COMPITINO di ARCHITETTURA DEI CALCOLATORI del 28-01-2020 **(v1.2)** **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 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).

```
typedef struct header {
   struct header *ptr; unsigned size;
} Header;
static Header base = {NULL,0 };
static Header *freep = NULL;

void myfree(void *ap) {
   Header *bp, *p;  bp = (Header *)ap - 1;
   for (p = freep; !(bp > p && bp < p->ptr); p = p->ptr) {
       if (p >= p->ptr && (bp > p || bp < p->ptr)) break;
   }
   if (bp + bp->size == p->ptr) {
      bp->size += p->ptr->size;
      bp->ptr = p->ptr->ptr;
   } else bp->ptr = p->ptr;
   if (p + p->size == bp) {
      p->size += bp->size;
      p->ptr = bp->ptr;
   } else p->ptr = bp;
   freep = p;
}
```

```
void *alloc_and_print_pun(int sz) {
   void *p = sbrk(sz);
   print_string("p=");
   print_int(p);
   print_string("\n");
   return (p);
}

int main() {
   void *p0, *p1, *p2, *p3;
   base.ptr = &base; freep=&base;
   p0 = alloc_and_print_pun(0);
   p1 = alloc_and_print_pun(256);
   p2 = alloc_and_print_pun(256);
   p3 = alloc_and_print_pun(256);
   myfree(p1); myfree(p2); myfree(p3);
   p0 = alloc_and_print_pun(0);
}
```

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

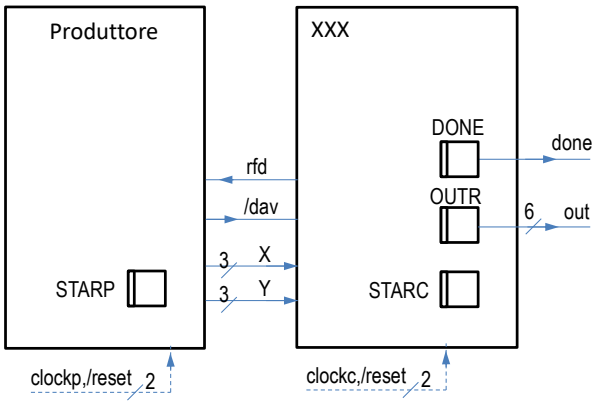
Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Args
print_int	1	a0=integer to print	---
print_float	2	fa0=float to print	---
print_double	3	fa0=double to print	---
print_string	4	a0=address of ASCIIZ string to print	---
read_int	5	---	a0=integer

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	---	---

2) [4/30] Descrivere in dettaglio la pseudo-istruzione JAL del RISC-V e il formato J, evidenziando le differenze rispetto al formato U (per MIPS: descrivere la pseudo-istruzione LA); discutere le istruzioni reali usate per implementare questa pseudo-istruzione.

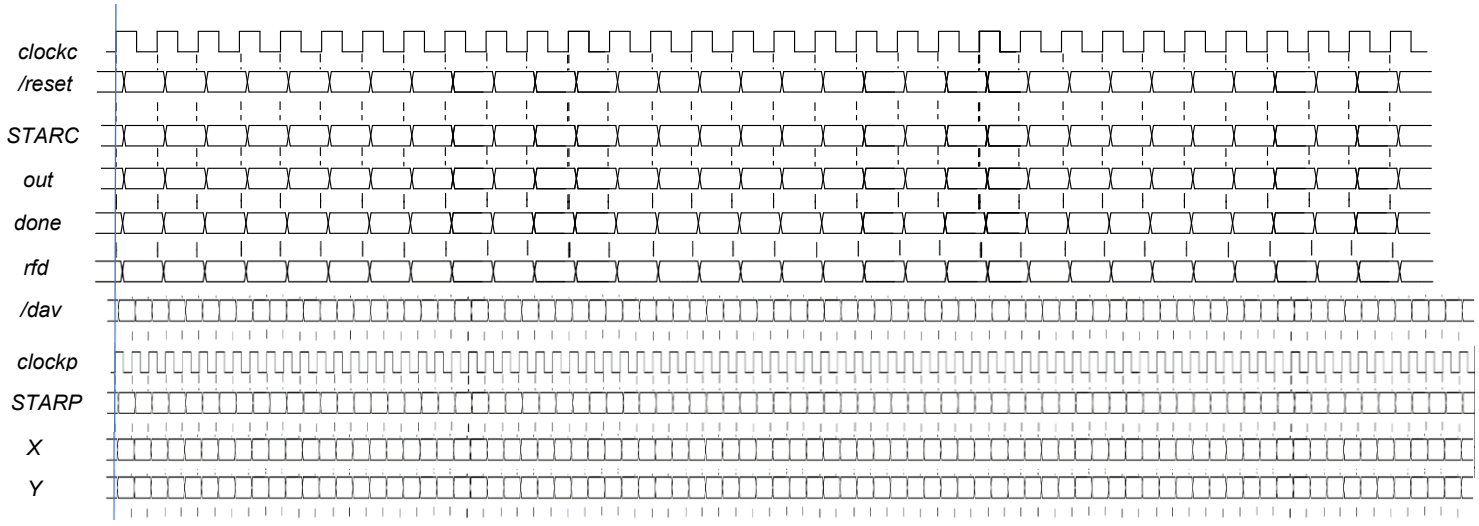
3) [4/30] Descrivere il "Read-Back command", utilizzato nella lettura dei contatori del timer 8254: quali informazioni debbono essere scritte nel registro di controllo e quali vantaggi comporta?

