

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

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 compute_frobenius_norm(float *matrix, int n) {
    float sum = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            float elem = *(matrix + i * n + j);
            sum += elem * elem;
        }
    }
    return sqrt(sum);
}
```

Nota: 'int' è un intero a 64 bit.

```
int main() {
    float *matrix = (float *)sbrk(9 * sizeof(float));

    for (int i = 0; i < 3; ++i)
        for (int j = 0; j < 3; ++j)
            *(matrix + i * 3 + j) = i * 3 + j + 3;

    float norm = compute_frobenius_norm(matrix, 3);
    print_string("Frobenius norm: ");
    print_float(norm);
    exit(0);
}
```

RISCV Instructions (RV64IMFD)

v230703

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	remainder	rem/remw x5,x6,x7	x5 ← x6 % x7	Remainder 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	sraiw/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 (lbu**)
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 ← 0x12345000	Load most significant 20 bits
PSEUDOINSTRUCTION			load address	la x5,var	x5 ← &var (PSEUDO INST.) load address of 'var' in x5	REAL INST.: 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)	-	61/63	jump/branch	j/b label	PC+=off (off=PC-&label) (PS.INST.)	REAL INST.: jal x0,offset/beq x0,x0,offset
imm[11:0](rs1=rs2=0)	-	6f	jump and link (offset)	jal label	x1 ← (PC+4); PC+=offset (PS. INST.)	REAL INST.: jal x1,offset (offset=PC-&label)
imm[31:12](rd=1)	-	6f	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 PLIE; 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

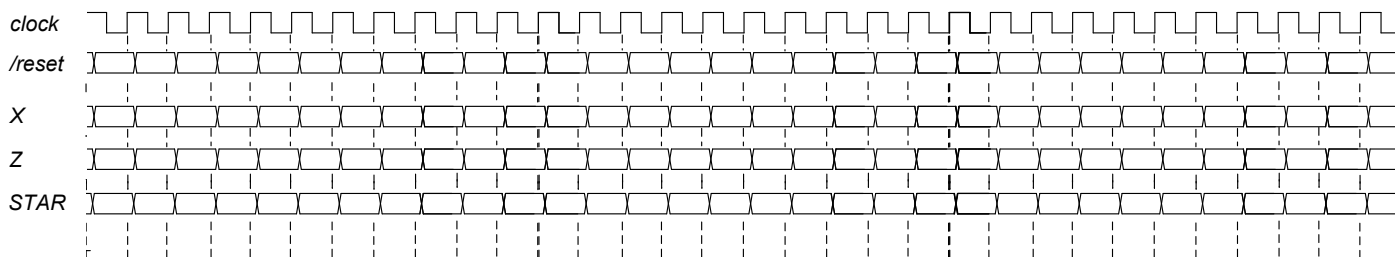
Register	ABI Name	Usage
f10-f11	fa0-fa1	Argument and Return values
f8-f9, f18-f27	fs0-fs1, fs2-fs11	Saved registers
f0 - f7, f28-f31	ft0-ft7, ft8-ft11	Temporaries registers
f12-17	fa2-fa7	Function arguments

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(0)	10	---	---

- 2) [5/30] Si consideri una cache di dimensione 64B e a 2 vie di tipo write-back/write-non-allocate. La dimensione del blocco e' 8 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' LRU. Il processore effettua i seguenti accessi in cache, ad indirizzi al byte: 3433, 3454, 3425, 3454, 3422, 3454, 3639, 3726, 3854, 3424, 3433, 3454, 3425, 3454, 3422. 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, i bit di modifica (se presenti) 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) [5/30] Descrivere l'algoritmo CSMA/CD per l'accesso al mezzo trasmissivo nello standard Ethernet e il relativo algoritmo di back off. Inoltre, Disegnare un grafico qualitativo che mette in relazione il numero di tentativi con il tempo di attesa fra le ritrasmissioni.
- 4) [10/30] Descrivere e sintetizzare in Verilog una rete sequenziale utilizzando il modello di Moore con un ingresso X su due bit e una uscita Z su un bit che funziona nel seguente modo: devono essere riconosciute le sequenze non-interallacciate 11,10,00,01,11; l'uscita Z va a 1 (per 1 ciclo di clock) se è presente tale sequenza. Gli stimoli di ingresso sono dati dal seguente modulo Verilog Testbench [11/30 di cui 4/30 per il diagramma di temporizzazione corretto].

Tracciare il diagramma di temporizzazione [4/10 punti] corrispondente alla esecuzione del modulo Verilog realizzato, come verifica della correttezza dell'unità. Nota: si può 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. Modello del diagramma temporale da tracciare:



```

module TopLevel;
  reg reset_;initial begin reset_=0; #22 reset_=1; #300; $stop; end
  reg clock_;initial clock_=0; always #5 clock_ <=(!clock_);
  reg[1:0] X;
  wire Z;
  wire[2:0] STAR=Xxx.STAR;
  initial begin X=0;
  wait(reset_==1); #5
  @(posedge clock_); X<=0; @(posedge clock_); X<=3; @(posedge clock_); X<=2;
  @(posedge clock_); X<=0; @(posedge clock_); X<=1; @(posedge clock_); X<=3; @(posedge clock_); X<=1;
  @(posedge clock_); X<=1; @(posedge clock_); X<=0; @(posedge clock_); X<=0; @(posedge clock_); X<=3;
  @(posedge clock_); X<=2; @(posedge clock_); X<=1; @(posedge clock_); X<=3; @(posedge clock_); X<=2;
  @(posedge clock_); X<=0; @(posedge clock_); X<=1; @(posedge clock_); X<=3; @(posedge clock_); X<=0;
  $finish;
  end
  XXX Xxx(X,Z,clock_,reset_);
endmodule

```

SOLUZIONE

ESERCIZIO 1

```
.data
norm_msg: .asciz "Frobenius norm: "
.text
.globl main
# -----
# Args: a0 = &matrix, a1 = size of matrix
compute_frobenius_norm:
li t0, 0 # t0: sum = 0
li t1, 0 # t1: i index

loop_i:
li t2, 0 # t2: j index
loop_j:
mul t5, t1, a1 # t5 = i * SIZE
add t5, t5, t2 # t5 = i * SIZE + j
slli t5, t5, 3 # t5=(.)*8 (dword offset)
add t5, t5, a0 # t5 = &matrix[i][j]
ld t6, 0(t5) # t6 = matrix[i][j]
mul t6, t6, t6 # t6 = elem * elem
add t0, t0, t6 # sum += elem * elem
addi t2, t2, 1 # j++
bne t2, a1, loop_j # loop_j if j!=SIZE
addi t1, t1, 1 # i++
bne t1, a1, loop_i # loop_i if i!=SIZE

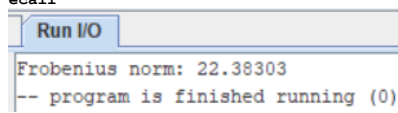
fcvt.s.w fa0, t0
fsqrt.s fa0, fa0 # Compute sqrt(sum)
ret

main:
li a0, 72 #3*3 matrix, each element 8B (72B)
li a7, 9 # ecall service 9 for sbrk
ecall # Allocate memory, address in a0

# Initialize the matrix
li t1, 0 # i = 0
li t4, 3 # SIZE = 3
m_loop_i:
li t2, 0 # j = 0
m_loop_j:
mul t5, t1, t4 # t5 = i * SIZE
add t5, t5, t2 # t5 = i * SIZE + j
slli t5, t5, 3 # t5=(.)*8 (dword offset)
add t5, t5, a0 # t5 = &matrix[i][j]
mul t6, t1, t4 # t6 = i * SIZE
add t6, t6, t2 # t6 = i * SIZE + j
addi t6, t6, 3 # t6 = i * SIZE + j + 3
sd t6, 0(t5) # matrix[i][j]=i*3+j+3

addi t2, t2, 1 # j++
bne t2, t4, m_loop_j # loop_j if j!=SIZE
addi t1, t1, 1 # i++
bne t1, t4, m_loop_i # loop_i if i!=SIZE

la a0, norm_msg # Load address of message
li a7, 4 # service 4 to print msg
ecall
li a7, 2 # service 2 to print fa0
ecall # make ecall
li a7, 10 # service 10 for exit
ecall
```



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=64/8/2, XM=X/B, XS=XM*S, XT=XM/S. A=2, B=8, C=64, RP=LRU, Thit=4, Tpen=40, 15 references:

===	T	X	XM	XT	XS	XB	H	[SET]:USAGE	[SET]:MODIF	[SET]:TAG
===	R	3433	429	107	1	1	0	[1]:1,0	[1]:0,0	[1]:107,-
===	W	3454	431	107	3	6	0	[3]:1,0	[3]:0,0	[3]:107,-
===	R	3425	428	107	0	1	0	[0]:1,0	[0]:0,0	[0]:107,-
===	W	3454	431	107	3	6	1	[3]:1,0	[3]:1,0	[3]:107,-
===	R	3422	427	106	3	6	0	[3]:0,1	[3]:1,0	[3]:107,106
===	W	3454	431	107	3	6	1	[3]:1,0	[3]:1,0	[3]:107,106
===	R	3639	454	113	2	7	0	[2]:1,0	[2]:0,0	[2]:113,-
===	W	3726	465	116	1	6	0	[1]:0,1	[1]:0,0	[1]:107,116
===	R	3854	481	120	1	6	0	[1]:1,0	[1]:0,0	[1]:120,116
===	W	3424	428	107	0	0	1	[0]:1,0	[0]:1,0	[0]:107,-
===	R	3433	429	107	1	1	0	[1]:0,1	[1]:0,0	[1]:120,107
===	W	3454	431	107	3	6	1	[3]:1,0	[3]:1,0	[3]:107,106
===	R	3425	428	107	0	1	1	[0]:1,0	[0]:1,0	[0]:107,-
===	W	3454	431	107	3	6	1	[3]:1,0	[3]:1,0	[3]:107,106
===	R	3422	427	106	3	6	1	[3]:0,1	[3]:1,0	[3]:107,106

LISTA BLOCCHI USCENTI:

(out: XM=429 XT=107 XS=1)
(out: XM=465 XT=116 XS=1)

CONTENUTI dei SET al termine

P1 Nmiss=8 Nhit=7 Nref=15 mrate=0.533333 AMAT=th+mrate*tpen=25.3333

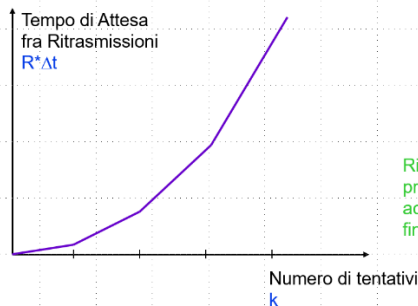
ESERCIZIO 3

CSMA/CD, Carrier Sense Multiple Access with Collision Detection

- Un nodo "ascolta prima di parlare" (carrier sense)
- Se il mezzo è 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 $R \cdot \Delta t$ per altri 15 tentativi
- Dopo 16 tentativi l'errore viene segnalato ai livelli superiori
- Δt è 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 è un numero casuale t.c. $0 \leq R \leq 2^K$

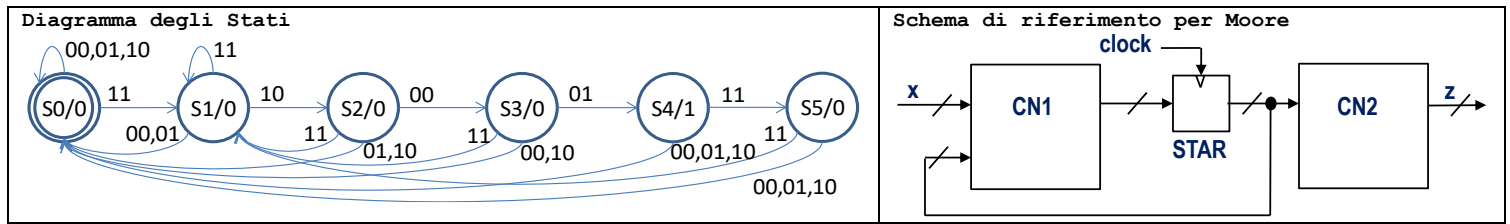


Risolve in maniera probabilisticamente accettabile i conflitti fino a 1024 nodi...

SOLUZIONE

ESERCIZIO 4

In corrispondenza del pattern $X_{t-4}, X_{t-3}, X_{t-2}, X_{t-1}, X_t = 11, 10, 00, 01, 11$ NON INTERALLACCIATO ottengo $\rightarrow Z_{t+1} = 1$; (ricordare che e' richiesto Moore).



Codice Verilog del modulo da realizzare (possibile soluzione con Moore):

```

module XXX(X,z,clock,reset_);
  input[1:0] X;
  input clock,reset_;
  output z;
  reg[2:0] STAR;
  parameter S0=0,S1=1,S2=2,S3=3,S4=4,S5=5;

  always @(reset_==0) #1 begin STAR<=S0; end
  assign z=(STAR==S5)?1:0;

  always @(posedge clock) if(reset_==1) #3
  casex (STAR)
    S0: begin STAR<=(X==2'b11)?S1:S0; end
    S1: begin STAR<=(X==2'b10)?S2:(X==2'b11)?S1:S0; end
    S2: begin STAR<=(X==2'b00)?S3:(X==2'b11)?S1:S0; end
    S3: begin STAR<=(X==2'b01)?S4:(X==2'b11)?S1:S0; end
    S4: begin STAR<=(X==2'b11)?S5:S0; end
    S5: begin STAR<=(X==2'b11)?S1:S0; end
  endcase
endmodule
    
```

Diagramma di Temporizzazione:

