ALU – 64 biți

-Proiect CN-

Hărdălău Elvis Costinel

Iorga Răzvan Vlad

An II – 2C.2.1

Unitatea aritmetică logică (ALU) este un circuit electronic digital complex care poate efectua operații aritmetice și logice. Constructiv, în calculator, ALU este un bloc fundamental al unității centrale de procesare (prelucrare) UCP (engleză CPU). Unitatea logică aritmetică este utilizată pentru a efectua transformări logice și aritmetice pe operanzii necesari, adesea comenzi sau coduri de numere. După finalizarea acțiunii, rezultatul este returnat dispozitivului de stocare pentru a fi utilizat în următoarele calcule.

Platformă hardware ALU:

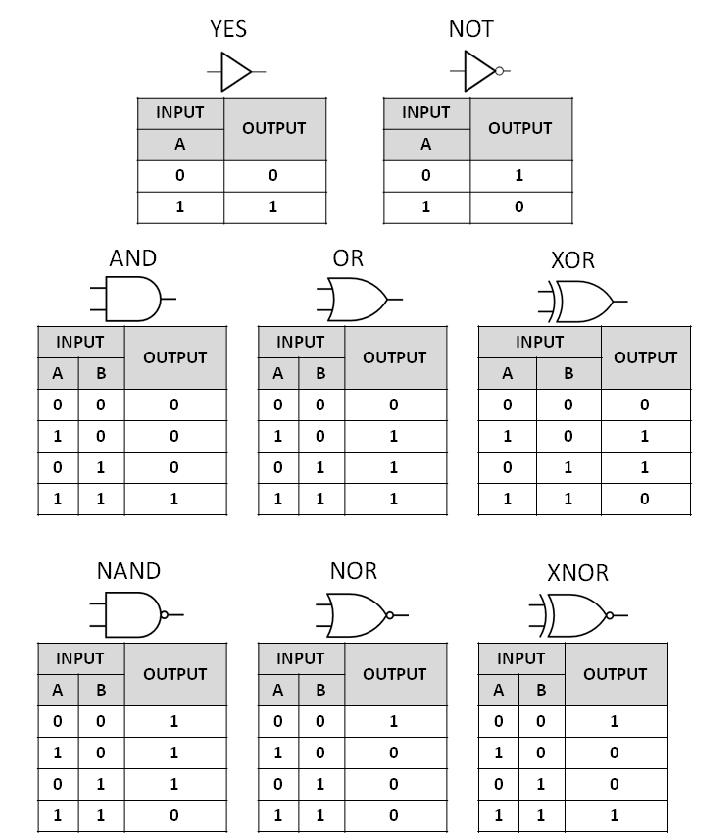
A paper with a diagram

Description automatically generated

A paper with writing on it

Description automatically generated

* ***Logic Unit*** – Unitate specializată realizării operațiilor logice între cei 2 operanzi pe 64 de biți.
* AND
* OR
* XOR
* NOT A
* NOT B
* NAND
* NOR
* XNOR



* AND

**module** bitwise\_and (

**input** in\_0,

**input** in\_1,

**output** and\_

);

**assign** and\_ = in\_0 & in\_1;

**endmodule**

**module** and\_wordgate #(**parameter** w = 64)(

**input** [w-1:0]in\_0,

**input** [w-1:0]in\_1,

**output** [w-1:0]AND

);

**generate**

**genvar** i;

**for**(i = 0 ; i < w ; i = i + 1) **begin**: and\_gates

bitwise\_and gate(.in\_0(in\_0[i]), .in\_1(in\_1[i]), .and\_(AND[i]));

**end**

**endgenerate**

**endmodule**

* OR

module bitwise\_or (

input in\_0,

input in\_1,

output or\_

);

assign or\_ = in\_0 | in\_1;

endmodule

**module** or\_wordgate #(**parameter** w = 64)(

**input** [w-1:0]in\_0,

**input** [w-1:0]in\_1,

**output** [w-1:0]OR

);

**generate**

**genvar** i;

**for**(i = 0 ; i < w ; i = i + 1) **begin**: or\_gates

bitwise\_or gate(.in\_0(in\_0[i]), .in\_1(in\_1[i]), .or\_(OR[i]));

**end**

**endgenerate**

**endmodule**

* XOR

**module** bitwise\_xor (

**input** in\_0,

**input** in\_1,

**output** xor\_

);

**assign** xor\_ = in\_0 ^ in\_1;

**endmodule**

**module** xor\_wordgate #(**parameter** w = 64)(

**input** [w-1:0]in\_0,

**input** [w-1:0]in\_1,

**output** [w-1:0]XOR

);

**generate**

**genvar** i;

**for**(i = 0 ; i < w ; i = i + 1) **begin**: xor\_gates

bitwise\_xor gate(.in\_0(in\_0[i]), .in\_1(in\_1[i]), .xor\_(XOR[i]));

**end**

**endgenerate**

**endmodule**

* NOT A or NOT B

**module** bitwise\_not (

**input** in,

**output** not\_

);

**assign** not\_ = ~in;

**endmodule**

**module** not\_wordgate #(**parameter** w = 64)(

**input** [w-1:0]in,

**output** [w-1:0]NOT

);

**generate**

**genvar** i;

**for**(i = 0 ; i < w ; i = i + 1) **begin**: not\_gates

bitwise\_not gate(.in(in[i]), .not\_(NOT[i]));

**end**

**endgenerate**

**endmodule**

* NAND

**module** bitwise\_nand (

**input** in\_0,

**input** in\_1,

**output** nand\_

);

**assign** nand\_ = ~(in\_0 & in\_1);

**endmodule**

**module** nand\_wordgate #(**parameter** w = 64)(

**input** [w-1:0]in\_0,

**input** [w-1:0]in\_1,

**output** [w-1:0]NAND

);

**generate**

**genvar** i;

**for**(i = 0 ; i < w ; i = i + 1) **begin**: nand\_gates

bitwise\_nand gate(.in\_0(in\_0[i]), .in\_1(in\_1[i]), .nand\_(NAND[i]));

**end**

**endgenerate**

**endmodule**

* NOR

**module** bitwise\_nor (

**input** in\_0,

**input** in\_1,

**output** nor\_

);

**assign** nor\_ = ~(in\_0 | in\_1);

**endmodule**

**module** nor\_wordgate #(**parameter** w = 64)(

**input** [w-1:0]in\_0,

**input** [w-1:0]in\_1,

**output** [w-1:0]NOR

);

**generate**

**genvar** i;

**for**(i = 0 ; i < w ; i = i + 1) **begin**: nor\_gates

bitwise\_nor gate(.in\_0(in\_0[i]), .in\_1(in\_1[i]), .nor\_(NOR[i]));

**end**

**endgenerate**

**endmodule**

* XNOR

**module** bitwise\_xnor (

**input** in\_0,

**input** in\_1,

**output** xnor\_

);

**assign** xnor\_ = ~(in\_0 ^ in\_1);

**endmodule**

**module** xnor\_wordgate #(**parameter** w = 64)(

**input** [w-1:0]in\_0,

**input** [w-1:0]in\_1,

**output** [w-1:0]XNOR

);

**generate**

**genvar** i;

**for**(i = 0 ; i < w ; i = i + 1) **begin**: xnor\_gates

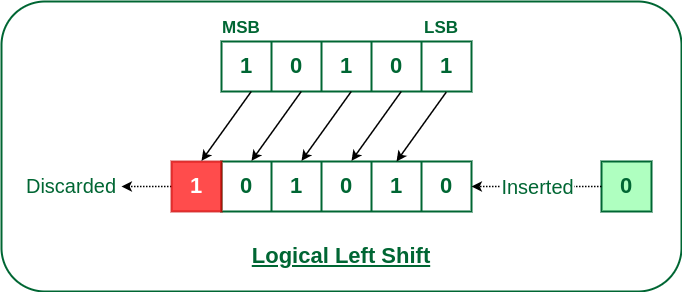
bitwise\_xnor gate(.in\_0(in\_0[i]), .in\_1(in\_1[i]), .xnor\_(XNOR[i]));

**end**

**endgenerate**

**endmodule**

* ***Shift/Rotate Unit*** – Unitate specializată realizării operațiilor de shiftare.
* Logic Left Shift
* Logic Right Shift
* Arithmetic Left Shift
* Arithmetic Right Shift
* Left Rotate
* Right Rotate



module sll (

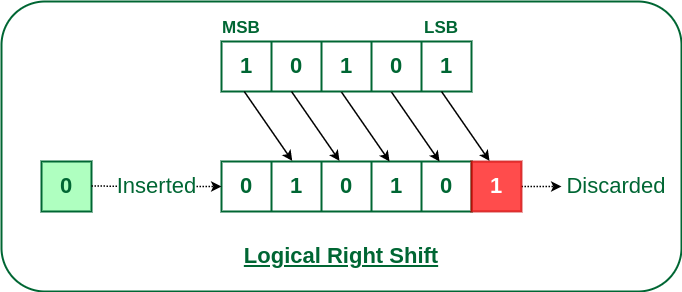
input [127:0]in,

output [127:0]out

);

