COMPITO di ARCHITETTURA DEI CALCOLATORI del 27-02-2020

MATRICOLA_____

COGNOME_____

NOME_____

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

NOTA: per l'esercizio 4 consegnare DUE files: il file del programma VERILOG e il file del diagramma temporale (screenshot o copy/paste)

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

```
{
   int r = 0, j=1;
   int k;

   while (j <= n) {
      for (k = 0; k < n; ++k) {
          if (T[j-1][k] != 0) {
             *nc += 1/((-T[j-1][k])*(-T[j-1][k]));
          } else {
             r = 1;
          }
      }
      ++j;
   }
   return (r);
}
```

```
float f = 0.0;
float A[][] =
   {{1.0,2.0,3.0},{4.0,5.0,6.0},{7.0,8.0,9.0}};

main()
{
    int ec;

    ec = num_cond(A, 3, &f);

    printf("esito=");
    printf(ec);
    printf("   n.cond=");
    printf(f);
    printf("\n");
}
```

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

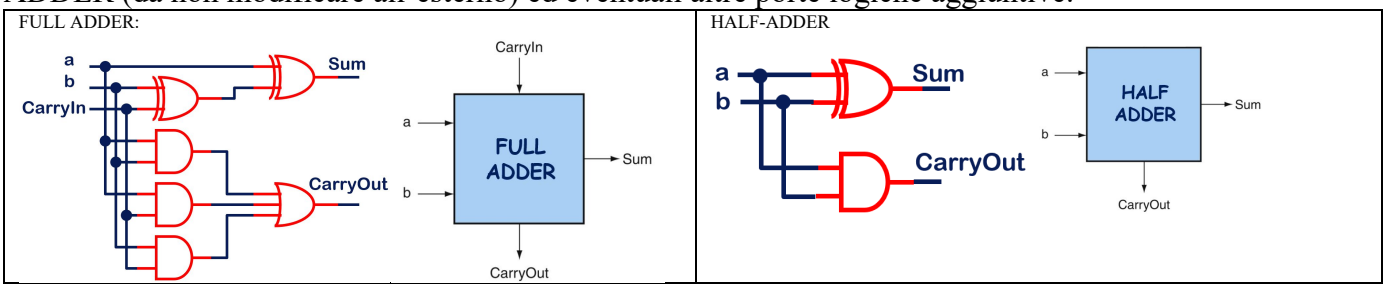
Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Args
print_int	1	a0=integer to print	---
print_float	2	fa0=float to print	---
print_double	3	fa0=double to print	---
print_string	4	a0=address of ASCIIZ string to print	---
read_int	5	---	a0=integer

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	---	---

COMPITO di ARCHITETTURA DEI CALCOLATORI del 27-02-2020

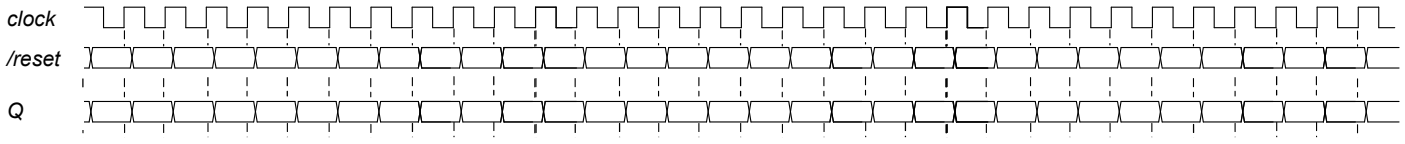
1) [4/30] Spiegare in dettaglio la differenza fra le istruzioni SLT, SLTI e SLTIU, SLTU e mostrare, con un esempio, il fatto che SLT e SLTU possono produrre risultati diversi nel confronto di due registri che contengano rispettivamente le rappresentazioni degli interi -4 e 3 (es. x5=-4 e x6=3).

2) [5/30] Realizzare la rete combinatoria di un FULL-ADDER utilizzando due reti combinatorie HALF-ADDER (da non modificare all'esterno) ed eventuali altre porte logiche aggiuntive.



