COMPITO di ARCHITETTURA DEI CALCOLATORI del 19-07-2024

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) [9/30] 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 sum_of_factorials(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        if (i % 2 == 0) {
            sum += i * i;
        } else {
            sum += i;
        }
    }
    return sum;
}
```

```
int double_sum_of_factorials(int n) {
    return 2 * sum_of_factorials(n);
}

int main() {
    int result = double_sum_of_factorials(5);
    print_int(result);
    exit();
}
```

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	**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	---	---

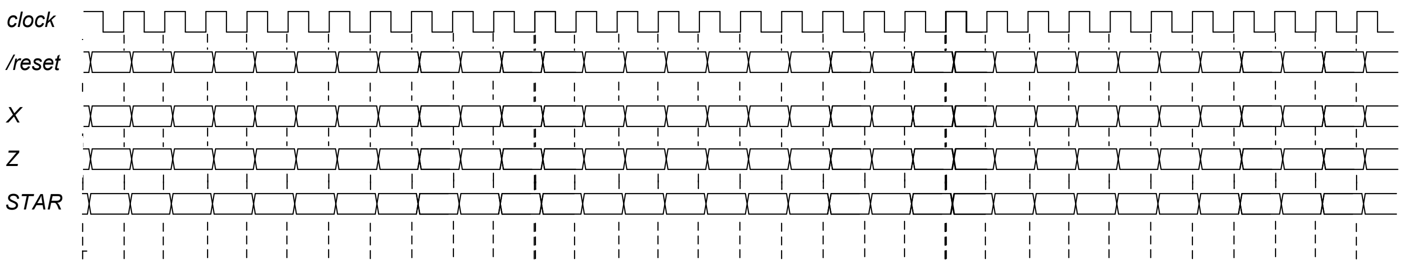
COMPITO di ARCHITETTURA DEI CALCOLATORI del 19-07-2024

2) [6/30] Disegnare l'organizzazione fisica di una memoria DRAM da 16Mbit indicando i collegamenti fra i blocchi CTRL, ROW_LATCHES, COL_LATCHES, ROW_DECODER, COL_DECODER, ROW_BUFFERS, PRECHARGE, TRISTATE e spiegare lo svolgimento delle operazioni di accesso ad un singolo bit.

3) [6/30] Spiegare il funzionamento del sommatore Carry-Look-Ahead. Ricavare le equazioni che legano le variabili booleane interne Propagate e Generate nel caso a 4 bit, mostrare l'architettura in termini di porte logiche elementari, sia del sommatore CLA che della rete di generazione del carry (sempre nel caso a 4 bit).

4) [9/30] Descrivere e sintetizzare in Verilog una rete sequenziale utilizzando il modello di Moore con un ingresso X su un bit e una uscita Z su due bit che funziona nel seguente modo: devono essere riconosciute le sequenze <u>interallacciate</u> 1,1,0,1,1, e 1,0,1,0,1; l'uscita Z0 va a 1 (per 1 ciclo di clock) se è presente una delle la prima sequenza, l'uscita Z1 va a 1 (per 1 ciclo di clock) se è presente la seconda sequenza. Gli stimoli di ingresso sono dati dal seguente modulo Verilog Testbench.

   Tracciare il diagramma di temporizzazione [4/9 punti] come verifica della correttezza dell'unità. 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. 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 X;
wire [1:0] Z;
wire [3:0] STAR=Xxx.STAR;
initial begin X=0;
wait(reset_==1); #5
  @(posedge clock);X<=1;@(posedge clock);X<=1;@(posedge clock);X<=0;@(posedge clock);X<=1;@(posedge clock);X<=1;
  @(posedge clock);X<=1;@(posedge clock);X<=0;@(posedge clock);X<=1;@(posedge clock);X<=0;@(posedge clock);X<=1;
  @(posedge clock);X<=1;@(posedge clock);X<=0;@(posedge clock);X<=1;@(posedge clock);X<=1;@(posedge clock);X<=0;
  @(posedge clock);X<=1;@(posedge clock);X<=1;@(posedge clock);X<=0;@(posedge clock);X<=1;@(posedge clock);X<=1;
  @(posedge clock);X<=0;@(posedge clock);X<=1;@(posedge clock);X<=0;@(posedge clock);X<=1;@(posedge clock);X<=1;
  $finish;
end
XXX Xxx(X,Z,clock,reset_);
endmodule
```