assign out = {in[126:0], 1'b0};

endmodule



module srl (

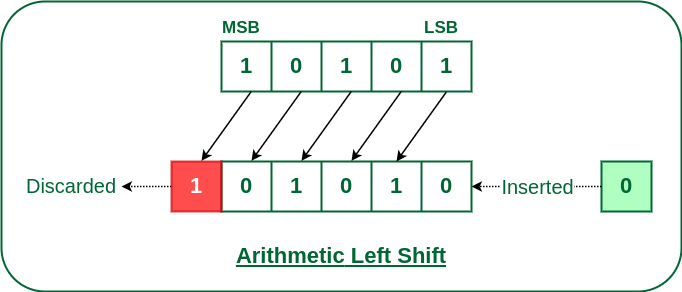
input [127:0]in,

output [127:0]out

);

assign out = {1'b0, in[127:1]};

endmodule



**module** sla (

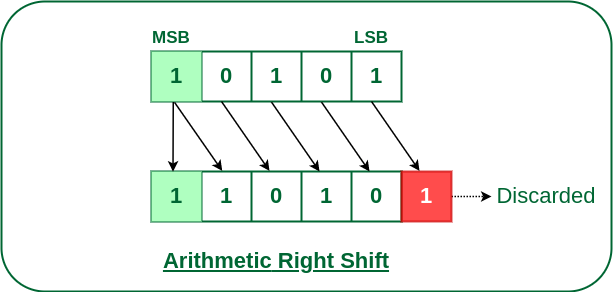
**input** [127:0]in,

**output** [127:0]out

);

**assign** out = {in[126:0], 1'b0};

**endmodule**



**module** sra (

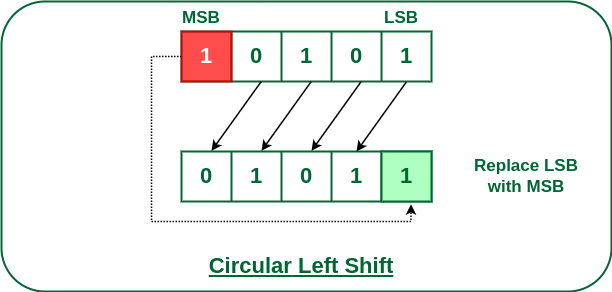
**input** [127:0]in,

**output** [127:0]out

);

**assign** out = {in[127], in[127:1]};

**endmodule**



**module** rotl (

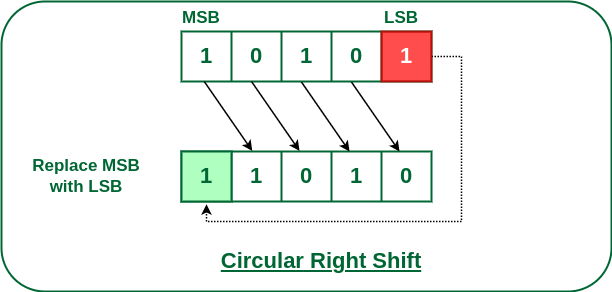
**input** [127:0]in,

**output** [127:0]out

);

**assign** out = {in[126:0], in[127]};

**endmodule**



**module** rotr (

**input** [127:0]in,

**output** [127:0]out

);

**assign** out = {in[0], in[127:1]};

**endmodule**

* Shift/Rotate Unit implementat printr-un registru secvențial parametrizat prin lațime și prin numărul de biți shiftați:

**module** sh\_reg #(**parameter** w = 64, **parameter** d = 1)(

**input** clk,

**input** rst\_b,

**input** load,

**input** sh,

**input** [2:0]sh\_mode,

**input** [d-1:0]sh\_in,

**input** [w-1:0]in,

**output** **reg** [w-1:0]q

);

**reg** [w-1:0]qq;

**initial** **begin**

qq = 0;

**end**

**always** @(**posedge** clk) **begin**

**if**(load)

qq = in;

**if**(sh)

**case**(sh\_mode)

3'b000: qq = {qq[w-1-d:0], {d{1'b0}}}; // logic left shift

3'b001: qq = {{d{1'b0}}, qq[w-1:d]}; // logic right shift

3'b010: qq = {qq[w-1-d:0], {d{1'b0}}}; // arithmetic left shift

3'b011: qq = {{d{qq[w-1]}}, qq[w-1:d]}; // arithmetic right shift

3'b100: qq = {qq[w-1-d:0], qq[w-1:w-d]}; // left rotate

3'b101: qq = {qq[d-1:0], qq[w-1:d]}; // right rotate

3'b110: qq = {qq[w-1-d:0], sh\_in}; // left shift sh\_in

3'b111: qq = {sh\_in, qq[w-1:d]}; // right shift sh\_in

**endcase**

**end**

**always** @(**posedge** clk **or** **negedge** rst\_b) **begin**

**if**(!rst\_b)

q <= {w{1'b0}};

**else**

q <= qq;

**end**

**endmodule**

* ***Arithmetic Unit*** – Unitate specializată realizării operațiilor aritmetice (+, -, \*, /) între cei 2 operanzi pe 64 de biți.

