constant value 0
x8, x2	**s0/fp, sp**	frame pointer, stack pointer
x1, x3	**ra, gp**	return address, global pointer
x4	**tp**	thread pointer

Register	ABI Name	Usage
f10-f11	**fa0-fa1**	Argument and Return values
f8-f9, f18-f27	**fs0-fs1, fs2-fs11**	Saved registers
f0 – f7, f28-f31	**ft0-ft7, ft8-ft11**	Temporaries registers
f12-17	**fa2-fa7**	Function arguments

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=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=double
read_string	8	a0=address of input buffer, a1=max chars to read	---
sbrk	9	a0=Number of bytes to be allocated	a0=pointer to allocated memory
exit	10	---	---

Una soluzione al primo micro-benchmark e':

```
#_____________________________________________________________________________________________BB1
# Preparazione costanti
        addi t3, x0, 400  # occupazione di memoria di una riga della matrice
        addi t4, x0, 4    # occupazione di memoria di un elemento della matrice
        add t0, x0, x0    # associo j a t0 e faccio j=0
forj:
#_____________________________________________________________________________________________BB2
        slti t6, t0, 100  # se j<100 allora t6=1
        beq t6, x0 endj   # se t6==0 (j>=100) allora salto alla fine del ciclo
#_____________________________________________________________________________________________BB3
        add t1, x0, x0    # associo k a t1 e faccio k=0
fork:
#_____________________________________________________________________________________________BB4
        slti t6, t1, 100 # se k<100 allora t6=1
        beq t6, x0 endk  # se t6==0 (k>=100) allora salto alla fine del ciclo
#_____________________________________________________________________________________________BB5
        add  a0, x0, x0   # associo t a a2 e faccio t=0
        add  t2, x0, x0   # associo i a t2 e faccio i=0
fori:
#_____________________________________________________________________________________________BB6
        slti t6, t2, 100  # se i<100 allora t6=1
        beq t6, x0 endi   # se t6==0 (i>=100) allora salto alla fine del ciclo
#_____________________________________________________________________________________________BB7

 # calcolo dell'indirizzo elemento M1[j][i]
        mul t6, t0, t3   # j * 400 e  prendo solo LO perche' j e' sempre < 100
        mul a4, t2, t4   # i * 4 e prendo solo LO perche' i e' sempre < 100
        add t6, t6, a4   # offset elemento: (j * 400) + (i * 4)
        add t6, a1, t6   # indirizzo elemento: sommo base (a1) e offset (t6)
         lw t5, 0(t6)     # carico M1[j][i] in t5
# calcolo dell'indirizzo elemento M2[i][k]
        mul t6, t2, t3   # i * 400 e prendo solo LO perche' i e' sempre < 100
        mul a4, t1, t4   # k *  4 e prendo solo LO perche' k e' sempre < 100
        add t6, t6, a4   # offset elemento: (i * 400) + (k * 4)
        add t6, a2, t6   # indirizzo elemento: sommo base (a2) e offset (t6)
        lw a5, 0(t6)     # carico M2[i][k] in t8
# moltiplicazione
        mul t6, t5, a5   # moltiplicazione M1[j][i] * M2[i][k] e tronco 32bit
        add a0, a0, t6   # t += (int)(M1[j][i] * M2[i][k])
# incremento il contatore del ciclo
        addi t2, t2, 1  # ++i
        j fori
endi:
#_____________________________________________________________________________________________BB8
# calcolo dell'indirizzo elemento MR[j][k]
        mul t6, t0, t3   # j * 400 e prendo solo LO perche' j e' sempre < 100
        mul a4, t1, t4   # k * 4 prendo solo LO perche' k e' sempre < 100
        add t6, t6, a4   # offset elemento: (j * 400) + (k * 4)
        add t6, a3, t6  # indirizzo elemento: sommo base (a3) e offset (t6)
        sw a0, 0(t6)     # scrivo MR[j][k] <-- t
# incremento il contatore del ciclo
        addi t1, t1, 1    # ++k
        j fork
endk:
#_____________________________________________________________________________________________BB9

# incremento il contatore del ciclo
        addi t0, t0, 1   # ++j
        j forj
endj
```

Una soluzione al secondo micro-benchmark e':

```
.data
prompt:  .asciz "Inserire un numero intero: "
output:  .asciz "Il numero di Fibonacci e': "
nca:     .asciz "Il numero totale di cicli per il processore A e': "
ncb:     .asciz "\nIl numero totale di cicli per il processore B e': "
BB:      .asciz "\nConteggio dei BB: "
virgola: .asciz ","

.text
.globl main

main:
     la a0,  prompt
     addi a7,x0,4       # stampa della stringa "Inserire un numero intero: "
     ecall
     addi a7,x0,5       # lettura del valore inserito
     ecall
     add a4, x0, a0     # salvo il valore letto in a4
     jal fibo           # chiamata della funzione fibo()
     add s2, a0, x0     # salvo il risultato in s2
     la a0,  output     # stampa della stringa "Il numero di Fibonacci e': "
     addi a7,x0,4
     ecall
     add a0, s2, x0     # stampo il numero di fibonacci
     addi a7,x0,1
     ecall

fibo:
#_____________________________________________________________________________________________BB1
        addi sp, sp, -8         # riservo due word nello stack
        sw ra, 0(sp)            # salvo l'indirizzo di ritorno nella prima word dello stack
        addi t0, x0, 2          # metto il valore fisso di confronto 2 in t0 (t0 = 2)
        slt t1, t0, a4          # se 2<n allora t1=1, (ovvero t1=0 se n<=2)
        bne t1, x0 return       # se t1!=0 (n>2) allora calcolo iterativamente (etichetta 'return')

#_____________________________________________________________________________________________BB2
        addi a0, x0, 1          # restituisco 1 (casi n<=2)
        j exit

return:
#_____________________________________________________________________________________________BB3
        addi s2, a4, -2         # s2 = n-2
        addi a4, a4, -1         # a4 = n-1
        sw s2, 4(sp)            # salva s2 nello stack
        jal fibo                # chiama fibo(n-1): risultato in a0
        lw a4, 4(sp)            # ripristino ex-x18 in a4 (n-2)
        sw a0, 4(sp)            # salva a0 nello stack

        jal fibo                # chiama fibo(n-2): risultato in a0
        lw s2, 4(sp)            # ripristina ex-a0 in s2 (fibo(n-1))
        add a0, a0, s2          # 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
```