Circuite integrate digitale

Curs 12

Cuprins

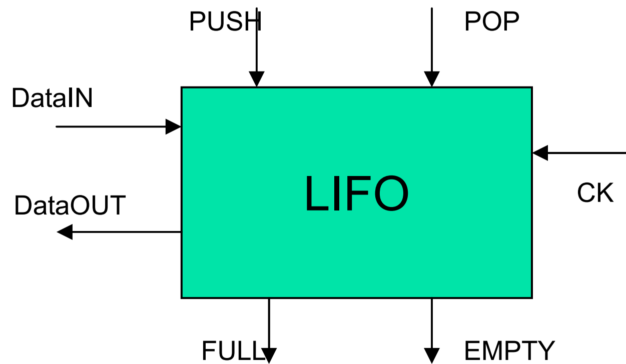
- aplicaţii
  - register file
  - LIFO
  - FIFO
- implementarea unor algoritmi
  - datapath
  - controller
- sisteme de ordin 3: procesoare

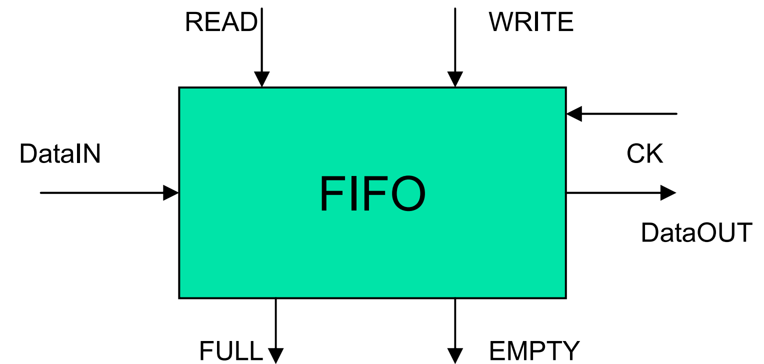
Register file

- colecţie (matrice) de registre
- principala aplicaţie: CPU din arhitectura procesoarelor, microprocesoarelor şi microcontrollerelor
- se implementează cu un SRAM multiport

```
module regfile(clock, reset, writeEnable, dest, source, dataIn,
dataOut);
parameter WIDTH = 16;
parameter DEPTH = 32;
parameter ADDRESSWIDTH = 5;
integer i,j;
input clock, reset, writeEnable;
input [ADDRESSWIDTH-1 : 0] dest;
input [ADDRESSWIDTH-1 : 0] source;
input [WIDTH-1 : 0] dataIn;
output [WIDTH-1 : 0] dataOut;
reg [WIDTH-1 : 0] dataOut; // registered output
reg [WIDTH-1 : 0] rf [DEPTH-1 : 0];
wire [DEPTH-1 : 0] writeEnableDecoded;

assign writeEnableDecoded = (writeEnable << dest);

// flip-flop for data-out
always@(posedge clock)
            begin
            if(!reset) dataOut <= 0;
            else dataOut <= rf[source];
            end

// memory array
always@(posedge clock)
            begin
            if(!reset)
                        begin
                        for(i = 0; i<DEPTH; i=i+1)
                        rf[i] <= 0;
                        end
            else
                        begin
                        for (j=0; j<DEPTH; j=j+1)
                        if(writeEnableDecoded[j]) rf[j] <= dataIn;
                        end
            end //always

endmodule
```

LIFO şi FIFO





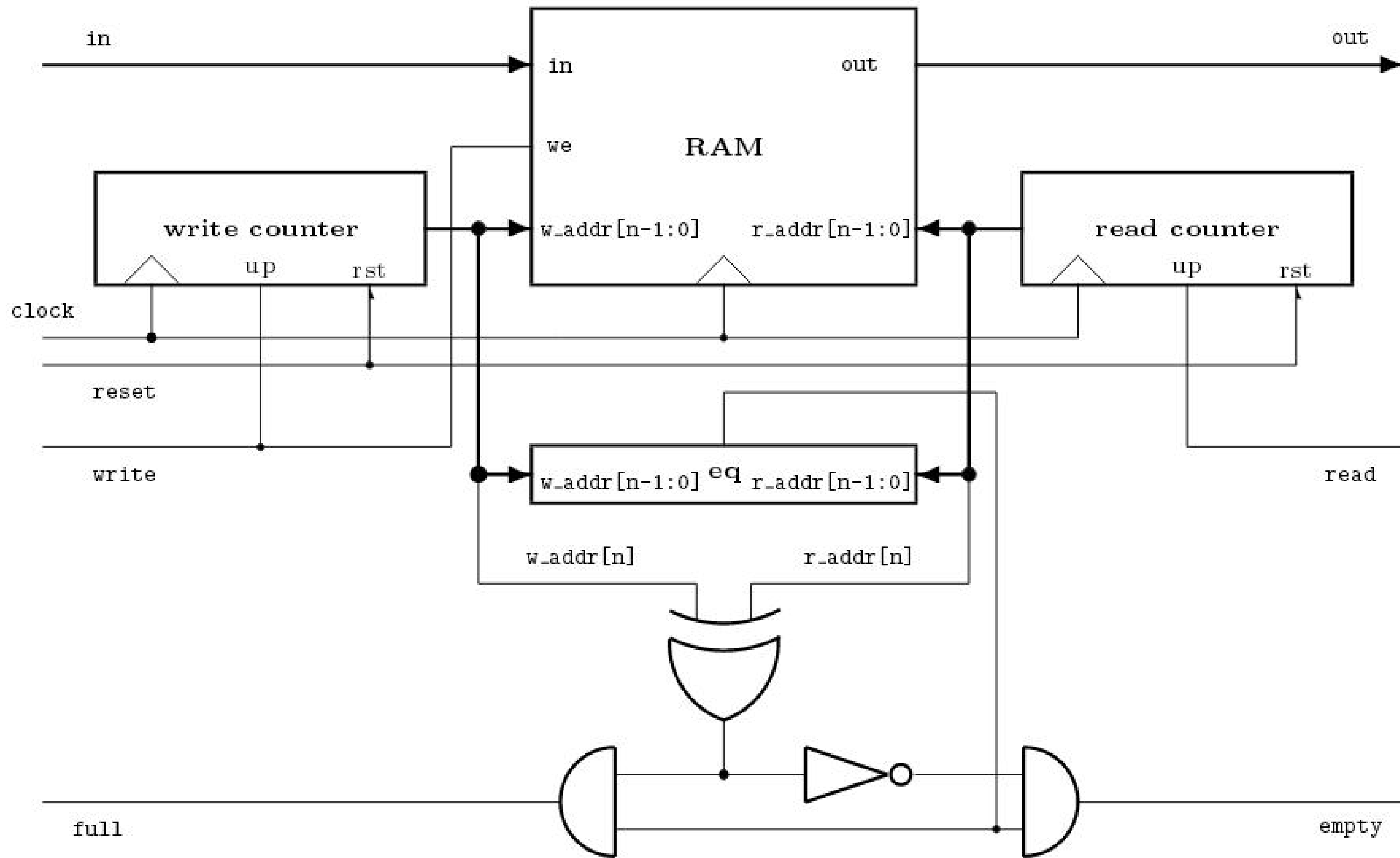
- memorii, implementate ca automate
- *stivă* vs. *coadă*
- LIFO: datele sunt accesibile numai în "vârful" stivei
- FIFO: datele pot fi citite doar în ordinea în care au fost scrise
- ambele nu necesită port de adrese
- citirea este destructivă

FIFO implementat cu RegisterFile

```
`include "regfile.v"

module fifo (clock, reset, inData, new_data, out_data, outData, full);

input clock;
input reset;
input [WIDTH-1 : 0] inData;
input new_data;
input out_data;
parameter WIDTH = 16;
parameter DEPTH = 16;
parameter ADDRESSWIDTH = 5;
integer k; //index for "for" loops
output [WIDTH-1 : 0] outData;
output full;
reg full; // registered output
wire fullD; // input to "full" flip-flop
reg [ADDRESSWIDTH-1 : 0] rear; // points to rear of list
reg [ADDRESSWIDTH-1 : 0] front; // points to front of list
```

```
/ flip-flops to hold value of "rear"
always@(posedge clock)
          begin
          if (!reset) rear <= 0;
          else if(new_data)
          begin
                    if (rear == DEPTH) rear <= 0;
                    else rear <= rear+1;
          end
          end

// flip-flops to hold value of "front"
always@(posedge clock)
          begin
          if (!reset) front <= 0;
          else if(out_data)
          begin
                    if (front == DEPTH) front <= 0;
                    else front <= front+1;
          end
          end

// flip-flop for "full" signal
always@(posedge clock)
          begin
          if (!reset) full <= 0;
                    else full <= fullD;
          end

// full signal
assign fullD = (front == ((rear==DEPTH) ? 0 : (rear+1)));

regfile u1 (clock, reset, new_data, rear, front, inData, outData);

endmodule
```

FIFO implementat cu RAM



```
module simple_fifo(output [31:0] out ,
                   output empty ,
                   output full ,
                   input [31:0] in,
                   input write ,
                   input read ,
                   input reset ,
                   input clock );
wire [9:0] write_addr, read_addr;

counter write_counter( .out (write_addr ),
                             .reset (reset),
                             .count_up (write),
                             .clock    (clock)),
read_counter( .out(read_addr ),
                   .reset(reset),
                   .count_up (read),
                   .clock(clock));
dual_ram memory(.out(out),
                   .in(in),
                   .read_addr (read_addr[8:0] ),
.write_addr (write_addr[8:0]),
                   .we(write),
                   .clock(clock));
assign eq = read_addr[8:0] == write_addr[8:0] ,
          phase = ~(read_addr[9] == write_addr[9]) ,
          empty = eq & phase ,
          full = eq & ~phase ;

endmodule
```

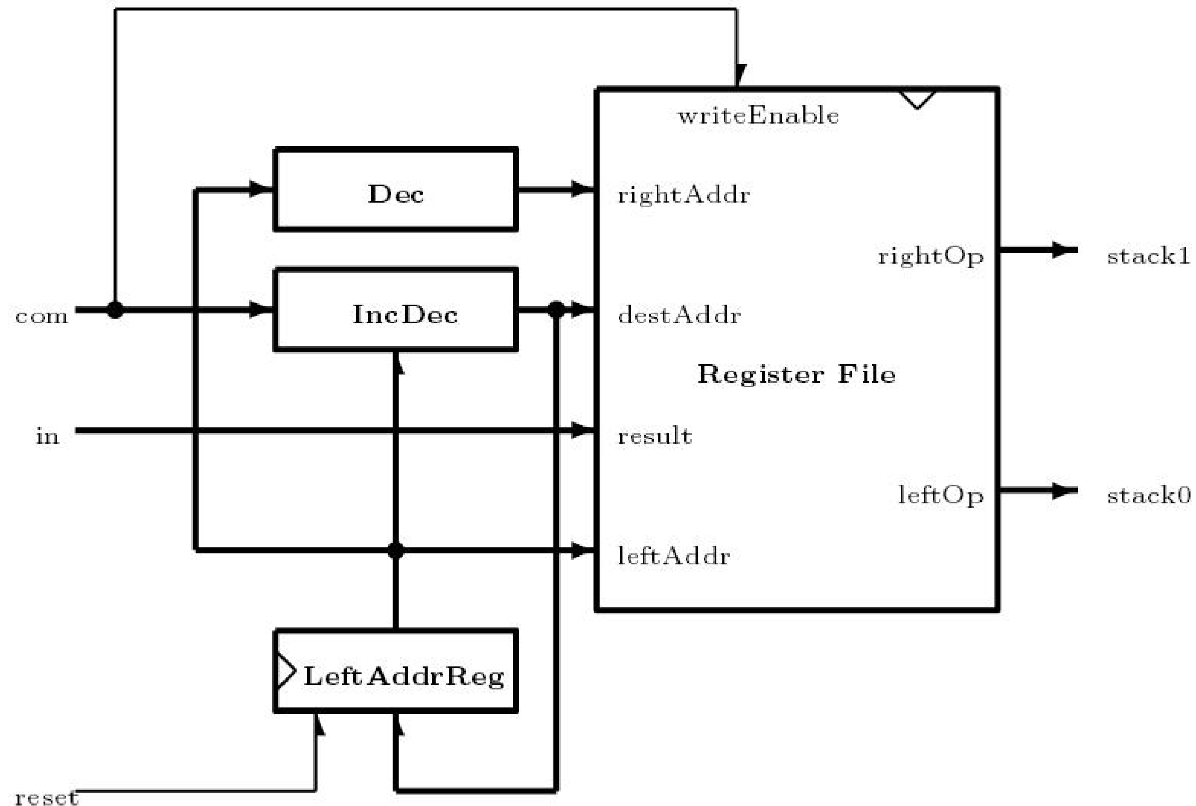
```
module counter(output reg [9:0] out ,
                    input reset ,
                    input count_up,
                    input clock );

always @(posedge clock) if (reset)
         out <= 0;
         else if (count_up) out <= out + 1;
endmodule


module dual_ram( output [31:0] out,
                    input [31:0] in,
                    input [8:0] read_addr ,
                    input [8:0] write_addr ,
                    input we,
                    input clock);

reg [63:0] mem[511:0];

assign out = mem[read_addr] ;
always @(posedge clock) if (we) mem[write_addr] <= in ;

endmodule
```

LIFO cu set extins de operaţii

stiva S = <s0, s1, s2, ...>

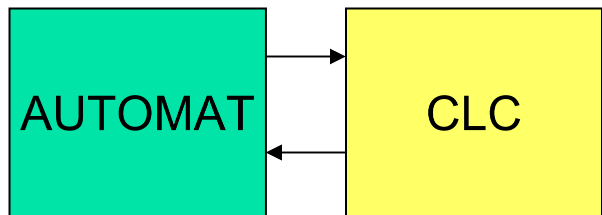
- **nop** : no operation;
- **write** a : scrie a in TOS (top of stack)
  <s0, s1, s2, ...> --> <a, s1, s2, ...>
  folosit pentru operaţii unare, de ex: a = s0 + 1
  (pop, push = write)
- **pop** : <s0, s1, s2, ...> --> <s1, s2, ...>
- **popwr** a : pop & write
  <s0, s1, s2, ...> --> <a, s2, ...>
  folosit pentru operaţii binare, de ex: a = s0 + s1
  (pop, pop, push = pop, write)
- **push** a : <s0, s1, s2, ...> --> <a, s0, s1, s2, ...>

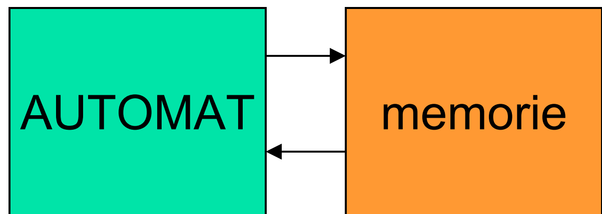
LIFO implementat cu RegisterFile

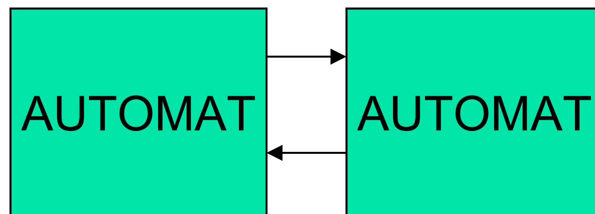


```
module lifo ( output [m-1:0] stack0, stack1,
                 input [m-1:0] in,
                 input [2:0] com,
                 input reset , clock );

// The command codes
nop = 3'b000, //
write = 3'b001, // we
pop = 3'b010, // dec
popwr = 3'b011, // dec, we
push = 3'b101; // inc, we */
reg [n-1:0] leftAddr; // the main pointer
wire [n-1:0] nextAddr;

// The increment/decrement circuit
assign nextAddr = com[2] ? (leftAddr + 1'b1) : (com[1] ? (leftAddr - 1'b1)
: leftAddr);
// The address register for TOS
always @(posedge clock)
        if (reset) leftAddr <= 0;
        else leftAddr <= nextAddr;
// The register file
reg [m-1:0] file[0:(1'b1 << n)-1];
assign stack0 = file[leftAddr],
        stack1 = file[leftAddr - 1'b1];
always @(posedge clock) if (com[0]) file[nextAddr] <= in;
endmodule
```

