COMPITO di ARCHITETTURA DEI CALCOLATORI del 16-11-2023

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: 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).

```
int main() {
    print_float(sin_taylor(1.0, 5));
    exit(0);
}
int fact(int n) {
    int fact = 1;
    for (int i = 1; i <= n; i++) fact *= i;
    return fact;
}
float power(float x, int n) {
    float result = 1.0;
    for (int i = 0; i < n; i++) result *= x;
    return result;
}
```

Nota: 'int' è un intero a 64 bit.

```
float sin_taylor(float x, int terms) {
    float result = 0.0;
    int sign = 1;
    int pwridx = 1;

    for (int i = 1; i <= terms; i++) {
        result += sign * power(x, pwridx) / fact(pwridx);
        sign *= -1;
        pwridx += 2;
    }
    return result;
}
```

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

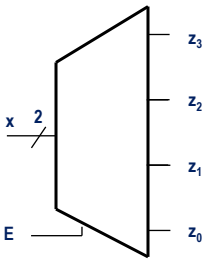
Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Arguments
read_float	6	---	fa0=float
read_double	7	---	fa0=double
read_string	8	a0=address of input buffer, a1=max chars to read	---
sbrk	9	a0=Number of bytes to be allocated	a0=pointer to allocated memory
exit	10	---	---

COMPITO di ARCHITETTURA DEI CALCOLATORI del 16-11-2023

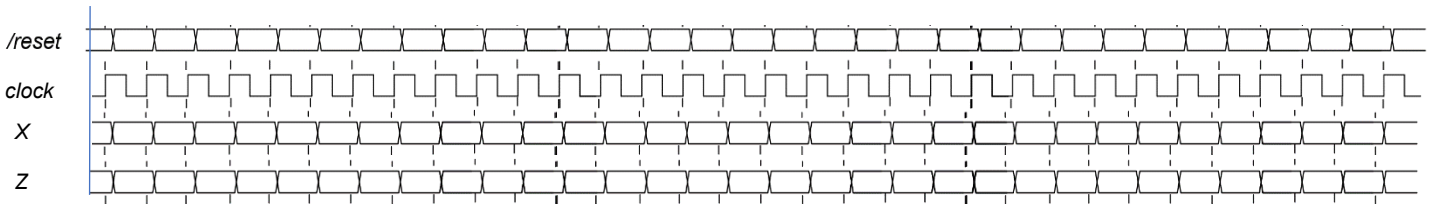
2) [5/30] Si consideri una cache di dimensione 96B e a 3 vie di tipo write-back/write-non-allocate. La dimensione del blocco e' 8 byte, il tempo di accesso alla cache e' 4 ns e la penalita' in caso di miss e' pari a 40 ns, la politica di rimpiazzamento e' FIFO. Il processore effettua i seguenti accessi in cache, ad indirizzi al byte: 1177, 2163, 123, 3181, 1200, 4221, 1175, 2184, 1182, 4201, 5176, 7173, 3176, 8183, 7251, 5176, 3201, 4180, 6171, 8178. 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) [4/30] Fornendo una spiegazione ragionata, con il dettaglio del significato dei vari bit per il formato dell'istruzione, determinare a quale istruzione assembly RISC-V corrisponde la seguente stringa binaria (codice macchina) 1111 1110 0000 0011 1001 0110 1110 0011 (i codici operativi e il significato delle istruzioni RISC-V sono riportate nella tabella a pagina iniziale).

4) [11/30] Il "decoder con abilitazione" è un decoder con un ingresso aggiuntivo E (Enable) che abilita le uscite quando E=1 mentre quando E =0 tutte le uscite valgono 0: la tabella di verità nel caso da-2-a-4 è rappresentata nella figura a lato. Realizzare in Verilog tale decoder con abilitazione da-2 a-4 ([punti 4]) e successivamente realizzare in Verilog un decoder "da-4-a-16" utilizzando il modulo Verilog del decoder con abilitazione "da-2-a-4" ([punti 4]). **Tracciare il diagramma di temporizzazione ([punti 3])** come verifica della correttezza dei moduli realizzati, utilizzando il testbench fornito di seguito. Modello del diagramma temporale da tracciare:



INGRESSI			USCITE			
E	x_1	x_0	z_3	z_2	z_1	z_0
0	X	X	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0




```
module testbench;
  reg reset_;initial begin reset_=0; #22 reset_=1; #300; $stop; end
  reg clock ;initial clock=0; always #5 clock <=(!clock);
  reg[3:0] X; wire[15:0] Z;
  always #10 if (reset_==1) X <= X+1;
  initial begin
    X = 0;          // Initialize inputs
    $monitor("Input: %b, Output: %b", X, Z); // Monitor outputs
    #180 $finish;  // End simulation
  end
  Decoder_4to16 decoder (Z,X);
endmodule
```

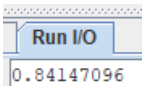
COMPITO di ARCHITETTURA DEI CALCOLATORI del 16-11-2023

SOLUZIONE

ESERCIZIO 1

```
.data
sin_value:  .float 0.0
one:        .float 1.0

.text
.globl main

#-----------------------------------------
fact: # input: a0=n - Output: a0=fact
  li  t1, 1      # fact = 1
  li  t0, 1      # i = 1
fact_for_start:
  slt  t2,a0,t0  # i>?n --> exitfor
  bne  t2,x0,fact_for_end
  mul  t1,t1,t0  # fact *= i
  addi t0,t0,1   # i++
  b fact_for_start
fact_for_end:
  mv   a0,t1     # return fact
  ret

#-----------------------------------------
power:# input:fa0=x,a0=n - output:fa0
  # Initialize variables
  la   t0,one     # &one
  flw  ft0,0(t0)  # 1.0
  li   t0, 0      # i = 0
p_for_start:
  beq  t0,a0,p_for_end # i == n --> exitfor
  fmul.s ft0,ft0,fa0   # result *= x
  addi t0,t0,1    # i++
  b p_for_start
p_for_end:
  fmv.s fa0,ft0   # return result
  ret
```

```
#-----------------------------------------
sin_taylor:#input:fa0=x,a0=terms -output:fa0
  addi  sp,sp,-52 # allocate frame
  sd    ra, 44(sp)
  sd    s0, 36(sp)
  sd    s1, 28(sp)
  sd    s2, 20(sp)
  sd    s3, 12(sp)
  fsw   fs0, 8(sp)
  fsw   fs1, 4(sp)
  fsw   fs2, 0(sp)

  fmv.s fs0,fa0  # save fa0 (x)
  mv    s0,a0    # save a0 (terms)
  fmv.s.x fs2,x0 # result = 0.0
  li    s1,1     # pwridx = 1
  li    s2,1     # sign = 1

  li    s3,1     # i=1
sin_for_start:
  slt   t6,s0,s3 # i>?terms --> exitfor
  bne   t6,x0,sin_for_end

  # sign * power(x, pwridx) / fact(pwridx)
  fmv.s fa0,fs0  # x
  mv    a0,s1    # pwridx
  call  power
  fmv.s fs1,fa0  # save power
  mv    a0,s1    # pwridx
  call  fact
  fcvt.s.w fa0,a0# (float)fact
  fdiv.s fs1,fs1,fa0 # power/fact
  fcvt.s.w ft1,s2# sign
  fmul.s ft0,fs1,ft1 # sign*power/fact
  fadd.s fs2,fs2,ft0 # result += (.)
  li    t0,-1    # -1
  mul   s2,s2,t0 # sign *= -1
  addi  s1,s1,2  # pwridx +=2

  addi  s3,s3,1  # ++i
  b sin_for_start

sin_for_end:
  fmv.s fa0,fs2   # result
  ld    ra, 44(sp)
  ld    s0, 36(sp)
  ld    s1, 28(sp)
  ld    s2, 20(sp)
  ld    s3, 12(sp)
  flw   fs0, 8(sp)
  flw   fs1, 4(sp)
  flw   fs2, 0(sp)
  addi  sp,sp,52 # deallocate frame
  ret

#-----------------------------------------
main:
  la   t0,one     # &one
  flw  fa0,0(t0)  # 1.0
  li   a0,5       # 2nd parm
  call sin_taylor # returns result in fa0
  li   a7,2       # print_float
  ecall
  li   a7,10      # 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=96/8/3,     XM=X/B,    XS=XM%S,   XT=XM/S.
A=3, B=8, C=96, RP=FIFO, Thit=4, Tpen=40, 20 references:
=== T    X    XM   XT  XS  XB  H [SET]:USAGE [SET]:MODIF  [SET]:TAG
=== R 1177  147  36   3   1  0 [3]:2,0,0 [3]:0,0,0  [3]:36,-,-
=== W 2163  270  67   2   3  0 [2]:2,0,0 [2]:0,0,0  [2]:67,-,-
=== R  123   15   3   3   3  0 [3]:1,2,0 [3]:0,0,0  [3]:36,3,-
=== W 3181  397  99   1   5  0 [1]:2,0,0 [1]:0,0,0  [1]:99,-,-
=== R 1200  150  37   2   0  0 [2]:1,2,0 [2]:0,0,0  [2]:67,37,-
=== W 4221  527 131   3   5  0 [3]:0,1,2 [3]:0,0,0  [3]:36,3,131
=== R 1175  146  36   2   7  0 [2]:0,1,2 [2]:0,0,0  [2]:67,37,36
=== W 2184  273  68   1   0  0 [1]:1,2,0 [1]:0,0,0  [1]:99,68,-
=== R 1182  147  36   3   6  1 [3]:0,1,2 [3]:0,0,0  [3]:36,3,131
=== W 4201  525 131   1   1  0 [1]:0,1,2 [1]:0,0,0  [1]:99,68,131
=== R 5176  647 161   3   0  0 [3]:2,0,1 [3]:0,0,0  [3]:161,3,131
=== W 7173  896 224   0   5  0 [0]:2,0,0 [0]:0,0,0  [0]:224,-,-
=== R 3176  397  99   1   0  1 [1]:0,1,2 [1]:0,0,0  [1]:99,68,131
=== W 8183 1022 255   2   7  0 [2]:2,0,1 [2]:0,0,0  [2]:255,37,36
=== R 7251  906 226   2   3  0 [2]:1,2,0 [2]:0,0,0  [2]:255,226,36
=== W 5176  647 161   3   0  1 [3]:2,0,1 [3]:1,0,0  [3]:161,3,131
=== R 3201  400 100   0   1  0 [0]:1,2,0 [0]:0,0,0  [0]:224,100,-
=== W 4180  522 130   2   4  0 [2]:0,1,2 [2]:0,0,0  [2]:255,226,130
=== R 6171  771 192   3   3  0 [3]:1,2,0 [3]:1,0,0  [3]:161,192,131
=== W 8178 1022 255   2   2  1 [2]:0,1,2 [2]:1,0,0  [2]:255,226,130
---------------------------------------
P1 Nmiss=16   Nhit=4  Nref=20  mrate=0.800000   AMAT=th+mrate*tpen=36
```

LISTA BLOCCHI USCENTI:

```
(out: XM=147 XT=36 XS=3 )

(out: XM=270 XT=67 XS=2 )
(out: XM=150 XT=37 XS=2 )

(out: XM=146 XT=36 XS=2 )
(out: XM=15 XT=3  XS=3 )
```

CONTENUTI dei SET al termine

ESERCIZIO 3

La rappresentazione binaria dell'istruzione data è: 1111 1110 0000 0011 1001 0110 1110 0011

Dai primi 7 bit vediamo che l'opcode e' 0b110 0011 (0x63) ed un eventuale funct3 e' 1 ovvero opcode/funct3=0x63/1 che corrisponde al codice operativo dell'istruzione BEQ, che quindi usa il "formato B". Una volta determinato il formato, riportiamo la distribuzione dei bit dati nei campi del formato B:

1111 111	00 000	0 0111	001	0110 1	110 0011
{imm[12],imm[10:5]}	**rs2**	**rs1**	**funct3**	**{imm[4:1],imm[11]}**	**opcode**

Quindi rs1=x7 e rs2=x0, mentre imm={imm[12],imm[11],imm[10:5],imm[4:1],0}={1,1,111111,0110,0} che in decimale corrisponde a -20, che significa che l'eventuale branch deve saltare a -20 byte indietro. Dunque, l'istruzione corrispondente è:

```
beq x7, x0, -20
```

