|  |
| --- |
| https://www.bme.hu/sites/default/files/mediakit/bme_logo_nagy.jpg |
| ALU egység tervezése |
| Dokumentáció |
|  |
| **Németh Noel – Hesz Gergely** |
| **2016. 05. 29.** |

|  |
| --- |
| A Budapest Műszaki és Gazdaságtudományi Egyetem Logikai tervezés tárgy házi feladatához tartozó dokumentáció |

Tartalom

[Specifikáció 2](#_Toc452298343)

[Az eszköz működésének leírása 3](#_Toc452298344)

[Felépítés 4](#_Toc452298345)

[Modulok részletes leírása 5](#_Toc452298346)

[ALU\_unit.v 5](#_Toc452298347)

[CPU.v 6](#_Toc452298348)

[Szintézis és implementációs eredmények 8](#_Toc452298349)

[Teszt és szimulációs eredmények 10](#_Toc452298350)

[Függelék 15](#_Toc452298351)

# Specifikáció

A modul egy ALU egységet valósítson meg, amellyel különböző aritmetikai és bitműveletek végezhetők. A modul belső regisztereket tartalmazzon, amelyekből az ALU egység az operandusait veszi és az eredményt tárolja. Azt, hogy melyik regiszter használandó az opcode-ban legyen kódolva.

Az eszköz a következő műveletek elvégzésére legyen képes:

* Összeadás – ADD
* Kivonás – SUB
* Bit shiftelés – LSHIFT, RSHIFT
* Kizáró VAGY művelet – XOR
* Komparálás – CMP
* Bitenkénti ÉS művelet – AND
* Negált ÉS művelet – NAND
* Bitenkénti VAGY művelet – OR
* negált VAGY művelet – NOR

A rendszer támogassa a kaszkádosítást.

# Az eszköz működésének leírása

A rendszer a specifikációban részletezett műveleteket elvégzésére képes. Ezek a kódban a következő definiált paraméterekkel hivatkozhatók: ADD, SUB, LSHIFT, RSHIFT, XOR, CMP, AND, NAND, OR, NOR.

A rendszer egy regisztertömböt tartalmaz, amely 8 regisztert tartalmaz. Ebből a legalsó (0-ás) regiszter az AKKU, amelyből veszi az eszköz az egyik operandusát és amelybe menti a művelet eredményét. A rendszer kimenetén addig kiolvasható az előző eredmény értéke, amíg új műveletet nem végez. A második operandust az opcode-ban kódolt regiszterből veszi az eszköz.

Az eszköz egy zárt rendszert alkot, így a regiszterei „kívülről” nem látszanak. Azoknak értéket adni load – töltő utasítással lehetséges. Az eszköznek tehát 8 bitnyi bemeneti adatvonala van, 7 bitnyi opcode illetve 1 bit jelzi a load utasítást. Az opcode 4 alsó bitje kódolja a megvalósítandó utasítást, míg a felső 3 bit kódolja, a második operandus regiszterét. Annak érdekében, hogy az AKKU-n kívüli többi regiszter is „látszódjon” kívülről egy CPY másoló utasítást is megvalósít az eszköz, amely az AKKU-ba másolja az adott regiszter értékét. Ezzel az utasítással lehetséges a regiszterek kiolvasása.

A kaszkádosítás támogatására az eszköz cin – carry bemenet illetve cout – carry kimenet használandó. Összeadás illetve kivonásnál ezek tényleges carry-ként működnek illetve shiftelésnél a beshiftelés értéke állítható a cin-nel illetve a kishiftelt bit olvasható a cout kimeneten.

Az egységnek van egy ce – chip enable jele, amely engedélyezi a műveletvégzést.