Sisteme de ordin 3

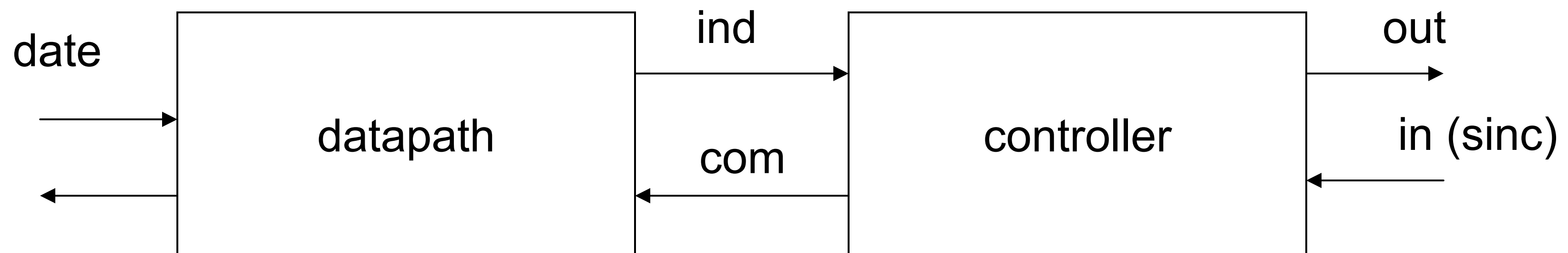






procesoare

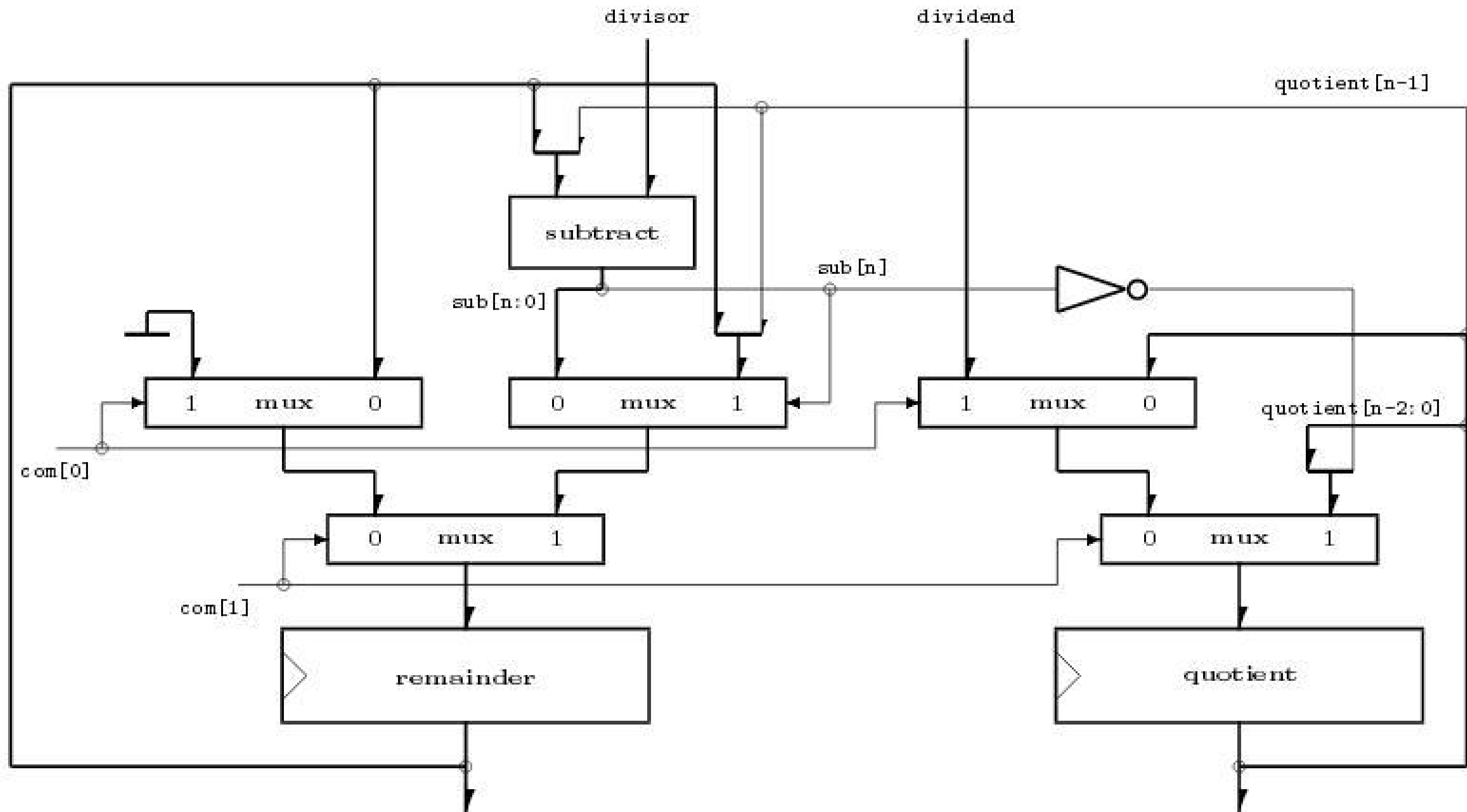
Implementarea unor algoritmi



- datapath: reuneşte circuitele de calcul, memorarea datelor, busuri
- controllerul generează semnalele necesare pentru funcţionarea acestora

Sequential divider

dividend/divisor = quotient + remainder

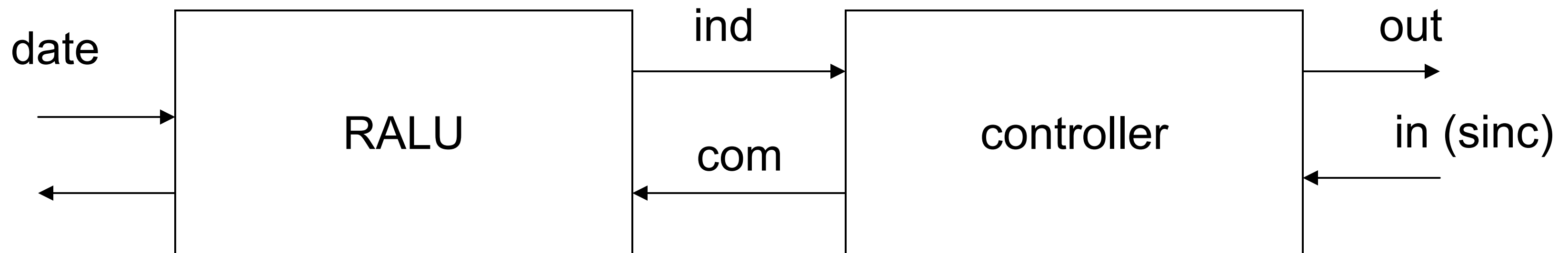


```
module divisor ( output reg [n-1:0] quotient ,
                    output reg [n-1:0] remainder ,
                    output error,
                    input [n-1:0] dividend ,
                    input [n-1:0] divisor ,
                    input [1:0] com,
                    input clk);

parameter n = 8;
parameter nop = 2'b00,
          ld = 2'b01,
          div = 2'b10;
wire [n:0] sub;

assign error = (divisor == 0) & (com == div);
assign sub = {remainder, quotient[n-1]} - {1'b0, divisor} ;

always @(posedge clk)
          if (com == ld)
          begin quotient <= dividend ;
          remainder <= 0;
          end
                    else if (com == div) begin quotient <= { quotient[n-2:0],~sub[n]};
                    remainder <= sub[n] ? {remainder[n-2:0], quotient[n-1]}:sub[n-1:0];
end

endmodule
```

Tema 14

- Explicaţi funcţionarea circuitului de împărţire secvenţială, evidenţiind secvenţa corectă de comenzi
- *indicaţie*: http://arh.pub.ro/gstefan/0-BOOK.pdf

Procesor

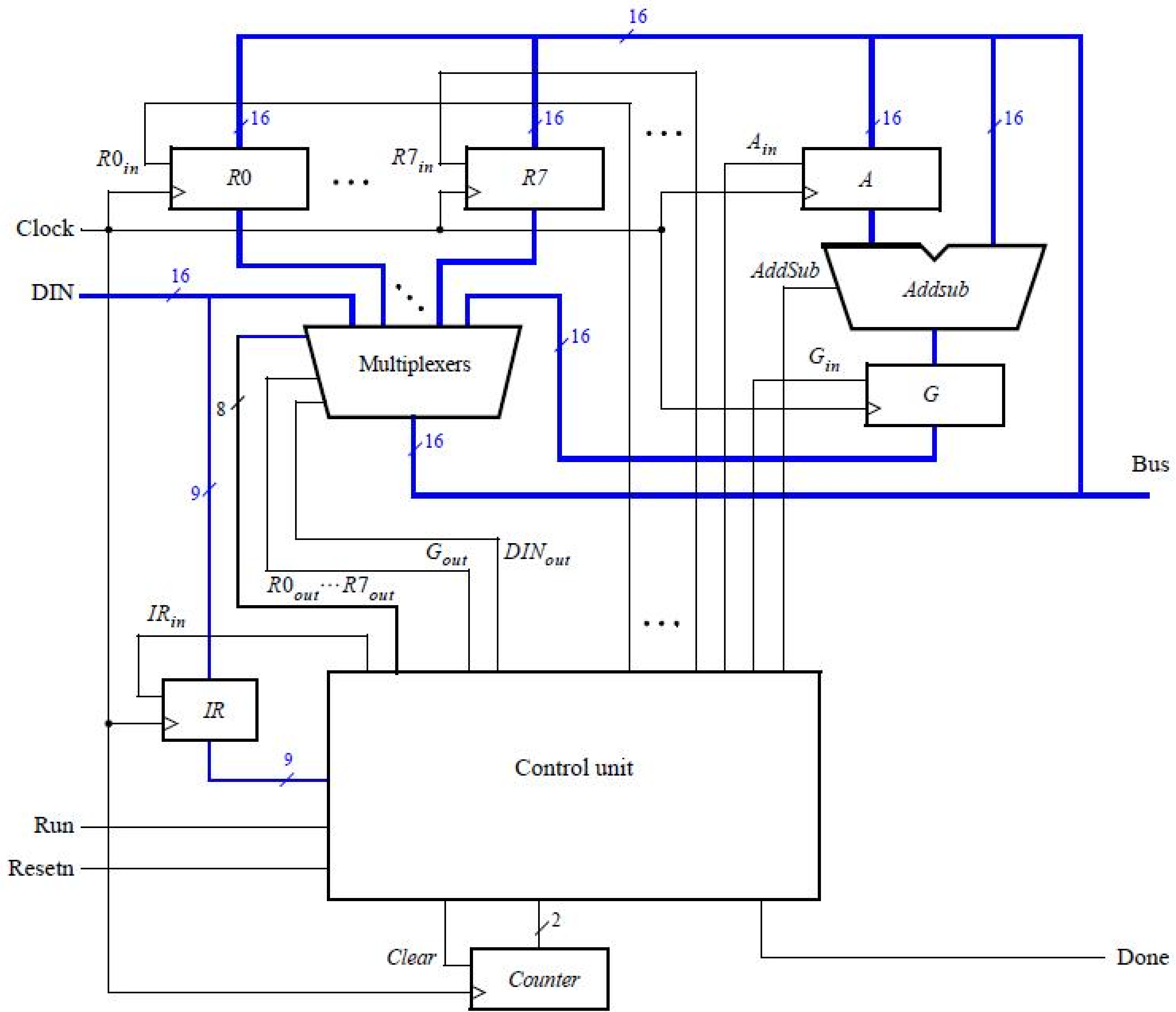


- RALU: register arithmetic-logic unit
- controller: de obicei, un CROM care memorează secvenţe de semnale de control (stări) pentru diferite operaţii (instrucţiuni)

Exemplu

- un sistem digital (procesor) care poate executa diferite instrucțiuni, în funcție de comenzile date de CU (control unit)
- 8 registre pe 16 biți
- 4 operații: aritmetice şi transfer de date

Operation	Function performed
mv Rx,Ry	$Rx \leftarrow [Ry]$
mvi $Rx,\#D$	$Rx \leftarrow D$
add Rx, Ry	$Rx \leftarrow [Rx] + [Ry]$
sub Rx, Ry	$Rx \leftarrow [Rx] - [Ry]$


