COMPITINO di ARCHITETTURA DEI CALCOLATORI del 14-01-2020

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).

```
union header {
    struct {
        union header *ptr;
        unsigned size;
    } s;
};
typedef union header Header;

static Header base;
static Heade
r *freep = NULL;


void *malloc(unsigned nbytes)
{
    Header *p, *prevp;
    unsigned nunits;
    nunits=(nbytes+sizeof(Header)-1)/sizeof(Header)+1;
    if((prevp = freep) == NULL) {
        base.s.ptr = freep = prevp = &base;
        base.s.size = 0;
    }
    for(p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if(p->s.size >= nunits) {
            if(p->s.size == nunits)
                prevp->s.ptr = p->s.ptr;
            else {
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void *)(p + 1);
        }
        if(p == freep)
            if((p = sbrk(nunits)) == NULL)
                return NULL;
    }
}
```

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

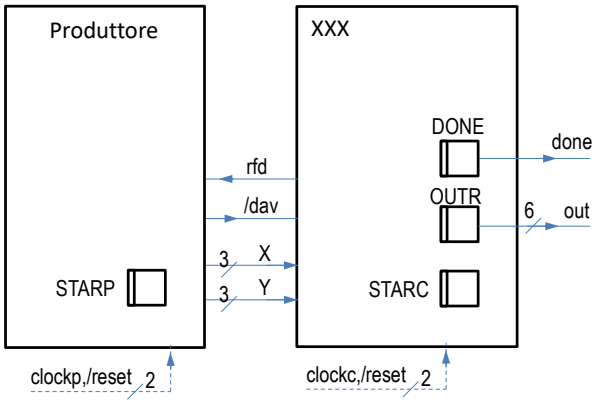
Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Arguments
read_float	6	---	fa0=float
read_double	7	---	fa0=double
read_string	8	a0=address of input buffer, a1=max chars to read	---
sbrk	9	a0=Number of bytes to be allocated	a0=pointer to allocated memory
exit	10	---	---

COMPITINO di ARCHITETTURA DEI CALCOLATORI del 14-01-2020

2) [5/30] Si consideri una cache di dimensione 128B e a 4 vie di tipo write-back. La dimensione del blocco e' 32 byte, il tempo di accesso alla cache e' 4 ns e la penalita' in caso di miss e' pari a 40 ns, la politica di rimpiazzamento e' FIFO. Il processore effettua i seguenti accessi in cache, ad indirizzi al byte: 99, 104, 140, 118, 112, 197, 178, 112, 250, 176, 125, 223, 133, 277, 256, 212, 163, 174, 184. Tali accessi sono alternativamente letture e scritture. Per la sequenza data, ricavare il tempo medio di accesso alla cache, riportare i tag contenuti in cache al termine e la lista dei blocchi (ovvero il loro indirizzo) via via eliminati durante il rimpiazzamento ed inoltre in corrispondenza di quale riferimento il blocco e' eliminato.

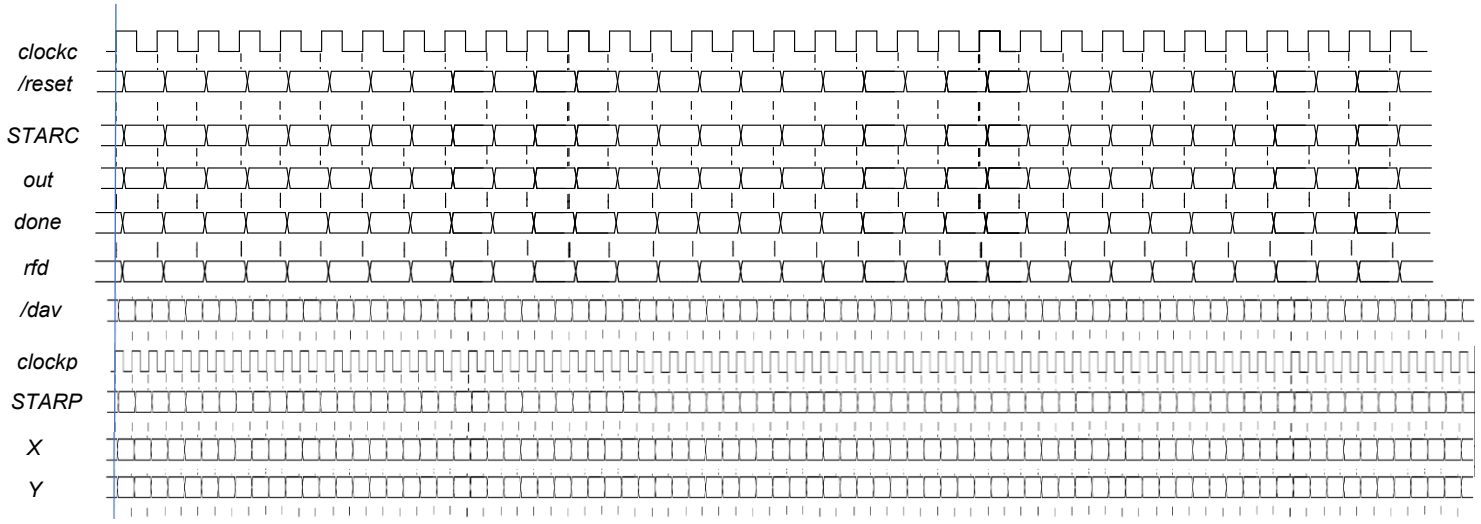
3) [4/30] Descrivere l'algoritmo CSMA/CD per l'accesso al mezzo trasmissivo nello standard Ethernet e il relativo algoritmo di back off.

4) [9/30] 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 14-01-2020

SOLUZIONE

ESERCIZIO 1

```
.data
base: .space 8
freep: .word 0
RTN: .asciz "\n"
HEAPtop: .asciz "Heap Top:"
heapmsg: .asciz "Malloc returns:"
.globl main
.text
malloc:
#____NO_CALL_FRAME_____________
S1: # a0=nbytes, sizeof(Header)=8
   addi t0, a0, 8         # nbytes +8
   addi t0, t0, -1        # (nbytes +8 -1)
   srli t0,t0, 3          # (.)/8
   addi t0, t0, 1         # (.)/8+1
   # t0=nunits
#---------------------------------------
S2:
   la t1,  freep          # &freep
   lw t2, 0(t1)           # VALORE_DI(freep)
   # t2 = prevp     # (prevp=freep)
   bne t2, x0 FINEIF1     # if((.)==NULL)
   la t3,  base           # &base
   add t2, x0, t3         # prevp=&base
   sw t3, 0(t1)           # freep=&base
   sw t3, 0(t3)           # base.s.ptr=&base
   sw x0, 4(t3)           # base.s.size=0

FINEIF1:
#---------------------------------------
S3:
#---------------------------------------
S31: # t4=p
    lw t4, 0(t2)          # p=prevp->s.ptr
#---------------------------------------
S3.2: # VUOTO!
#---------------------------------------
S3.3:
INIFOR:
S331:
   # (t0=nunits), (t4=p)
   lw t5, 4(t4)          # p->s.size
   # t5 = p->s.size
   slt t6, t5, t0        # A>=B => !(A<B)
   bne t6, x0 FINEIF2 # if (!(p->s.size<nunits))
CORPOIF2:
   bne t5, t0 ELSE3    # if (p->s.size==nunits)
   lw a6, 0(t4)        # (p->s.ptr)
   sw a6, 0(t2)        # prevp->s.ptr=(.)
   j FINEIF3
ELSE3:
   sub t5, t5, t0      # (p->s.size-nunits)
   sw t5, 4(t4)        # p->size=(.)
   add t4, t4, t5      # p+=p->s.size
   sw t0, 4(t4)        # p->s.size=nunits
FINEIF3:
   sw t2, 0(t1)        # freep=prevp
   addi a0, t4, 8      # p+1, il "+1" e' un Header
   # a0 e' il valore di ritorno di "malloc"
   j FINEFUN
FINEIF2: #chiudo S3.3.1
#---------------------------------------
S3.3.2:
   lw t3, 0(t1)        # freep aggiornato
   bne t4, t3 FINEIF4 # if (p==freep)
   #addi a0, x0, 9
   add a0, t0, x0      # a0=units da allocare
      # NOTA: IL SEGUENTE CODICE E' ADATTATO
      # AL FINE DI AVERE UNA VERISONE FUNZIONANTE
      slli a0,a0,3     # sbrk(nunits*sizeof(Header))
   addi a7,x0,9        # servizio 9 => SBRK
   ecall               # up=a0=pun.mem.allocata
   add  x26,$0,0
      addi x26,$0,-1   # x26 = -1
   beq  a0,x26,FINENULL # if((.) == -1) ADATTATO!
      sw t0, 4(a0)     # up->s.size=nunits
      sw t4, 0(a0)     # up->s.ptr = p
      sw a0, 0(t4)     # p->s.ptr = up
      add t4, t3, x0   # p = freep
   j FINEIF5
FINENULL:
   add a0, x0, x0      # a0 <= 0
   j FINEFUN
FINEIF5:
FINEIF4: # chiude S332 e CORPOFOR
   add t2, x0, t4      # prevp=p
   lw t4, 0(t4)        # p=p->s.ptr
   j INIFOR            # chiudo FOR, S3.3.2 e S3.3
#---------------------------------------
FINEFUN: jr ra
#---------------------------------------
#TESTBENCH
printHEAPtop:
   add a0, x0, x0      # get HEAPtop
   addi a7,x0,9        # serv.9
   ecall               #sbrk
   add a1, x0, a0      # salvo in a1
   la a0,  HEAPtop     # stampa msg
   addi a7,x0,4        # serv.4
   ecall               #print_str
   add a0, x0, a1      # nuovo HEAPtop
   addi a7,x0,1        #serv. 1
   ecall                #print_int
   la a0,  RTN         # stampa RTN
   addi a7,x0,4        #serv 4
   ecall               #print_str
   jr ra
printMALLOC:
   #a0 = pun. da stampare
   add t0, a0, x0      # t0=pun.
   la a0,  heapmsg
   addi a7,x0,4        # serv.4
   ecall               #print_str
   add a0, t0, x0      # a0=pun
   addi a7,x0,1        #serv.1
   ecall               #print_int
   la a0,  RTN         # stampa RTN
   addi a7,x0,4        # serv.4
   ecall               # print_str
   jr ra
#-------------------------------------
main:
   jal printHEAPtop
   addi a0, x0, 256    # alloca 256B
   jal malloc
   jal printMALLOC
   jal printHEAPtop
   addi a0, x0, 256    # alloca 256B
   jal malloc
   jal printMALLOC
   jal printHEAPtop
   addi a0, x0, 256    # alloca 256B
   jal malloc
   jal printMALLOC
   jal printHEAPtop
   addi a7,x0,10 #serv.10
   ecall         #exit
```

OUTPUT:

```
Heap Top:268697600
Malloc returns:268697608
Heap Top:268697864
Malloc returns:268697872
Heap Top:268698128
Malloc returns:268698136
Heap Top:268698392
```

ESERCIZIO 2

Sia X il generico riferimento, A=associativita', B=dimensione del blocco, C=capacita' della cache.
Si ricava S=C/B/A=# di set della cache=128/32/4, XM=X/B, XS=XM%S, XT=XM/S.
A=4, B=32, C=128, RP=FIFO, Thit=4, Tpen=40, 19 references:

	T	X	XM	XT	XS	XB	H	[SET]:USAGE	[SET]:MODIF	[SET]:TAG
===	R	99	3	3	0	3	0	[0]:3,0,0,0	[0]:0,0,0,0	[0]:3,-,-,-
===	W	104	3	3	0	8	1	[0]:3,0,0,0	[0]:1,0,0,0	[0]:3,-,-,-
===	R	140	4	4	0	12	0	[0]:2,3,0,0	[0]:1,0,0,0	[0]:3,4,-,-
===	W	118	3	3	0	22	1	[0]:2,3,0,0	[0]:1,0,0,0	[0]:3,4,-,-
===	R	112	3	3	0	16	1	[0]:2,3,0,0	[0]:1,0,0,0	[0]:3,4,-,-
===	W	197	6	6	0	5	0	[0]:1,2,3,0	[0]:1,0,0,0	[0]:3,4,6,-
===	R	178	5	5	0	18	0	[0]:0,1,2,3	[0]:1,0,0,0	[0]:3,4,6,5
===	W	112	3	3	0	16	1	[0]:0,1,2,3	[0]:1,0,0,0	[0]:3,4,6,5
===	R	250	7	7	0	26	0	[0]:3,0,1,2	[0]:0,0,0,0	[0]:7,4,6,5
===	W	176	5	5	0	16	1	[0]:3,0,1,2	[0]:0,0,0,1	[0]:7,4,6,5
===	R	125	3	3	0	29	0	[0]:2,3,0,1	[0]:0,0,0,1	[0]:7,3,6,5
===	W	223	6	6	0	31	1	[0]:2,3,0,1	[0]:0,0,1,1	[0]:7,3,6,5
===	R	133	4	4	0	5	0	[0]:1,2,3,0	[0]:0,0,0,1	[0]:7,3,4,5
===	W	277	8	8	0	21	0	[0]:0,1,2,3	[0]:0,0,0,0	[0]:7,3,4,8
===	R	256	8	8	0	0	1	[0]:0,1,2,3	[0]:0,0,0,0	[0]:7,3,4,8
===	W	212	6	6	0	20	0	[0]:3,0,1,2	[0]:0,0,0,0	[0]:6,3,4,8
===	R	163	5	5	0	3	0	[0]:2,3,0,1	[0]:0,0,0,0	[0]:6,5,4,8
===	W	174	5	5	0	14	1	[0]:2,3,0,1	[0]:0,1,0,0	[0]:6,5,4,8
===	R	184	5	5	0	24	1	[0]:2,3,0,1	[0]:0,1,0,0	[0]:6,5,4,8

LISTA BLOCCHI USCENTI:

(out: XM=3 XT=3 XS=0)
(out: XM=4 XT=4 XS=0)
(out: XM=6 XT=6 XS=0)
(out: XM=5 XT=5 XS=0)
(out: XM=7 XT=7 XS=0)
(out: XM=3 XT=3 XS=0)

CONTENUTI dell'unico SET al termine:

P1 Nmiss=10 Nhit=9 Nref=19 mrate≈0.526316 AMAT=th+mrate*tpen≈25.0526

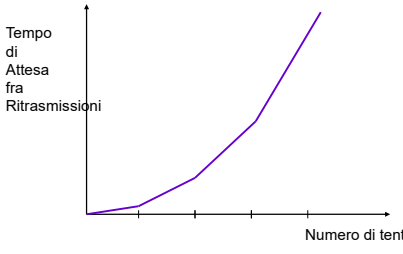
ESERCIZIO 3

CSMA/CD, Carrier Sense Multiple Access with Collision Detection

- Un nodo "ascolta prima di parlare" (carrier sense)
- Se il mezzo e' libero si attende un tempo "Defer Time"
- Se ci sono collisioni (multiple access) l'energia sul mezzo aumenta
- Un nodo cerca di rilevare collisioni durante la trasmissione (collision detect)
- Se un nodo rileva una collisione, continua a trasmettere 4-6 bytes per essere sicuro che la collisione venga rilevata dagli altri nodi e poi "lascia" il mezzo
- Se un nodo ha rilevato collisione, la trasmissione viene ripetuta dopo R*Δt per altri 15 tentativi
- Dopo 16 tentativi l'errore viene segnalato ai livelli superiori
- Δt e' fisso mentre R viene calcolato secondo un "algoritmo di back off"

Algoritmo di back off per il calcolo di R

- Sia n l'n-esimo tentativo di trasmissione (n=1,2,…, 15)
- K= min(n,10)
- R e' un numero casuale t.c. 0 <= R <= 2^K



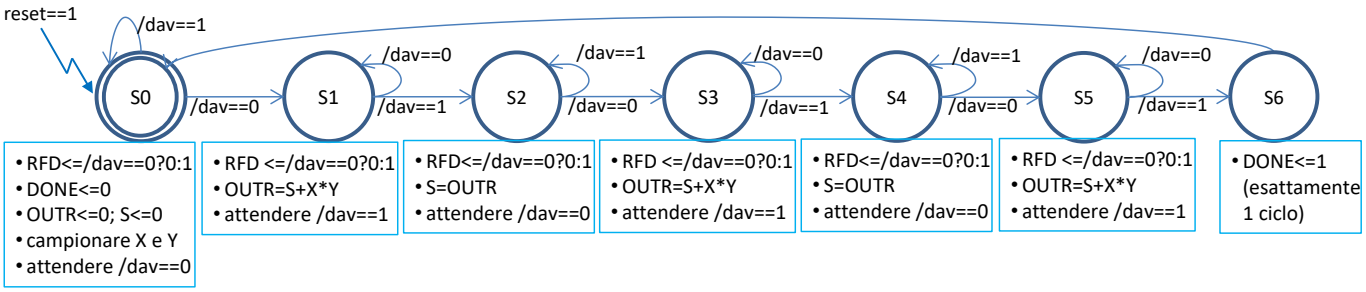
Risolve in maniera probablisticamente accettabile i conflitti fino a 1024 nodi…

COMPITINO di ARCHITETTURA DEI CALCOLATORI del 14-01-2020

SOLUZIONE

ESERCIZIO 4

Un possibile diagramma degli stati:



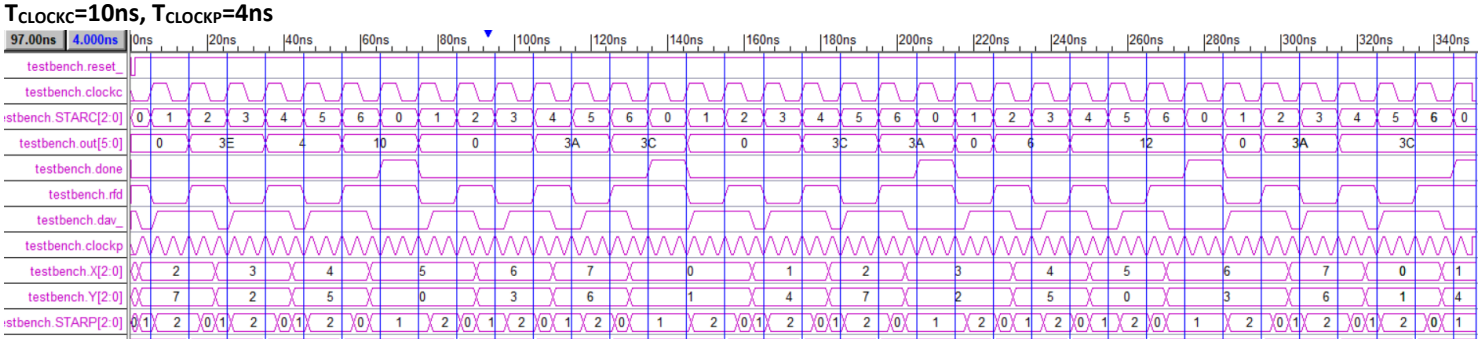
Codice Verilog del modulo da realizzare

```
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<=OUTR;           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