COMPITO di ARCHITETTURA DEI CALCOLATORI del 16-11-2023

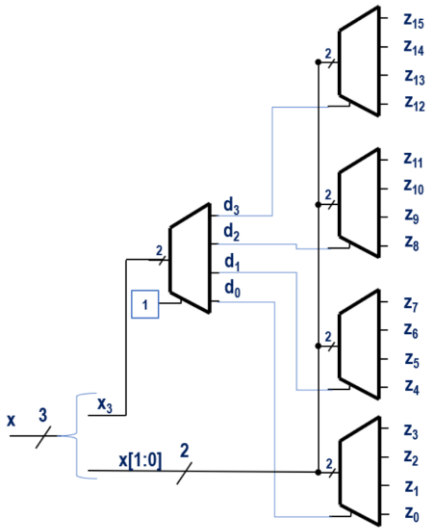
SOLUZIONE

ESERCIZIO 4

Si può realizzre il decoder con abilitazione da-2-a-4 con il seguente codice Verilog:

```
module Decoder_2to4plusE(z,  x,E);
  output[3:0] z; reg[3:0] z;
  input[1:0] x;
  input E;
  always @(x or E) casex ({E,x})
    3'b0xx: z = 4'b0000;
    3'b100: z = 4'b0001;
    3'b101: z = 4'b0010;
    3'b110: z = 4'b0100;
    3'b111: z = 4'b1000;
  endcase
endmodule
```

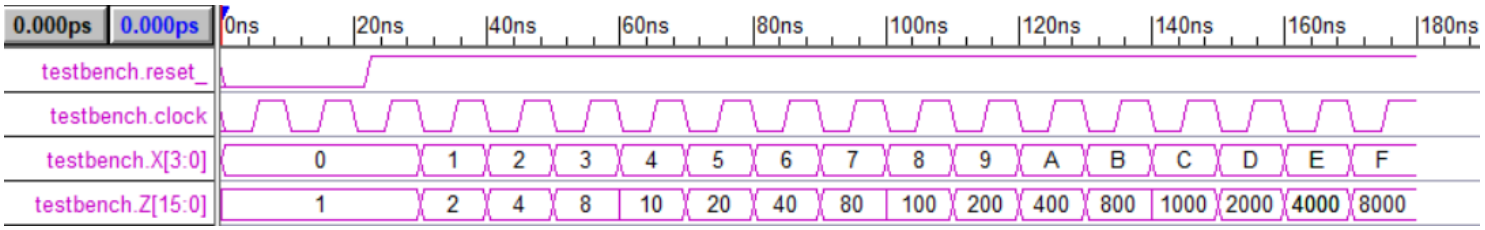
Dopodiche il decoder da-4-a-16 può essere così realizzato:



Ovvero:

```
module Decoder_4to16(z,  x);
  output[15:0] z;
  input[3:0] x;
  wire[3:0] d;

  Decoder_2to4plusE dc(d,         x[3:2],1'b1);
  Decoder_2to4plusE d3(z[15:12], x[1:0],d[3]);
  Decoder_2to4plusE d2(z[11:8],  x[1:0],d[2]);
  Decoder_2to4plusE d1(z[7:4],   x[1:0],d[1]);
  Decoder_2to4plusE d0(z[3:0],   x[1:0],d[0]);

endmodule
```

Diagramma di Temporizzazione:



Sullo standard output la primitiva “monitor” mostra:

```
Input: 0000, Output: 0000000000000001
Input: 0001, Output: 0000000000000010
Input: 0010, Output: 0000000000000100
Input: 0011, Output: 0000000000001000
Input: 0100, Output: 0000000000010000
Input: 0101, Output: 0000000000100000
Input: 0110, Output: 0000000001000000
Input: 0111, Output: 0000000010000000
Input: 1000, Output: 0000000100000000
Input: 1001, Output: 0000001000000000
Input: 1010, Output: 0000010000000000
Input: 1011, Output: 0000100000000000
Input: 1100, Output: 0001000000000000
Input: 1101, Output: 0010000000000000
Input: 1110, Output: 0100000000000000
Input: 1111, Output: 1000000000000000
```