16
R0in
R0
R7in
R7
Ain
A
Clock
DIN
AddSub
Addsub
Multiplexers
Gin
G
8
9
Bus
Gout
DINout
R0out ··· R7out
IRin
IR
Control unit
Run
Resetn
2
Clear
Counter
Done

Detalii datapath

- IR – registrul de instrucţiuni – format IIIXXXYYY
  - 3 biţi pentru codul instrucţiunii pentru completarea ulterioară
  - XXX şi YYY: adresele pe 3 biţi pentru RX şi RY
- A şi G – registre suplimentare necesare pentru efectuarea operaţiilor
- cu ajutorul multiplexorului se implementează un bus de date tristate

Executarea instrucţiunilor

- operațiile aritmetice se execută în 4 ciclii (perioade ale ceasului)
- operațiile de transfer se execută în 2 cicli
- T0 este identic: $IRin$

	T_1	T_2	T_3
(mv): I_0	RY_{out}, RX_{in}, *Done*		
(mvi): I_1	DIN_{out}, RX_{in}, *Done*		
(add): I_2	RX_{out}, A_{in}	RY_{out}, G_{in}	G_{out}, RX_{in}, *Done*
(sub): I_3	RX_{out}, A_{in}	RY_{out}, G_{in}, *AddSub*	G_{out}, RX_{in}, *Done*

Descrierea Verilog

```
always @(Tstep_Q or I or Xreg or Yreg)
begin
. . . specify initial values
case (Tstep_Q)
2'b00: // store DIN in IR in time step 0
begin
IRin = 1'b1;
end
2'b01: //define signals in time step 1
...
2'b10: //define signals in time step 2
...
2'b11: //define signals in time step 3
...

endcase
end
```

Observaţie. case în case...

```
module proc (DIN, Resetn, Clock, Run, Done, BusWires);
input [15:0] DIN;
input Resetn, Clock, Run;
output Done;
output [15:0] BusWires;
. . . declare variables
wire Clear = . . .
upcount Tstep (Clear, Clock, Tstep_Q);
assign I = IR[1:3];
dec3to8 decX (IR[4:6], 1'b1, Xreg);
dec3to8 decY (IR[7:9], 1'b1, Yreg);

always @(Tstep_Q or I or Xreg or Yreg)
begin
..
end

regn reg_0 (BusWires, Rin[0], Clock, R0);
. . . instantiate other registers and the adder/subtracter unit
. . . define the bus
endmodule
```

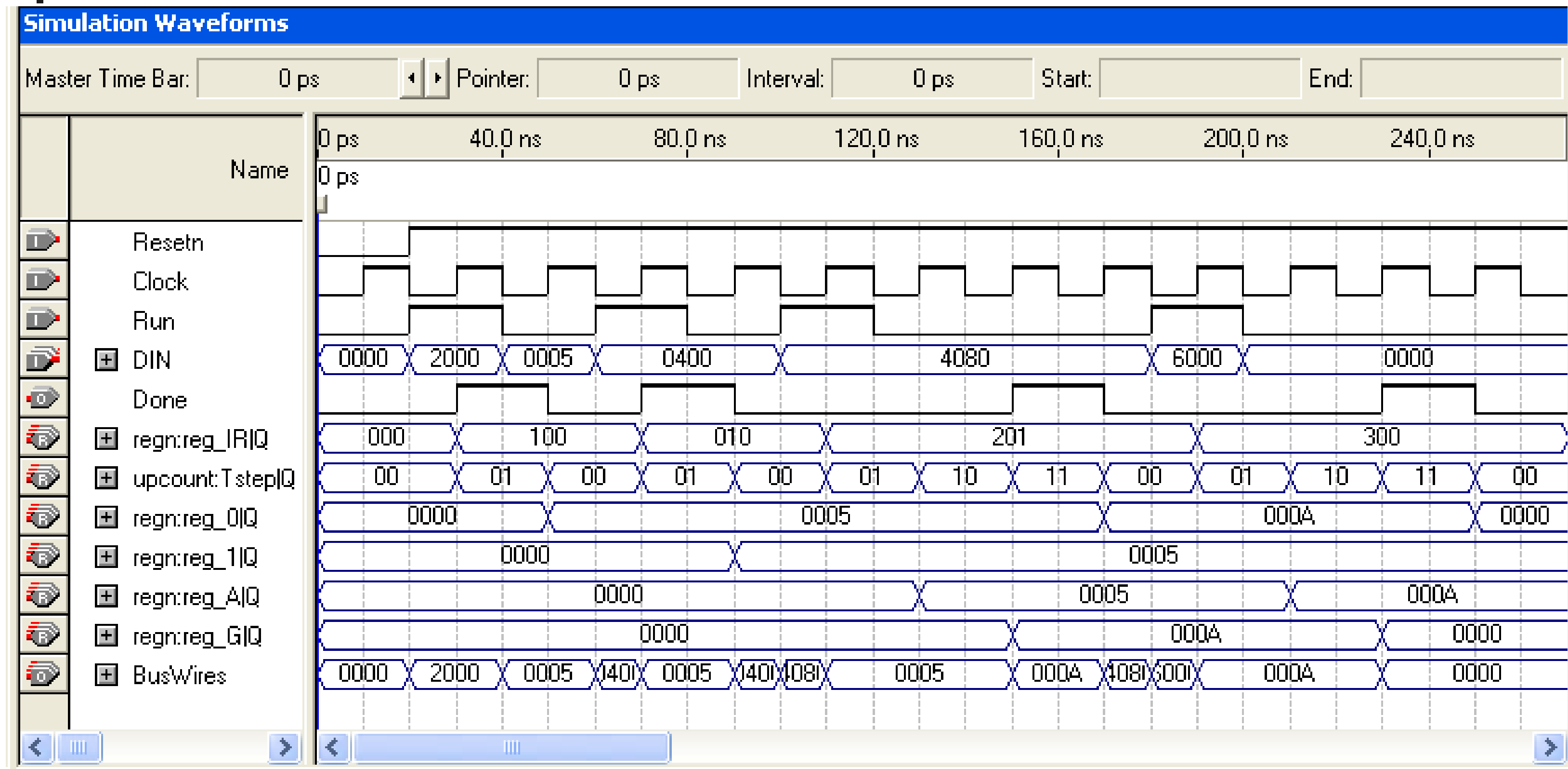
descrierea numaratorului

```
module upcount(Clear, Clock, Q);
input Clear, Clock;
output [1:0] Q;
reg [1:0] Q;
always @(posedge Clock)
if (Clear)
Q <= 2'b0;
else
Q <= Q + 1'b1;
endmodule
```