3) [9/30] Realizzare in Verilog un contatore ad anello su 4 bit. Il testbench e' dato. Tracciare il diagramma di temporizzazione come verifica della correttezza dell'unità riportando i segnali clock, /reset, uscita Q che rappresenta il valore del conteggio ad ogni ciclo di clock. Nota: si può svolgere l'esercizio su carta oppure con l'ausilio del simulatore, salvando una copia dell'output (diagramma temporale) e del programma Verilog su USB-drive del docente.

```
`timescale 1ns/1ps
module testbench;
  reg clock,_reset;
  wire[3:0] Q;
  contatore_ad_anello rc(Q,clock,_reset);
  always #10 clock = ~clock;
  initial begin
    $display("time,\t clock,\t _reset,\t QQQQ");
    $monitor("%g,\t %b,\t %b,\t %b",
       $time,clock,_reset,Q);
    _reset=1'b1; clock=0;
    #5  _reset=1'b0;
    #20 _reset=1'b1;
    #600 $finish;
  end
endmodule
```



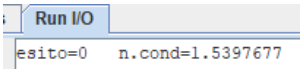
COMPITO di ARCHITETTURA DEI CALCOLATORI del 27-02-2020

SOLUZIONE

ESERCIZIO 1

```
base:  .word 0
.data
f:
 .float 0.0
A:
 .float 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0
ret:
 .asciz "\n"
esi:
 .asciz "esito="
nco:
 .asciz "   n.cond="


.text
 .globl main

   # non necessario su RARS: caricare exp.s e
ncond.s in sequenza
   # a3=**T
   # a1=n
   # a2=*nc
   # a0=r
   # t0=j
   # t1=k

num_cond:
   # spazio per a3,a0,t0,t1
   addi sp, sp, -24
   sw   ra, 0(sp)  # salva ra perche' usa altra f.
   sw   s0, 4(sp)  # salva fp perche' usa frame
   add  s0, sp, x0
   add  a0, x0, x0 # j = 1
   addi t0, x0, 1  # costante f.p. 1
   addi t2, x0, 1
   fcvt.s.w fa1, t2# costante f.p. 0
   addi t2, x0, 0
   fmv.s.x  fa0, t2# la codifica di 0.0 e' 0...0
while_ini:
   slt t2, a1, t0  # j>?n
   bne t2, x0, while_end   # se si while_end
   add t1, x0, x0  # k = 0
for_ini:
   slt t2, t1, a1  # k<?n
   beq  t2, x0, for_end   # j -1
   addi t3, t0, -1
   add  t2, t3, t3
   add  t2, t2, t3   # (j-1)*3
   add  t2, t2, t1   # (j-1)*3+k
   slli t2, t2, 2    # *4
   add  t2, t2, a3   # &T[j-1][k]
   flw  fa2, 0(t2)   # == 0.0
   feq.s t5,fa2,fa0
   bne  t5, x0, ramo_else
   sw   a3,  8(s0)   # salva a3
   sw   a0, 12(s0)   # salva a0
   sw   t0, 16(s0)   # salva t0
   sw   t1, 20(s0)   # salva t1
   fneg.s fa2, fa2   # cambio segno
   fmv.x.s a3, fa2
   fmul.s fa2, fa2, fa2 # fa2= (.)*(.)
   lw   t1, 20(s0)   # salva t1
   lw   t0, 16(s0)   # salva t0
   lw   a0, 12(s0)   # ripristina a0
   lw   a3, 8(s0)    # ripristina a3
   fdiv.s fa2, fa1, fa2   # 1/sqr(-T[][])
   flw  ft0, 0(a2)   # *nc
   fadd.s ft0, ft0, fa2   # +=
   fsw  ft0, 0(a2)   # *nc=
   j if_end
ramo_else:
   # r = 1
   addi a0, x0, 1
if_end:
   # ++k
   addi t1, t1, 1
   j for_ini
for_end:
   # ++j
   addi t0, t0, 1
   j while_ini
while_end:

   lw   ra, 0(s0)
   lw   s0, 4(s0)
   addi sp, sp, 24   # ritorna a0
   ret
main:
   # param.1
   la   a3,  A   # param.2
   addi a1, x0, 3   # param.3
   la   a2,  f
   jal  num_cond

   add  s0, a0, x0   #print "esi...
   la   a0,  esi
   addi a7,x0, 4
   ecall
   add  a0, s0, x0   # ripristina ec
   addi a7,x0, 1
   ecall   # print "nco...
   la   a0,  nco
   addi a7,x0, 4
   ecall   # print f
   la   a0,  f
   flw  fa0, 0(a0)
   addi a7,x0, 2
   ecall  # print ret
   la a0,  ret
   addi a7,x0, 4
   ecall   # exit

   addi a7,x0, 10
   ecall
```

OUTPUT:



ESERCIZIO 2

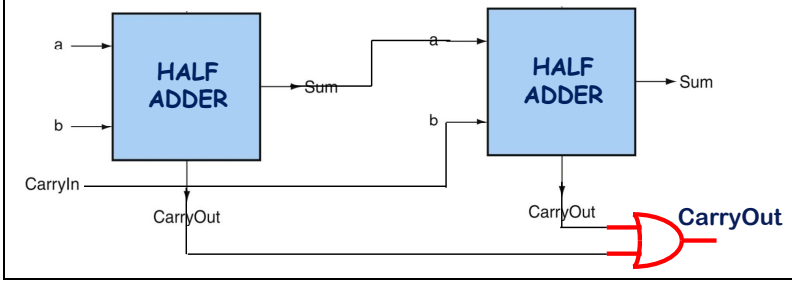
Aritmetica con segno e senza segno

- Le istruzioni aritmetiche fin qui esaminate operano su interi con segno
  - Gli operandi (a 64 bit, nei registri) sono rappresentati in complemento a due: i valori vanno da -2^{63} a $2^{63}-1$
  - Anche i byte-offset di `ld` e `sd` e il numero di istruzioni di cui saltare nella `bne/beq` sono interi con segno da -2^{11} a $2^{11}-1$

- E' possibile utilizzare anche operandi senza segno
  - In tal caso i valori vanno da 0 a $2^{64}-1$
  - Invece di `slt`, `slti` si useranno `slt`, `sltiu`
    - `slt` e `sltu` forniranno un risultato diverso se un operando e' negativo (es. -4 < 3 → 1 con `slt`, ma 0 con `sltu`)
      - Infatti il -4 verrebbe interpretato come un numero molto grande (es. 7 se gli oprandi fossero solo di 3 bit), quindi 7<3 risulterebbe falso

ESERCIZIO 3

Full-Adder-Composto realizzato con due Half-Adder



Inputs			Outputs	
a	b	CarryIn	CarryOut	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

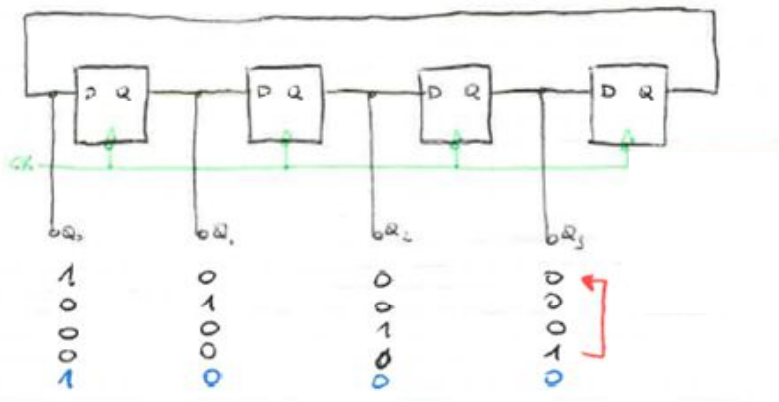
Dalla forma delle reti all'interno dell'half-adder e del full-adder si vede che la somma finale del full-adder-composto si ottiene mettendo in cascata due XOR, che sono l'elemento che genera la somma nell'half-adder: questo accade indipendentemente da come si commutino sugli ingressi del full-adder (i segnali a ,b, CarryIn). Questo giustifica la giustapposizione di due half-adder in cascata per generare la somma del full-adder: resta da risolvere il problema della generazione del CarryOut finale del full-adder. L'espressione del CarryOut del full-adder (si desume anche dalla figura della domanda): e' CarryOut=a*b+a*CarryIn+b*CarryIn. Dato che il primo half-adder produce come carry in uscita a*b e il secondo half-adder produce come carry di uscita (a⊕b)*CarryIn.
e osservando che l'espressione a*b+a*CarryIn+b*CarryIn equivale a a*b+(a⊕b)*CarryIn, si deduce che per ottenere il CarryOut basta fare l'OR delle due uscite degli half-adder.

COMPITO di ARCHITETTURA DEI CALCOLATORI del 27-02-2020

SOLUZIONE

ESERCIZIO 4

Ring Counter




```
`timescale 1ns/1ps
module testbench;
  reg clock,_reset;
  wire[3:0] Q;
  contatore_ad_anello rc(Q,clock,_reset);
  always #10 clock = ~clock;
  initial begin
    $display("time,\t clock,\t _reset,\t QQQQ");
    $monitor("%g,\t %b,\t %b,\t %b",
       $time,clock,_reset,Q);
    _reset=1'b1; clock=0;
    #5  _reset=1'b0;
    #20 _reset=1'b1;
    #600 $finish;
  end
endmodule
```

Codice Verilog del modulo da realizzare

```
module contatore_ad_anello(q,clock,_reset);
  input clock,_reset;
  output[3:0] q;
  reg[3:0] q;
  always @(posedge clock)
    if(!_reset)
      q<=4'b1000;
    else begin
      q[3]<=q[0];
      q[2]<=q[3];
      q[1]<=q[2];
      q[0]<=q[1];
    end
endmodule
```

Diagramma di Temporizzazione: