COMPITO di ARCHITETTURA DEI CALCOLATORI del 07-02-2023 (REV.230508) 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: dovrà essere consegnato l'elaborato dell'es.1 come file **<COGNOME>.s** e quelli dell'es. 4 come files **<COGNOME>.v** e **<COGNOME>.png**
NOTA2: per il recupero della prima prova in itinere svolgere gli esercizi 3 e 4; per il recupero della seconda, esercizi 1 e 2.

1) [10/30] Trovare il codice assembly RISC-V corrispondente al seguente micro-benchmark (**utilizzando solo e unicamente istruzioni dalla tabella sottostante**), rispettando le convenzioni di uso dei registri dell'assembly (riportate qua sotto, per riferimento).

```
float A[3][3] ={{1.0,2.0,3.0},{4.0,5.0,6.0},{7.0,8.0,9.0}};

double fun2(float T[3][3], int n) {
   double r = 0;
   int i, j, ii, jj;
   for (i = 0; i < n; ++i) {
      ii = (i + 1) % n;
      for (j = 0; j < n; ++j) {
         jj = (i - 1) % n;
         r += fun1(T[ii][jj], T[i][j]);
         T[i][j] = (float)r;
      }
   }
   return (r);
}

double fun1 (float x, float y) {
   double z = (x + 1) * y + 100;
   return (z);
}

int main() {
   double sum;
   sum = fun2(A, 3);
   print_string("sum=");
   print_double(sum);
   exit(0);
}
```

RISCV Instructions (RV64IMFD) **v221117**

Instruction coding (hexadecimal)			Instruction	Example	Register operation	Meaning (** instructions available only in RV64, i.e. 64-bit case)
funct7/imm	funct3	opcode				
00	0	33/3b	**add**	`add/addw x5,x6,x7`	x5 ← x6 + x7	Add two operands; exception possible (addw**)
20	0	33/3b	**subtract**	`sub/subw x5,x6,x7`	x5 ← x6 – x7	Subtracts two operands; exception possible (subw**)
imm	0	13/1b	**add immediate**	`addi/addiw x5,x6,100`	x5 ← x6 + 100	Add a constant ; exception possible (addiw**)
01	0	33/3b	**multiply**	`mul/mulw x5,x6, x7`	x5 ← x6 * x7	(signed/word) Lower 64 bits of 128-bits product (mulw**)
01	1	33	**multiply high**	`mulh x5,x6,x7`	x5 ← x6 * x7	Higher 64bits of 128-bits product
01	4	33/3b	**division**	`div/divw x5,x6,x7`	x5 ← x6/x7	(signed/word) division (divw**)
01	6	33/3b	**reminder**	`rem/remw x5,x6,x7`	x5 ← x6 % x7	Reminder of the division (remw**)
00	2/3	33	**set on less than**	`slt/sltu x5,x6,x7`	if (x6 < x7) x5 ← 1; else x5 ← 0	signed compare x6 and x7 (less than)
imm	2/3	13	**set on less than immediate**	`slti/sltiu x5,x6,100`	if (x6 < 100) x5← 1; else x5 ← 0	unsigned compare x6 and 100 (less than)
00	7/6/4	33	**and / or / xor**	`and/or/xor x5,x6,x7`	x5← x6&x7 / x6|x7 / x6^ x7	Logical AND/OR/XOR register operand
imm	7/6/4	13	**and /or / xor immediate**	`andi/ori/xori x5,x6,100`	x5 ← x6&100 / x6|100 / x6^100	Logical AND/OR/XOR constant operand
0	1	33/3b	**shift left logical**	`sll/sllw x5,x6,x7`	x5 ← x6 << x7	Shift left by register (sllw**)
imm	1	13/1b	**shift left logical immediate**	`slli/slliw x5,x6,10`	x5 ← x6 << 10	Shift left by the immediate value (slliw**)
0	5	33/3b	**shift right logical**	`srl/srlw x5,x6,x7`	x5 ← x6 >> x7	Shift right by register (srlw**)
imm	5	13/1b	**shift right logical immediate**	`srli/srliw x5,x6,10`	x5 ← x6 >> 10	Shift left by immediate value (srliw**)
20	5	33/3b	**shift right arithmetic**	`sra/sraw x5,x6,x7`	x5 ← x6 >> x7 (arith.)	Shift right by register (sign is preserved) (sraw**)
imm	5	13/1b	**shift right arithmetic immediate**	`srai/sraiw x5,x6,10`	x5 ← x6 >> 10 (arith.)	Shift right by immediate value (sraiw**)
imm	3/2/0	03	**load dword / word / byte**	`ld/lw/lb x5,100(x6)`	x5 ← MEM[x6+100]	Data from memory to register
imm	6/4	03	**load word / byte unsigned**	`lwu/lbu x5,100(x6)`	x5 ← MEM[x6+100]	Data from mem. To reg.; no sign extension (lwu**)
imm	3/2	23	**store dword / word / byte**	`sd/sw/sb x5,100(x6)`	MEM[x6+100] ← x5	Data from register to memory (sw**)
imm[31:12]	-	37	**load upper immediate**	`lui x5,0x12345`	x5 ← 0x1234'5000	Load most significant 20 bits
PSEUDOINSTRUCTION			**load address**	`la x5,var`	x5 ← &var **(PSEUDO INST.)** load address of 'var' in x5	`REAL: lui x5,H20(&var);ori x5, L12(&var) INST.` (H20=high 20 bits of &var; L12=low 12 bits of &var)
imm[31:12] (rd=0) imm[11:0] (rs1=rs2=0)	- 0	6f/63	**jump/branch**	`j/b label`	PC+=off (off=PC-&label) **(PS.INST.)**	`REAL INST.: jal x0,offset/beq x0,x0,offset`
imm[31:12] (rd=1)	-	6f	**jump and link (offset)**	`jal label`	x1←(PC+4); PC+=offset **(PS. INST.)**	`REAL INST.: jal x1,offset` (offset=PC-&label)
Imm (rd=0,rs=1)	0	67	**return from procedure**	`ret`	PC←x1 **(PSEUDO INST.)**	`REAL INST.: jalr x0,0(x1)`
imm	0	67	**jump and link register**	`jalr x1, 100(x5)`	x1 ← (PC + 4); PC=x5+100	Procedure return; indirect call
imm+2	0/1	63	**branch on equal / not-equal**	`beq/bne x5,x6,100`	if (x5 = =/!= x6) PC=PC+100	Equal / Not-equal test; PC relative branch
00 (rs1=0,rs2=0,rd=0)	0	73	**ecall**	`ecall`	SEPC←PC;PC←STVEC;save PL/IE;PL=1;IE=0	Call OS (service number in a7); PL= privilege lev; IE=int.en.
08 (rs1=0,rs2=2,rd=0)	0	73	**sret**	`sret`	PC←SEPC; restore PL/IE	Exit supervisor mode; PL= privilege lev; IE=int.en.
PSEUDOINSTRUCTION			**move**	`mv x5,x6`	x5 ← x6 **(PSEUDO INST.)**	`REAL INST.: add x5,x0,x6`
PSEUDOINSTRUCTION			**load immediate**	`li x5,100`	x5 ← 100 **(PSEUDO INST.)**	`REAL INST.: addi x5,x0,100`
PSEUDOINSTRUCTION			**no operation (nop)**	`nop`	do nothing **(PSEUDO INST.)**	`REAL INST.: addi x0,x0,0`
{0,1} / {4,5}	0	53	**floating point add/sub**	`fadd/fsub.{s,d} f0,f1,f2`	f0←f1+f2 / f0←f1-f2	Single or double precision add / subtract
{8,9} / {c,d}	0	53	**floating point multiplication/division**	`fmul/fdiv.{s,d} f0,f1,f2`	f0←f1*f2 / f0←f1/f2	Single or double precision multiplication / division
PSEUDOINSTRUCTION			**floating point move between f-regs**	`fmv.{s,d} f0,f1`	f0←f1 **(PSEUDO INST.)**	`REAL INST.: fsgnj.{s,d} f0,f1,f1`
PSEUDOINSTRUCTION			**floating point negate**	`fneg.{s,d} f0,f1`	f0← – (f1) **(PSEUDO INST.)**	`REAL INST.: fsgnjn.{s,d} f0,f1,f1`
PSEUDOINSTRUCTION			**floating point absolute value**	`fabs.{s,d} f0,f1`	f0← | f1 | **(PSEUDO INST.)**	`REAL INST.: fsgnjx.{s,d} f0,f1,f1`
{50,51}	0/1/2	53	**floating point compare**	`fle/flt/feq.{s,d} x5,f0,f1`	x5← (f0<f1)	Single and double: compare f0 and f1 <=,<,==
{70,71} (rs2=0)	0	53	**move between x (integer) and f regs**	`fmv.x.{s,d} x5,f0`	x5←f0 (no conversion)	Copy (no conversion)
{78,79} (rs2=0)	0	53	**move between f and x regs**	`fmv.{s,d}.x f0,x5`	f0←x5 (no conversion)	Copy (no conversion)
imm	2	7	**load/store floating point (32bit)**	`flw/fsw f0,0(x5)`	f0←MEM[x5] / MEM[x5]←f0	Data from FP register to memory
imm	3	7	**load/store floating point (64bit)**	`fld/fsd f0,0(x5)`	f0←MEM[x5] / MEM[x5]←f0	Data from FP register to memory
21/20 (rs2=0)	7	53	**convert to/from double from/to single**	`fcvt.d.s/fcvt.s.d f0,f1`	f0← (double)f1 / f0← (single)f1	Type conversion
{60,61} (rs2=0)	7	53	**convert to integer from {single,double}**	`fcvt.w.{s,d} x5,f0`	x5← (int)f0	Type conversion
{68,69} (rs2=0)	7	53	**convert to {single,double} from integer**	`fcvt.{s,d}.w f0,x5`	f0← ({single,double})x5	Type conversion
{2c,2d} (rs2=0)	0	53	**square root**	`fsqrt.{s,d} f0,f1`	f0← square root of f1	Single or double square root
{10,11}	0/1/2	53	**sign injection**	`fsgnj/jn/jx.{s,d} f0,f1,f2`	f0←sgn(f2)|f1| / –sgn(f2)|f1| / sgn(f2)*f1	Extract the mantissa and exp. from f1 and sign from f2

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	---	---

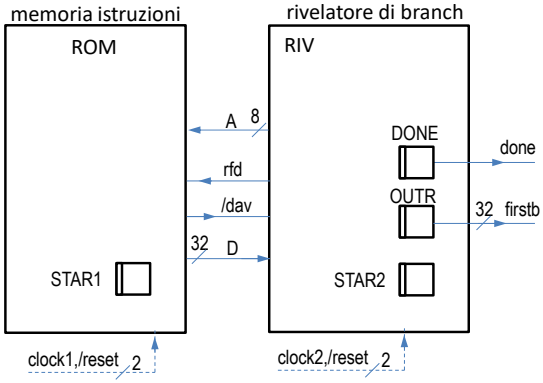
COMPITO di ARCHITETTURA DEI CALCOLATORI del 07-02-2023 (REV.230508)

2) [5/30] In una pipeline RISC-V a 5 stadi, con propagazione (forwarding) abilitata, delay slot, decisioni di salto nello stadio D, individuare le criticità (hazards) del seguente codice assembly (inizialmente t0=0, t1=1024):

```
inizio: add t0, t1, t2; add t0, t0, t1; ld t2, 0(t0); add t0, t1, t2; beq t0, t1, inizio; add t1, t2, t0
```

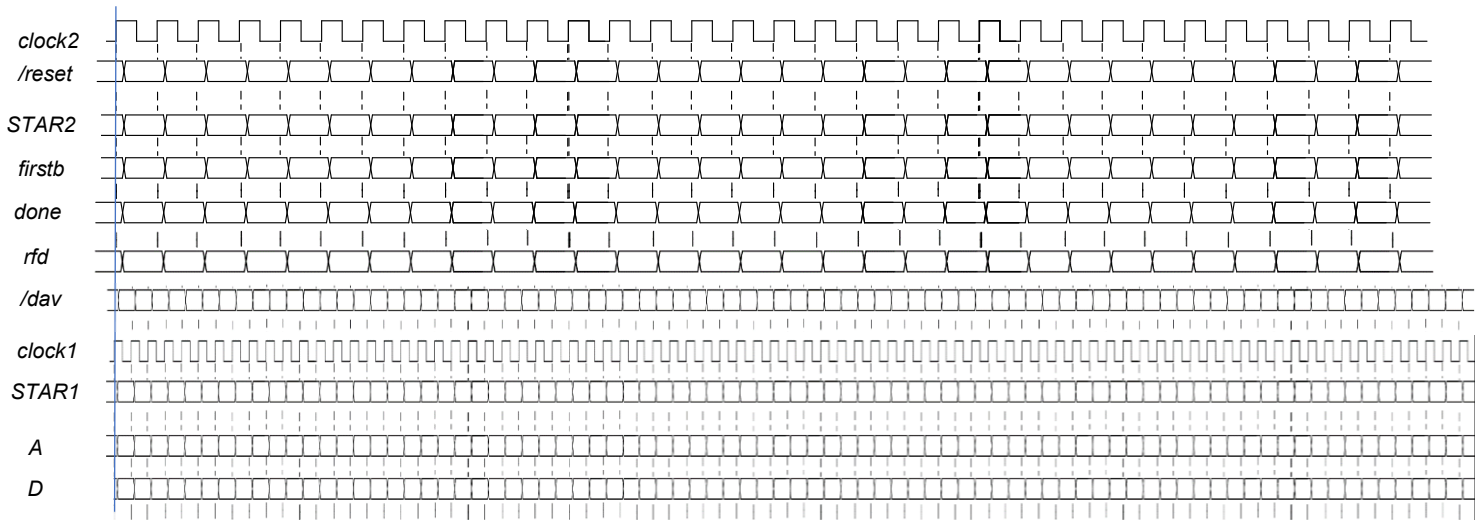
3) [4/30] Descrivere il funzionamento della lettura di un contatore del timer 8254 tramite "read-back command" e i vantaggi dell'uso di tale modalità di lettura.

4) [11/30] Descrivere e sintetizzare in Verilog il modulo **RIV** di figura che funziona nel seguente modo:
specificando un indirizzo **A** (partendo dall'indirizzo 0x00), riceve dal modulo **ROM** una istruzione RISC-V (su 32 bit) sul bus **D** e poi genera l'indirizzo **A** per la successiva istruzione; ogni tre istruzioni ricevute di tipo BEQ o BNE segnala questo fatto mettendo ad 1 per un ciclo l'uscita **done** e fornendo su **firstb** la prima delle tre istruzioni branch riconosciute (per i codici operativi delle istruzioni branch si veda la tabella delle istruzioni RISC-V qui allegata); il colloquio fra il modulo **RIV** e il modulo **ROM** avviene in maniera asincrona tramite i segnali **rfd** e **/dav**; ognuno dei due moduli lavora in maniera localmente sincronizzata come specificato in figura e stabilito dal qui allegato testbench. **Tracciare il diagramma di temporizzazione (punti 5/11)** come verifica della correttezza del modulo realizzato.




```
module testbench;
  reg clock2; initial clock2 =0; always #2 clock2 <=(!clock2);
  reg reset_; initial begin reset_=0; #1 reset_=1;
  #200; $stop; end
  wire[1:0] STAR2=RIV.STAR;
  wire[31:0] firstb; wire done, rfd, dav_;
  reg clock1; initial clock1 =0; always #1 clock1 <=(!clock1);
  wire[1:0] STAR1=ROM.STAR;
  wire[7:0] A; wire[31:0] D;
  SDROM ROM(A,rfd,clock1,reset_,   dav_,D);
  BRDET RIV(D,dav_,clock2,reset_,  rfd,firstb,done,A);
endmodule
```

```
module SDROM(A,rfd,clock,reset_,  dav_,D);
  input[7:0] A; output[31:0] D;
  input rfd,clock,reset_; output dav_;
  reg DAV_; assign dav_=DAV_;
  reg[31:0] QP, DP; assign D=DP;
  reg[11:0] Q1, Q0;
  reg[7:0] OPCODE;
  reg G;
  reg[1:0] STAR; parameter S0=0, S1=1, S2=2, IB=7'b1100011;
  always @(reset_==0) begin G<=0; STAR<=S0; Q0<=76; end
  always @(posedge clock) if (reset_==1) #0.1
    casex (STAR)
      S0: begin if (G==0) begin Q1=263*Q0+71; Q0=Q1; G=1;
                if (Q1[4]==1) OPCODE=IB; else OPCODE=Q1[7:0];
                QP={Q1[11:4],5'b0,5'b0,3'b0,Q1[3:0],OPCODE};end
                DAV_=1; STAR<=(rfd==1)?S1:S0; end
      S1: begin DP=QP; G=0; DAV_=0; STAR<=S2; end
      S2: begin STAR<=(rfd==1)?S2:S0;end
    endcase
endmodule
```



COMPITO di ARCHITETTURA DEI CALCOLATORI del 07-02-2023 (REV.230508)

SOLUZIONE

ESERCIZIO 1 (si assume che 'int' sia un intero a 64 bit)

```
.data
A: .float 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
st1: .asciz "sum="

.globl main
.text

fun1:
# fa0=x fa1=y fa0(return)=z
   li       t0,1        #(int)1
   fcvt.s.w ft0,t0      #(float)1
   fadd.s   ft0,ft0,fa0 # (x+1)
   fmul.s   ft0,ft0,fa1 # (.)*y
   li       t0,100      #(int)100
   fcvt.s.w ft1,t0      #(float)100
   fadd.s   ft0,ft0,ft1 #(.)+100
   fcvt.d.s fa0,ft0     #(double)
   ret

fun2:
# s0=a0=&T s1=a1=n fa0=fs0=r s2=i s3=j
# s4=ii s5=12 s6=&T[i][j] t0=jj
   addi sp,sp,-72# allocate frame
   sd   s0, 0(sp)
   sd   s1, 8(sp)
   sd   s2,16(sp)
   sd   s3,24(sp)
   sd   s4,32(sp)
   sd   s5,40(sp)
   sd   s6,48(sp)
   fsd  fs0,56(sp)
   sd   ra,64(sp)
   mv   s0,a0    # save a0 (&T)
   mv   s1,a1    # save a1 (n)
   li   s5,12    # save T row length
   fmv.d.x fs0,x0# r=0.0
   li   s2,0     # i=0
f2_for1_start:
   slt  t1,s2,a1 # i <? n
   beq  t1,x0,f2_for1_end
        addi s4,s2,1  # ii=(i+1)
        rem  s4,s4,s1 # ii=(.) % n
        li   s3,0     # j=0
   f2_for2_start:
      slt  t1,s3,a1 # j<?n
      beq  t1,x0,f2_for2_end

      addi t0,s2,-1  # jj=(i-1)
      rem  t0,t0,s1 # jj=(.) % n

      mul  t2,s4,s5 # ii*12
      slli t3,t0,2  # jj*4
      add  t2,t2,t3 # byte offset
      add  t2,s0,t2 # &T[ii][jj]
      flw  fa0,0(t2)# T[ii][jj]

      mul  t2,s2,s5 # i*12
      slli t3,s3,2  # j*4
      add  t2,t2,t3 # byte offset
      add  t2,s0,t2 # &T[i][j]
      flw  fa1,0(t2)# T[i][j]
      mv   s6,t2    # save &T[i][j]

      call fun1     # returns a double

      fadd.d fs0,fs0,fa0# r+=(.)
      fcvt.s.d ft0,fs0  #(float)
      fsw  ft0,0(s6)    # T[i][j]=(.)

      addi s3,s3,1  # ++j
      b    f2_for2_start
   f2_for2_end:
      addi s2,s2,1  # ++i
      b    f2_for1_start
f2_for1_end:

   fmv.d fa0,fs0  #return value (r)
   ld   s0, 0(sp)
   ld   s1, 8(sp)
   ld   s2,16(sp)
   ld   s3,24(sp)
   ld   s4,32(sp)
   ld   s5,40(sp)
        ld   s6,48(sp)
   fld  fs0,56(sp)
   ld   ra,64(sp)
   addi sp,sp,+72 # free frame
   ret

main:
   la   a0,A     # a0=&A
   li   a1,3     # a1=3
   call fun2     # result in fa0
   la   a0,st1   # a0=&st1
   li   a7,4
   ecall         # print "sum="
   li   a7,3
   ecall         # print result (fa0)
   li   a7,10    # exit(0)
   ecall
```

```
Run I/O
sum=6156.0
-- program is finished running (0) --
```