4) [10/30] Descrivere e sintetizzare in Verilog il modulo `XXX` di figura che funziona nel seguente modo: riceve due interi con segno su tre bit (`X` e `Y`) dal modulo produttore col quale colloquia tramite i segnali `rfd` e `/dav`; ogni tre coppie (Xi,Yi) il modulo presenta sull'uscita `out` il prodotto scalare $\sum_{i=1}^{3} X_i \cdot Y_i$, indicandone la disponibilita' abilitando il segnale `done` per 1 ciclo di clock di `XXX`. Il modulo `XXX` opera con un `clockc` di periodo 10ns mentre il modulo Produttore, con `clockp`, puo' avere periodo sia 4ns (attuale codice) che 12ns: verificare il corretto funzionamento per entrambi i valori di `clockp`. Il codice del produttore e del testbench e' dato qua sotto. **Tracciare il diagramma di temporizzazione** come verifica della correttezza del modulo realizzato. 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 testbench;
 reg reset_; initial begin reset_=0;  #1 reset_=1; #350; $stop;
end
 reg clockc ; initial clockc =0; always #5 clockc <=(!clockc);
 wire[2:0] STARC=Xxx.STAR;
 wire[5:0]out; wire done, rfd, dav_;
 reg clockp ; initial clockp =0; always #2 clockp <=(!clockp);
 wire[2:0] X,Y;
 wire[2:0] STARP=PRO.STAR;
 Consumatore Xxx(dav_,X,Y,  rfd,out,done,  clockc,reset_);
 Produttore  PRO(rfd,       dav_,X,Y,      clockp,reset_);
endmodule
```

```
module Produttore(rfd, dav_,X,Y, clock,reset_);
 input         rfd, clock,reset_;
 output        dav_;
 output [2:0]  X,Y;
 reg DAV_;     assign dav_=DAV_;
 reg [2:0]     PX,PY,QX,QY; assign X=PX, Y=PY;
 reg [1:0] STAR;   parameter S0=0, S1=1, S2=2;

 always @(reset_==0) begin QX='B010; QY='B111; DAV_=1; STAR=S0; end
 always @(posedge clock) if (reset_==1)  #0.1
  casex (STAR)
   S0: begin DAV_<=(rfd==1)?0:1; PX<=QX;   PY<=QY;   STAR<=(rfd==0)?S0:S1; end
   S1: begin DAV_<=(rfd==1)?0:1; QX<=PX+1; QY<=PY+3; STAR<=(rfd==1)?S1:S2; end
   S2: begin DAV_<=1; STAR<=(rfd==0)?S2:S0;end
 endcase
endmodule
```



COMPITINO di ARCHITETTURA DEI CALCOLATORI del 28-01-2020

SOLUZIONE

ESERCIZIO 1

```
base:   .word 0
        .word 0
freep: .word 0
RTN:    .asciz "\n"
puneq: .asciz "p="