COMPITO di ARCHITETTURA DEI CALCOLATORI del 19-07-2024

SOLUZIONE

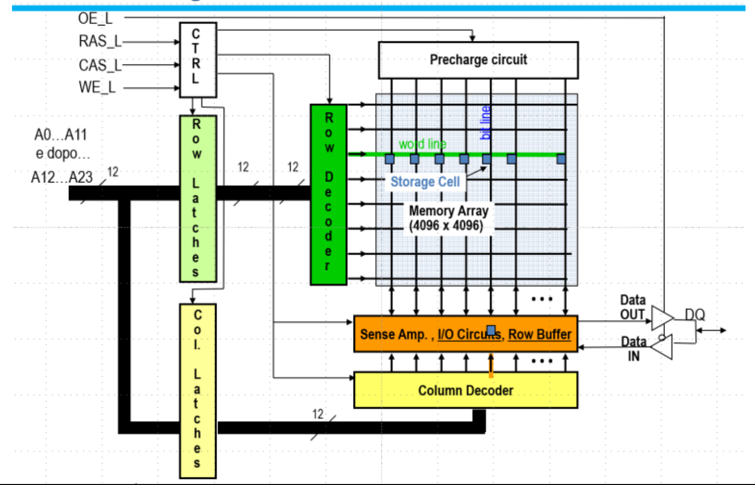
ESERCIZIO 1

```
.text
.globl main

# Funzione sum_of_factorials
sum_of_factorials:
  li t0, 0 # sum = 0
  li t1, 1 # i = 1

loop_start:
  slt t2, a0, t1      # i >? n
  bne t2, x0, loop_end # se (i > n) esci

loop_body:
  andi t2, t1, 1      # t2 = i % 2
  beq t2, x0, even    # se i è pari, vai a even_case
  add t0, t0, t1      # somma += i (caso dispari)
  b increment         # vai a increment

even:
  mul t2, t1, t1      # t2 = i * i (quadrato)
  add t0, t0, t2  # somma += t2 (aggiungi quadrato)

increment:
  addi t1, t1, 1   # i++
  b loop_start     # ripeti il ciclo

loop_end:
  mv a0, t0        # restituisci la somma
  ret

# Funzione double_sum_of_factorials
double_sum_of_factorials:
  addi sp, sp, -8  # Creare spazio nello stack
  sd ra, 0(sp)     # Salvare l'indirizzo di ritorno
  call sum_of_factorials   # Chiamare
sum_of_factorials
  slli a0, a0, 1   # a0 = a0 * 2 (raddoppiare il
risultato)
  ld ra, 0(sp) # Ripristina l'indirizzo di ritorno
  addi sp, sp, 8   # Ripristina lo stack
  ret

# Funzione main
main:
  li a0, 5   # n = 5
  call double_sum_of_factorials # Chiamare
double_sum_of_factorials

  # Stampa il risultato
  li a7, 1   # syscall per print_int
ecall        # Chiamare il sistema

  li a7, 10  # ecall per uscire
ecall        # Chiamare il sistema
```

```
58
```

ESERCIZIO 2

DRAM: Organizzazione fisica di memoria a 16 Mbit



- I 24 bit che compongono l'indirizzo vengono inviato in due gruppi di 12 bit ciascuno e memorizzando i 12 bit più significativi ad es. nei ROW_LATCHES e I 12 meno significative nei COL_LATCHES
- **In ogni caso i 24 bit non possono essere utilizzati simultaneamente, in quanto a causa della struttura compatta della cella DRAM deve innanzitutto essere letta una intera riga della matrice 4096x4096 e bufferizzata nel ROW_BUFFER**
- In una fase successiva si possono usare I bit del COL_LATCHES per Selezionare il singolo bit all'interno della riga di 4096 bit bufferizzata
- Una volta selezionato un bit questo può essere letto dirigendolo sull'uscita DQ attraverso opportuna selezione dei tristate buffer (DATA_IN, DATA_OUT)
- **L'operazione di lettura si svolge in modo analogo ma andando preventivamente a scrivere nel ROW_BUFFER il bit di interesse e poi riscrivendo l'intera riga letta all'interno della matrice di bit.**

ESERCIZIO 3

(V. PHRV1, APPENDICE A.6)