ESERCIZIO 2

In questo codice sono presenti diverse dipendenze fra i registri t0, t1 e t2, ma considerando le ipotesi dell'esercizio solo alcune generano effettivamente delle criticità, in particolare:

1) L'istruzione "LD" mette in t2 il valore del dato letto all'indirizzo 0(t0) soltanto nello stadio M: risulta pertanto impossibile propagare tale valore letto all'istruzione successiva che dovrebbe utilizzarlo nello stadio X per effettuare la ADD;
2) L'istruzione BEQ deve utilizzare il valore contenuto di t0 nello stadio D (utilizzando il comparatore a valle dei registri, per le ipotesi date), ma tale valore deve essere ancora prodotto dalla precedente ADD nello stadio X e quindi non risulta possibile propagarlo.

Complessivamente esistono quindi le due precedenti criticità di tipo RAW di cui la prima denominata "load-use" e la seconda dovuta al comparatore nello stadio D. Nel diagramma di esecuzione seguente sono evidenziati gli stalli (simbolo "-"). Nota: nel seguente diagramma le prime due istruzioni realizzano nel codice (le due ipotesi iniziali t1=1024, t0=0 – non fanno parte della soluzione). Questo codice può essere verificato anche sul simulatore di pipeline https://webriscv.dii.unisi.it

EXECUTION TABLE

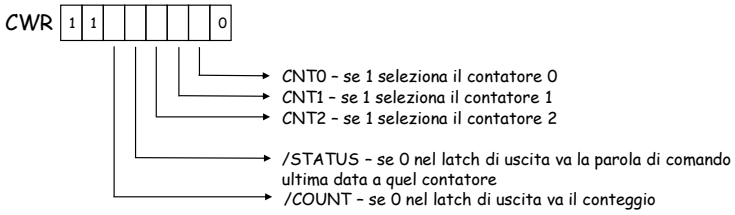
FULL LOOPS	CPU Cycles													
Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
addi t1, x0, 1024	F	D	X	M	W										
add t2, x0, x0		F	D	X	M	W									
add t0, t1, t2			F	D	X	M	W								
lw t2, 0(t0)				F	D	X	M	W							
add t0, t1, t2					F	-	D	X	M	W					
beq t0, t1, -24							F	-	D	X	M	W			
add t1, t2, t0									F	D	X	M	W		
add t0, t1, t2										F	D	X	M	W	
lw t2, 0(t0)											F	D	X	M	W

ESERCIZIO 3

Lettura del conteggio (2)

- **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 più 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**

COMPITO di ARCHITETTURA DEI CALCOLATORI del 07-02-2023 (REV.230508)

SOLUZIONE

ESERCIZIO 4

Si può adottare uno schema produttore-consumatore esssendo il seguente modulo il “consumatore”. Questa è una possibile soluzione.

```
module BRDET(D,dav_,clock,reset_,  rfd,FIRSTB,done,A);
  output[31:0] FIRSTB; input[31:0] D; output[7:0] A;
  input dav_,clock, reset_; output rfd,done;
  reg[31:0] OUTR; assign FIRSTB=OUTR; reg[7:0] A1; assign A=A1;
  reg RFD,DONE; assign rfd=RFD, done=DONE;
  reg[1:0] STAR; reg G; reg[1:0] COUNT;
  parameter S0=0,S1=1,S2=2, IB=7'b1100011;
  always @(reset_==0) begin STAR<=S0; G<=0; A1<=0; DONE<=0; OUTR<=0;end
  always @(posedge clock) if (reset_==1) #0.1
    casex (STAR)
      S0: begin if (G==0) begin COUNT<=0; G=1; end
                RFD=1; STAR<=(dav_==0)?S1:S0; end
      S1: begin if (D[6:0] == IB) begin COUNT=COUNT+1; if (COUNT==1) OUTR=D; end
                if (COUNT==3) begin DONE=1; G=0; end
                RFD=0; STAR<=S2; end
      S2: begin A1<=(dav_==1)?A1+4:A1; DONE<=0; STAR<=(dav_==0)?S2:S0; end
    endcase
endmodule
```