.globl main
.text
myfree:
#-------------------------------------
# a0=ap,bp   t0=p, sizeof(Header)=8
          addi a0, a0, -8  # bp=ap-8
          la   t1,  freep  # &freep
          lw   t0,  0(t1)  # t1=p=freep
MF_INIFOR1: #scan list of free blocks
          slt  a6, t0, a0  # bp>?p
          lw   t2, 0(t0)   # t2=p->ptr
          slt  a5, a0, t2  # bp<?p->ptr
          and  a4, a5, a6
          bne  a4, x0, MF_FINEFOR1
          slt  a4, t0, t2  # p>=?p->ptr
                 # => !(p<?p->ptr)
          xor  a4, a4, -1  #not(.)=> p<?p->ptr
          or   a5, a5, a6  # (... || ...)
          and  a4, a4, a5  # (... && (.))
          bne  a4, x0, MF_FINEFOR1

          add  t0, t2, x0  # p=p->ptr
          j MF_INIFOR1

MF_FINEFOR1:
          lw   t3, 4(a0)   # bp->size
          add  t4, t3, a0  # bp+bp->size
          bne  t4, t2, MF_E1 # (.)!=p->ptr
          lw   t5, 4(t2)   # p->ptr->size
          add  t4, t3, t5  # bp->size+(.)
          sw   t4, 4(a0)   # bp->size=(.)
          lw   t5, 0(t2)   # p->ptr->ptr
          sw   t5, 0(a0)   # bp->ptr=(.)
          j MF_F1

MF_E1:
          sw   t2, 0(a0)   #bp->ptr=p->ptr

MF_F1:
          lw   t6, 4(t0)   # p->size
          add  a4, t6, t0  # p+p->size
          bne  a4, a0, MF_E2 # (.)!=bp
          lw   a5, 4(a0)   # bp->size
          add  t6, t6, a5  # p->size+(.)
          sw   t6, 4(t0)   # p->size=(.)
          lw   t6, 0(a0)   # bp->ptr
          sw   t6, 0(t0)   # p->ptr=(.)
          j MF_F2

MF_E2:
          sw   a0, 0(t0)   # p->ptr=bp

MF_F2:
          sw   t0, 0(t1)   # freep=p
          jr ra

#-------------------------------------
# a0=sz,  v1=p
alloc_and_print_pun:
          addi a7, x0, 9   # serv.9
          ecall            # sbrk
          add  a1, x0, a0  # salvo in a1
          la   a0, puneq   # stampa msg
          addi a7, x0, 4   # serv.4
          ecall            # print_str
          add  a0, x0, a1  # p
          addi a7, x0, 1   # serv.1
          ecall            # print_int
          la   a0, RTN     # stampa RTN
          addi a7, x0, 4   # serv.4
          ecall            # print_str
          add  a0, a1, x0  # return(p)
          jr ra
#-------------------------------------
main:  #s2=p0, s3=p1, s4=p2, s5=p3
#-------------------------------------
# PROLOGO
          addi sp, sp, -24  # alloco frame
          sw   ra, 20(sp)   # salvo OLD-ra
          sw   s0, 16(sp)   # salvo OLD-fp
          add  s0, x0, sp   # NUOVO fp
          sw   s5, 12(s0)   # salvo s5
          sw   s4, 8(s0)    # salvo s4
          sw   s3, 4(s0)    # salvo s3
          sw   s2, 0(s0)    # salvo s2
          la   t0, base     # &base
          la   t1, freep    # &freep
          sw   t0, 0(t0)    # base.ptr=&base
          sw   t0, 0(t1)    # freep=&base
          add  a0, x0, x0   # sz=0 (HEAPtop)
          jal  alloc_and_print_pun
          addi s2, a0, 0    # salva p0

          addi a0, x0, 256  # sz=256
          jal  alloc_and_print_pun
          addi s1, a0, 0    # salva p1

          addi a0, x0, 256  # sz=256
          jal  alloc_and_print_pun
          addi s4, a0, 0    # salva p2

          addi a0, x0, 256  # sz=256
          jal  alloc_and_print_pun
          addi s5, a0, 0    # salva p3

          add  a0, s3, x0   # p1
          jal  myfree
          add  a0, s4, x0   # p2
          jal  myfree
          add  a0, s5, x0   # p3
          jal  myfree
          add  a0, x0, x0   # sz=0 (HEAPtop)
          jal  alloc_and_print_pun
          add  s2, a0, x0   # salva p0
  #-------------------------------------
# EPILOGO
          lw   s2, 0(s0)    # ripristina s2
          lw   s3, 4(s0)    # ripristina s3
          lw   s4, 8(s0)    # ripristina s4
          lw   s5, 12(s0)   # ripristina s5
          lw   ra, 20(s0)   # ripristina ra
          lw   s0, 16(s0)   # ripristina fp
          addi sp, sp, 24   # DEALLOCO FRAME
          addi a7, x0, 10   # serv.10
          ecall             # exit
```

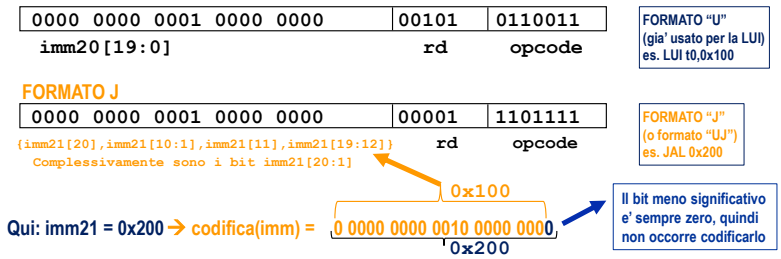
OUTPUT:

```
-- program is finished running --

p=268697600
p=268697600
p=268697856
p=268698112
p=268698368

-- program is finished running --
```

ESERCIZIO 2

La pseudo-istruzione JAL e il formato J

- Nel RISC-V l'etichetta "myfun" viene tradotta in un **offset** rispetto all'indirizzo della JAL stessa (simile a BEQ)
  - Rispetto alla BEQ non ho necessità di una condizione (due registri) → posso allora specificare un offset piu' ampio della BEQ, usando una variante del formato U → "formato J"



- Come per la BEQ, il valore immediato (imm), che rappresenta l'offset in byte che separa l'istruzione di salto (es. JAL) dal «target» del salto (es. myfun), è sempre multiplo di 2 (le istruzioni sono allineate alla half-word)
- Il registro rd codifica l'indice del registro ra (ovvero x1) che conterra' PC+4 (punti «A» e «B» della slide precedente)

ESERCIZIO 3

- **Read-Back Command**

  1) Si realizza in questo modo: scrittura in CWR delle seguenti info

  2) Successiva lettura di CRx

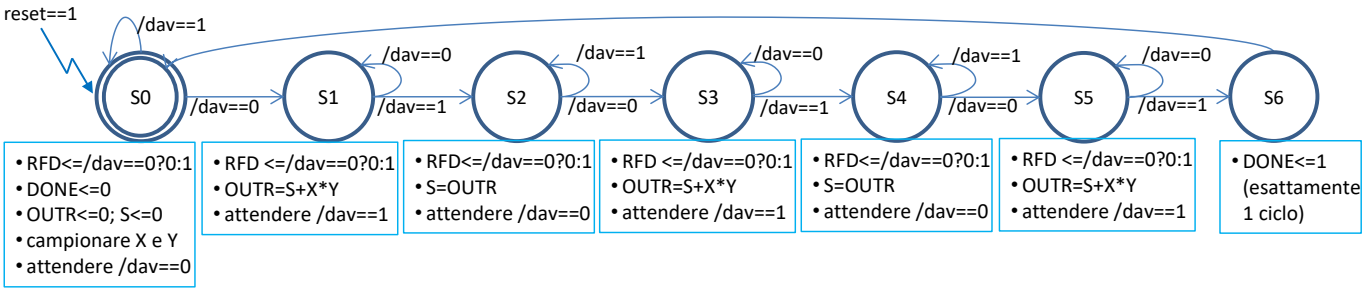
- **Vantaggi:**
  - Consente di leggere i conteggi di piu' contatori simultaneamente (caso di bit /COUNT attivo)
  - Consente di leggere come sono stati programmati i 3 contatori (caso di bit /STATUS attivo)
    - In particolare, i 6 bit meno significativi danno i valori precedentemente scritti in CWR per quel contatore

COMPITINO di ARCHITETTURA DEI CALCOLATORI del 28-01-2020

SOLUZIONE

ESERCIZIO 4

Un possibile diagramma degli stati:



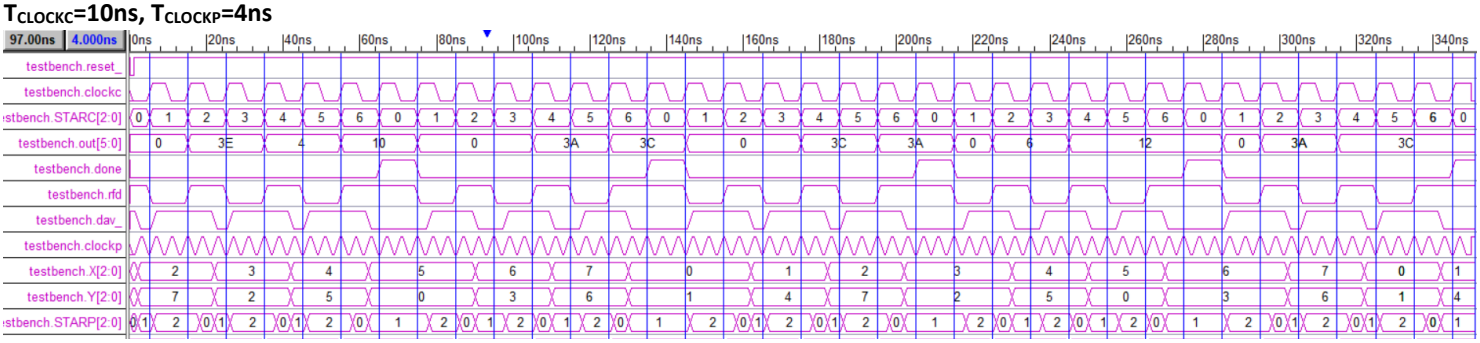
Codice Verilog del modulo da realizzare