Carry Look-Ahead Adder (Riporto Parallelo)

$$s_i = a_i \oplus b_i \oplus c_{i-1}$$

$$c_i = a_i b_i + a_i \cdot c_{i-1} + b_i \cdot c_{i-1} = a_i b_i + c_{i-1}(a_i + b_i) = a_i b_i + c_{i-1}(a_i \oplus b_i)$$

ai+bi si può sostituire col suo xor perché nell'espressione quando ai=1 e bi=1 anche ai*bi=1

$$\left.\begin{aligned} G_i &\triangleq a_i b_i \\ P_i &\triangleq a_i \oplus b_i \end{aligned}\right] \Rightarrow \begin{aligned} s_i &= P_i \oplus c_{i-1} \\ c_i &= G_i + P_i \cdot c_{i-1} \end{aligned}$$

Gi=generate
Pi=propagate

$$C_0 = G_0 + P_0 \cdot C_{in}$$
$$C_1 = G_1 + P_1 \cdot C_0 = G_1 + G_0 P_1 + P_1 P_0 \cdot C_{in}$$
$$C_2 = G_2 + P_2 C_1 = G_2 + G_1 P_2 + G_0 P_1 P_2 + P_2 P_1 P_0 \cdot C_{in}$$
$$C_3 = G_3 + P_3 \cdot C_2 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + P_3 P_2 P_1 P_0 C_{in}$$

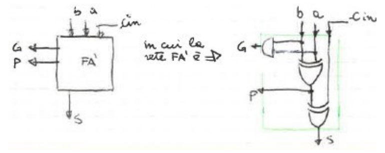
Carry Look-Ahead Adder (2)

c_{i-1}	a_i	b_i	s_i	c_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Un altro modo di vedere la funzione di Generate e Propagate:

In questi due casi il carry di uscita è presente perché "si genera" a causa del fatto che nella somma ai e bi sono entrambi "a uno"

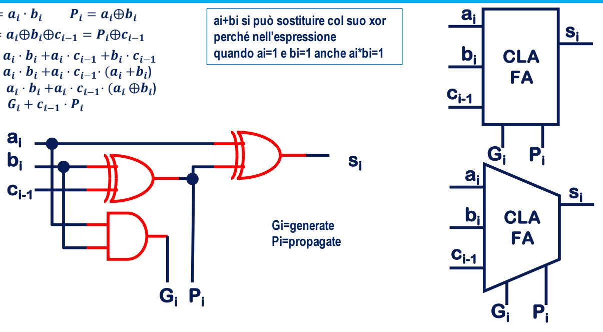
In questi altri due casi il carry di uscita è presente perché "si propaga" dal carry precedente (ovvero dal Full Adder precedente)



Full Adder per Carry-Look-Ahead in VERILOG

$$G_i = a_i \cdot b_i \qquad P_i = a_i \oplus b_i$$
$$s_i = a_i \oplus b_i \oplus c_{i-1} = P_i \oplus c_{i-1}$$
$$\begin{aligned} c_i &= a_i \cdot b_i + a_i \cdot c_{i-1} + b_i \cdot c_{i-1} \\ &= a_i \cdot b_i + a_i \cdot c_{i-1} \cdot (a_i + b_i) \\ &= a_i \cdot b_i + a_i \cdot c_{i-1} \cdot (a_i \oplus b_i) \\ &= G_i + c_{i-1} \cdot P_i \end{aligned}$$

ai+bi si può sostituire col suo xor perché nell'espressione quando ai=1 e bi=1 anche ai*bi=1

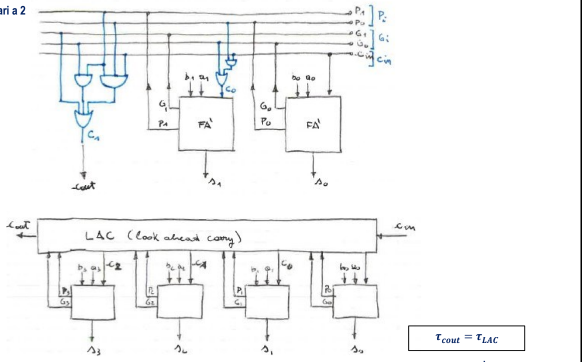


Gi=generate
Pi=propagate

```
module cla_full_adder(input a,input b,input c,output g,output p,output s);
        assign g = a & b;
        assign p = a ^ b;
        assign s = a ^ (b ^ c);
endmodule
```