**module** arithmetic\_unit (

**input** clk,

**input** bgn,

**input** rst\_b,

**input** [1:0]opcode, //00 - addition, 01 - substraction, 10 - multiplication, 11 - division

**input** [63:0]inbus,

**output** stop,

**output** [63:0]outbus

);

**wire** c0;

**wire** c1;

**wire** c2;

**wire** c3;

**wire** c4;

**wire** c5;

**wire** c6;

**wire** c7;

**wire** c8;

**wire** c9;

**wire** Q\_lsb;

**wire** Q\_msb;

**wire** q\_1;

**wire** count63;

**wire** A\_lsb;

**wire** A\_msb;

**wire** [63:0]M;

**wire** [63:0]A;

**wire** [63:0]out\_parallel;

**wire** [63:0]out\_xor;

reg\_M uut\_M (

.clk(clk),

.rst\_b(rst\_b),

.c0(c0),

.inbus(inbus),

.q(M)

);

reg\_A uut\_A (

.clk(clk),

.rst\_b(rst\_b),

.c0(c0),

.c2(c2),

.c4(c4),

.c5(c5),

.c8(c8),

.sum(out\_parallel),

.A\_lsb(A\_lsb),

.A\_msb(A\_msb),

.Q\_msb(Q\_msb),

.q(A),

.outbus(outbus)

);

reg\_Q uut\_Q (

.clk(clk),

.rst\_b(rst\_b),

.c1(c1),

.c4(c4),

.c6(c6),

.c8(c8),

.c9(c9),

.A\_lsb(A\_lsb),

.Q\_lsb(Q\_lsb),

.Q\_msb(Q\_msb),

.inbus(inbus),

.outbus(outbus)

);

reg\_Q\_1 uut\_Q\_1 (

.clk(clk),

.rst\_b(rst\_b),

.c0(c0),

.c4(c4),

.Q\_lsb(Q\_lsb),

.q(q\_1)

);

counter uut\_counter (

.clk(clk),

.rst\_b(rst\_b),

.c0(c0),

.c4(c4),

.count63(count63)

);

control\_unit uut\_control\_unit (

.clk(clk),

.rst\_b(rst\_b),

.bgn(bgn),

.opcode(opcode),

.q\_1(q\_1),

.q0(Q\_lsb),

.count63(count63),

.a63(A\_msb),

.c0(c0),

.c1(c1),

.c2(c2),

.c3(c3),

.c4(c4),

.c5(c5),

.c6(c6),

.c7(c7),

.c8(c8),

.c9(c9),

.stop(stop)

);

xor\_wordgate uut\_xor (

.in\_0({64{c3}}),

.in\_1(M),

.XOR(out\_xor)

);

ml\_csea\_8 uut\_adder (

.x(out\_xor),

.y(c7 ? inbus : A),

.c\_in(c3),

.z(out\_parallel)

);

**endmodule**

Modul ALU – 64 biți:

**module** ALU\_64 (

**input** clk,

**input** rst\_b,

**input** bgn,

**input** [63:0]in\_0,

**input** [63:0]in\_1,

**input** [4:0]sel,

**output** stop,

**output** [63:0]out

);

**wire** [63:0]out\_arithmetic;

**wire** [63:0]out\_logic;

**wire** [127:0]out\_shift;

arithmetic\_unit uut\_arithmetic (

.clk(clk),

.bgn(bgn),

.rst\_b(rst\_b),

.opcode(sel[1:0]),

.inbus(in\_0),

.stop(stop),

.outbus(out\_arithmetic)

);

logic\_unit uut\_logic (

.in\_0(in\_0),

.in\_1(in\_1),

.sel(sel[2:0]),

.out(out\_logic)

);

shift\_rotate\_unit uut\_shift (

.in({in\_1, in\_0}),

.sel(sel[2:0]),

.out(out\_shift)

);

mux4to1 uut\_mux (

.in\_0(out\_arithmetic),

.in\_1(out\_logic),

.in\_2(out\_shift[63:0]),

.in\_3({64{1'bz}}),

.sel(sel[4:3]),

.out(out)

);

**endmodule**