```
module Consumatore(dav_,X,Y, rfd,out,done, clock,reset_);
 input        clock, reset_;
 output [5:0] out;
 output       rfd,done;
 input        dav_;
 input  [2:0] X,Y;
 reg    [5:0] OUTR,S;   assign out=OUTR;
 reg      RFD,DONE;   assign rfd=RFD, done=DONE;
 reg    [2:0] STAR;   parameter S0=0, S1=1, S2=2, S3=3, S4=4, S5=5, S6=6;

 function [5:0] dotprod;
  input [2:0] x,y;
  input [5:0] s;
  reg [5:0] x1,y1; reg[5:0] p;
  begin
   assign x1={{3{x[2]}},x};
   assign y1={{3{y[2]}},y};
   assign p=x1*y1;
   dotprod = s+p;
  end
 endfunction

 always @(reset_==0) begin DONE<=0; RFD<=1; OUTR<=0; STAR<=S0; end

 always @(posedge clock) if (reset_==1)  #0.1
  casex (STAR)
   S0: begin DONE<=0; RFD<=(dav_==0)?0:1; OUTR<=0; S<=0;         STAR<=(dav_==1)?S0:S1; end
   S1: begin          RFD<=(dav_==0)?0:1; OUTR<=dotprod(X,Y,S); STAR<=(dav_==0)?S1:S2; end
   S2: begin          RFD<=(dav_==0)?0:1; OUTR<=S;              STAR<=(dav_==1)?S2:S3; end
   S3: begin          RFD<=(dav_==0)?0:1; OUTR<=dotprod(X,Y,S); STAR<=(dav_==0)?S3:S4; end
   S4: begin          RFD<=(dav_==0)?0:1; S<=OUTR;              STAR<=(dav_==1)?S4:S5; end
   S5: begin          RFD<=(dav_==0)?0:1; OUTR<=dotprod(X,Y,S); STAR<=(dav_==0)?S5:S6; end
   S6: begin DONE<=1;                                           STAR<=S0; end
  endcase
endmodule
```

Diagramma di Temporizzazione:

I numeri attesi sull'uscita out sono 2*1+3*2+(-4)(-3)=16,(-3*0+(-2)*3+(-1)*(-2)=-4,0*1+1*(-4)+(-1)*(-2)=-6,3*2+(-4)(-3)+(-3)*0=18, ovvero in esadecimale 10, 3C, 3A, 12.

T_{CLOCKC}=10ns, T_{CLOCKP}=4ns



T_{CLOCKC}=10ns, T_{CLOCKP}=12ns