Carry Look-Ahead Adder (3)

Es. Nel caso pari a 2



$$\boxed{\tau_{cout} = \tau_{LAC}}$$

$$\tau_{tot} = \tau_{LAC} + \tau_{FA'}$$

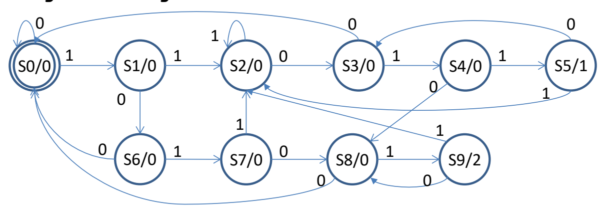
COMPITO di ARCHITETTURA DEI CALCOLATORI del 19-07-2024

SOLUZIONE

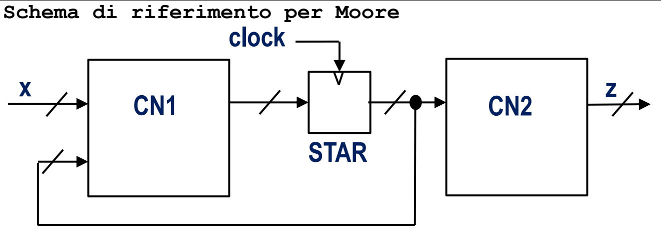
ESERCIZIO 4

In corrispondenza del pattern X_{t-3}, X_{t-2},X_{t-1},X_t = 1,1,1,1 oppure 1,0,0,1 INTERALLACCIATI ottengo → Z_{t+1} = 1; (ricordare che e' richiesto Moore).

Diagramma degli Stati



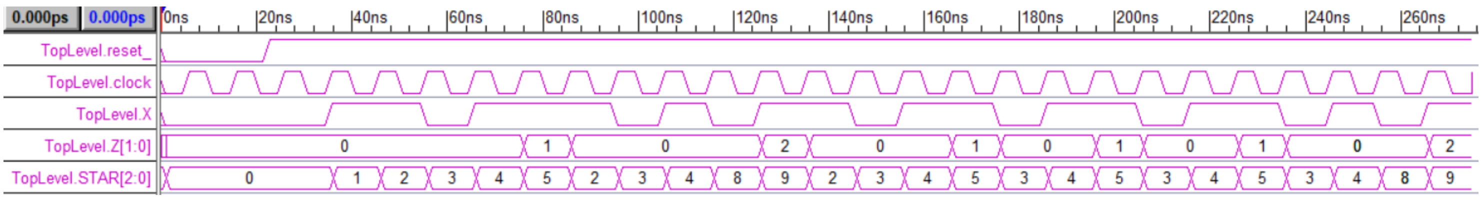
Schema di riferimento per Moore



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

```
module XXX(X, Z, clock, reset_);
  input X;
  input clock, reset_;
  output [1:0] Z;
  reg [3:0] STAR;
  parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3, S4 = 4, S5 = 5, S6 = 6, S7 = 7, S8 = 8, S9 = 9, S10 = 10;

  // Reset logic
  always @(reset_==0)#1 begin STAR<=S0; end

  // Output logic
  assign Z = (STAR == S5) ? 1 : (STAR == S9) ? 2 : 0;

  // State transition logic
  always @(posedge clock) if (reset_ == 1) #1 begin
    casex (STAR)
      S0: STAR <= (X == 1) ? S1 : S0;
      S1: STAR <= (X == 1) ? S2 : S6;
      S2: STAR <= (X == 1) ? S2 : S3;
      S3: STAR <= (X == 1) ? S4 : S0;
      S4: STAR <= (X == 1) ? S5 : S8;
      S5: STAR <= (X == 1) ? S2 : S3;
      S6: STAR <= (X == 1) ? S7 : S0;
      S7: STAR <= (X == 1) ? S2 : S8;
      S8: STAR <= (X == 1) ? S9 : S0;
      S9: STAR <= (X == 1) ? S2 : S8;
    endcase
  end
endmodule
```

Diagramma di Temporizzazione:


0.000ps
0.000ps
0ns
20ns
40ns
60ns
80ns
100ns
120ns
140ns
160ns
180ns
200ns
220ns
240ns
260ns
TopLevel.reset_
TopLevel.clock
TopLevel.X
TopLevel.Z[1:0]
TopLevel.STAR[2:0]