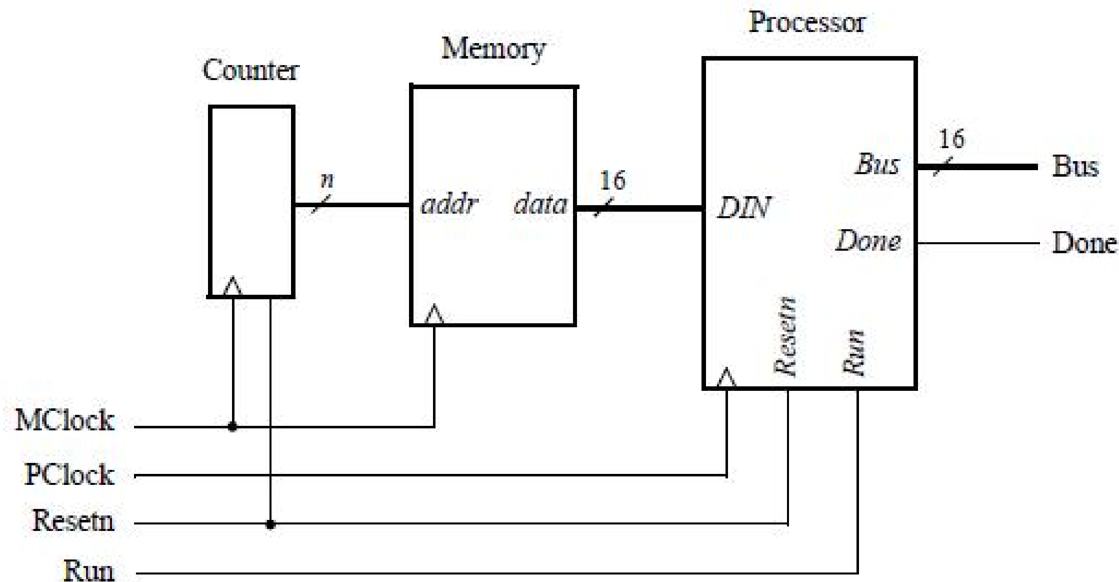
```
module dec3to8(W, En, Y); // DESCRIEREA DECODOARELOR
input [2:0]W;
input En;
output [0:7] Y;
reg [0:7] Y;
always @(W or En)
begin
if (En == 1)
case (W)
3'b000: Y = 8'b10000000;
3'b001: Y = 8'b01000000;
3'b010: Y = 8'b00100000;
3'b011: Y = 8'b00010000;
3'b100: Y = 8'b00001000;
3'b101: Y = 8'b00000100;
3'b110: Y = 8'b00000010;
3'b111: Y = 8'b00000001;
endcase
else
Y = 8'b00000000;
end
endmodule
```

```
module regn(R, Rin, Clock, Q);
parameter n = 16;
input [n-1:0] R;
input Rin, Clock;
output [n-1:0] Q;
reg [n-1:0] Q;
always @(posedge Clock)
if (Rin)
Q <= R;
endmodule
```

simularea procesorului



Conectarea procesorului cu o memorie

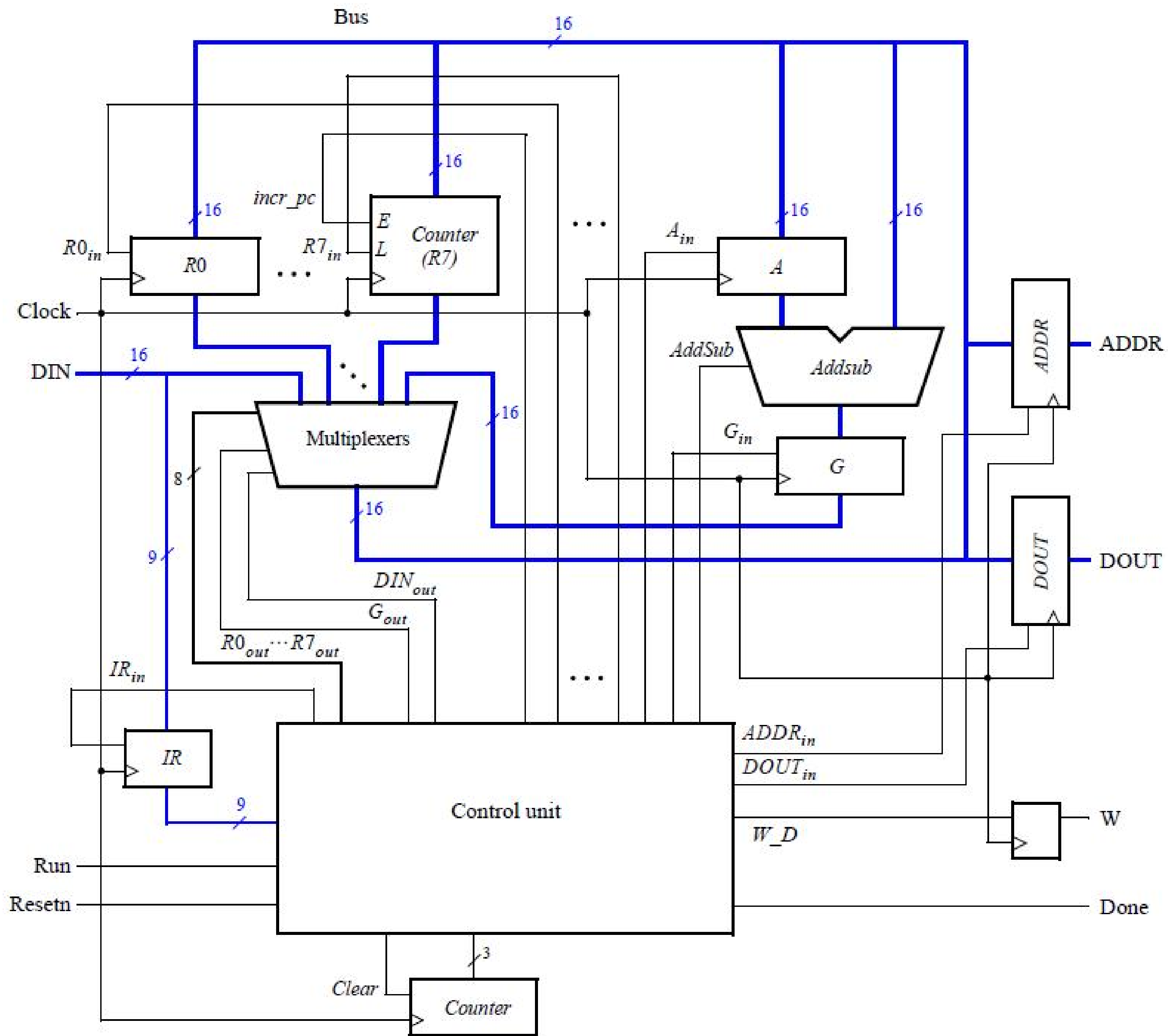


- numărătorul este necesar pentru adresarea memoriei

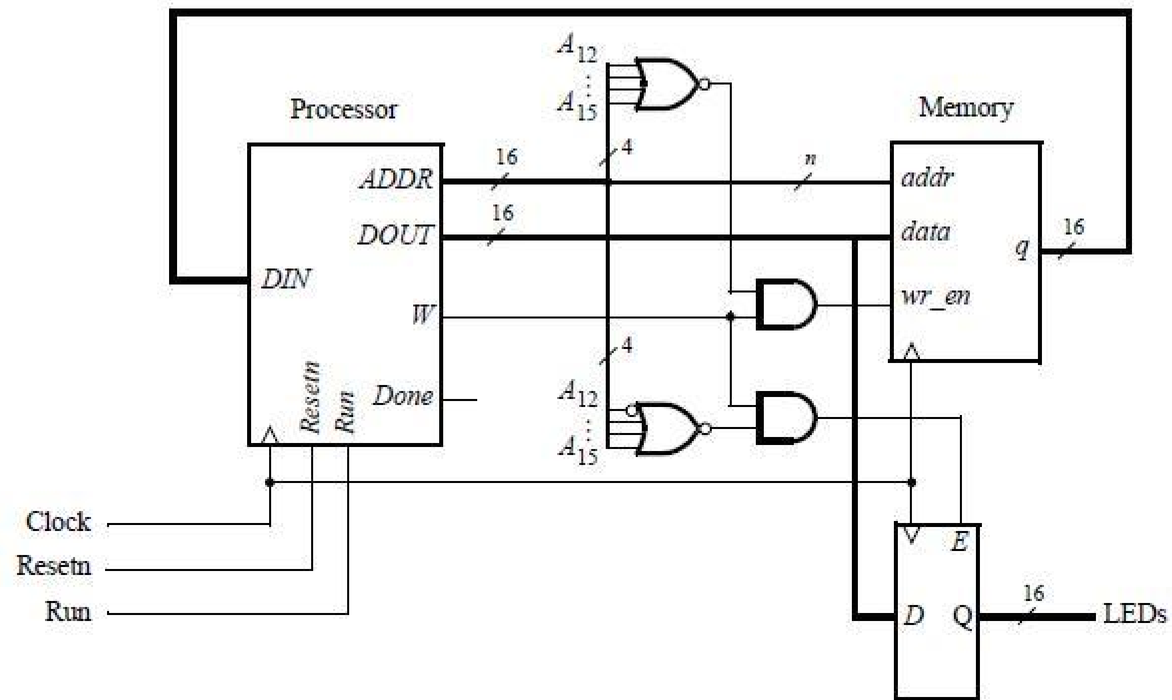
Variantă mai performantă

- procesorul poate folosi memoria externă fără a mai avea nevoie de numărător
- adresarea indirectă: [Ry] este o adresă
- deplasarea condiţionată: *move if not zero*

Operation	Function performed
ld Rx,$[Ry]$	$Rx \leftarrow [[Ry]]$
st Rx,$[Ry]$	$[Ry] \leftarrow [Rx]$
mvnz Rx, Ry	if G != 0, $Rx \leftarrow [Ry]$


Bus
16
16
incr_pc
E
L
Counter
(R7)
R0in
R0
R7in
16
16
Ain
A
Clock
AddSub
Addsub
ADDR
ADDR
DIN
16
16
Multiplexers
Gin
G
8
16
9
DOUT
DOUT
DINout
Gout
R0out ··· R7out
IRin
IR
ADDRin
DOUTin
9
Control unit
W
W_D
Run
Resetn
Done
3
Clear
Counter

Conectarea procesorului cu memoria



- Arhitectura microprocesoarelor
- Proiect 2
- Sisteme cu autoorganizare
- Structura microsistemelor

Recapitulare (posibile subiecte examen la test)

- algebră logică
- aritmetică binară
- circuite combinaţionale
- circuite CMOS
- minimizarea funcţiilor logice
- memorii
- automate
- sisteme de ordin 3
- Verilog