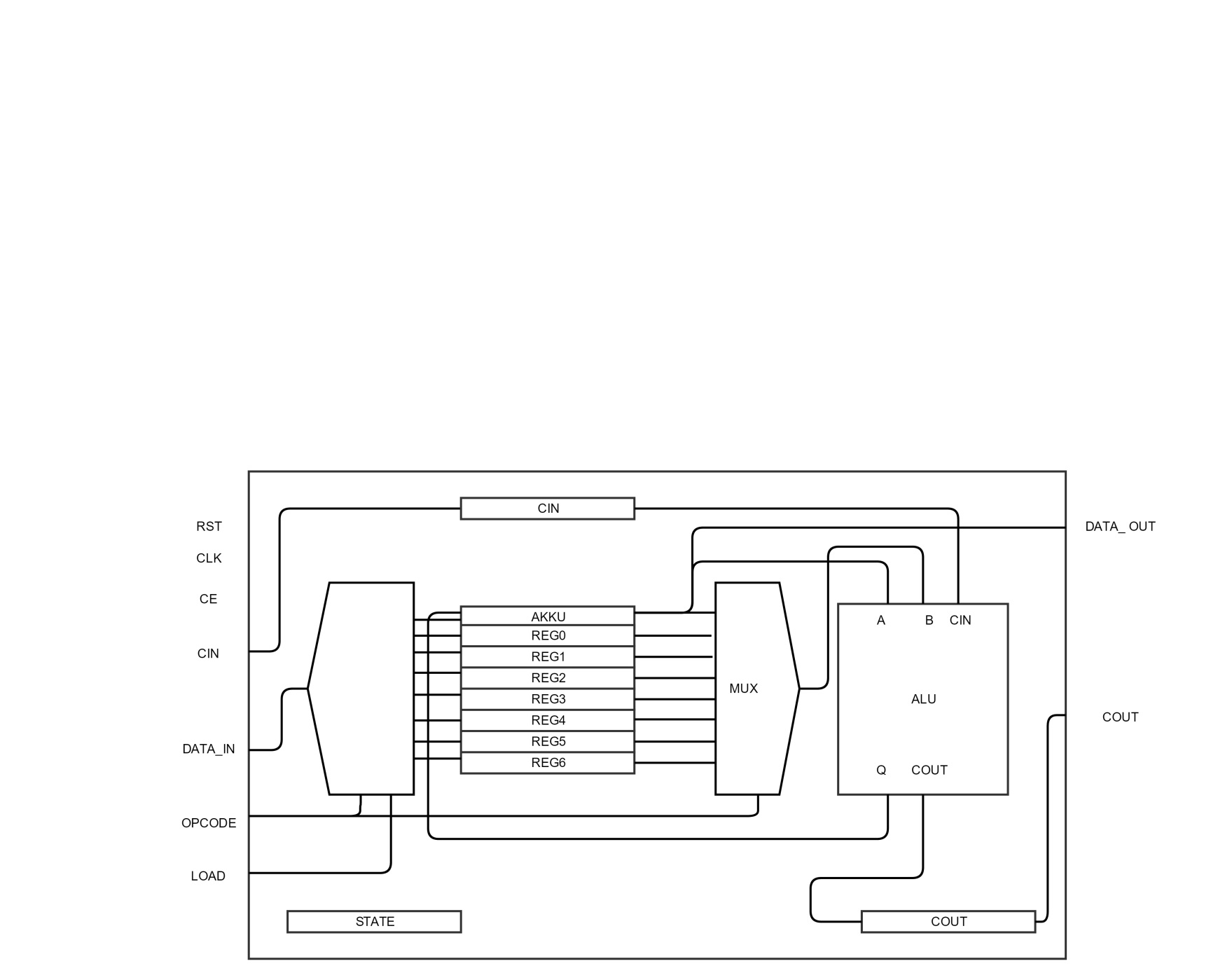
Az eszköznek 8 bites adat bemenete van.

Amennyiben ce==1 és load==1 akkor az opkódban megadott regiszter értékét tölti fel a data\_in buszon lévő értékkel. Ha ce==1 és load==0 akkor történik műveletvégzés az opcodeban kódolt regiszterrel és művelettel. Az ALU egység maga tisztán kombinációs logika. A mintavételezést követően elkezdődik a műveletvégzés, annak értéke viszont csak egy órajel ciklus múltán jelenik meg a kimeneten. Ehhez a működéshez az ALU köré épített kezdetleges CPU kétféle állapotban lehet: IDLE, és OPERATION. IDLE állapotban mintavételezi az eszköz a bemeneteit. Amennyiben műveletvégzés történik, az eszköz átkerül OPERATION állapotba, amely egy órajelciklusig tart. Ennek a végén történik a kimeneti regiszterek írása. Ezeknek az állapotoknak a felhasználásával kerülhető el, hogy az ALU új értéket kapjon műveletvégzés közben illetve ezek használata teszi lehetővé, hogy a kimeneten már csak érvényes adat legyen illetve definiált időzítéssel lehessen kiolvasni azt.

A chip enable jel nulla értékén az egység nem végez műveletet, kimenetein a legutolsó művelet eredménye olvasható.

# Felépítés

Az eszköz két modulból áll. Az egyik a tisztán kombinációs hálózatot tartalmazó ALU egység illetve az eköré épített kezdetleges CPU-nak nevezett modul. Ez tartalmazza a regisztereket, némi vezérlő logikát illetve az ALU egység bemenetei és kimenetei időzítését. A rendszer blokkvázlata a következő ábrán látható:



Az ábrán látható az ALU egység illetve a belső regiszterek. Az ábra csak a logikai egységeket mutatja, arról még hiányzik a vezérlő logika, amelyet az RST, CLK és CE bemenetek is befolyásolnak. Természetesen a regiszterek betöltése, így aztán a ki-bemenetek írása olvasása ezeknek a jeleknek a felhasználásával történik.

# Modulok részletes leírása

## ALU\_unit.v

Ez tartalmazza a megvalósítandó ALU egységet.

A bemenetei/kimenetei:

* opA[7:0]: az első operandus
* opB[7:0]: a második operandus
* outData[7:0]: az ALU adatkimenete
* opcode[3:0]: Ez határozza meg, hogy melyik műveletet kell elvégeznie az ALU-nak
* cin: carry input
* cout: carry output

A modul leírás az opcode dekódolásához szükséges paraméterek definiálásával kezdődik. Ezen kívül egy darab always blokk található benne, amely leírja az ALU egység működését. Az érzékenységi lista a két operandust illetve a cin és az opcode-ot tartalmazza, órajelet nem, így ebből egy tisztán kombinációs hálózat fog szintetizálódni. Az always blokkban először egy pár if- else is található, amelyben a feltétel az opcode lehetséges értékeinek vizsgálata. Ez azért volt szükséges, hogy ahol a cout\_regiszter is értéket kap, azok az esetek ne kerüljenek bele a második részben leírt nagyobb case szerkezetbe. A case elem egy multiplexert hoz létre, amely dekódolja az utasítást és annak megfelelően választja ki az egyes műveleteket elvégző kombinációs logikát.

Az adat illetve cout kimenetet két regiszter tartalmazza, amelyek egyszerűen ki vannak kötve a kimenetre.

A teljes kód a függelékben található.

## CPU.v

Ebben a modulban vannak definiálva az operandus regiszterek illetve itt van példányosítva az ALU modul.

A bemenetei/kimenetei:

* clk: órajel
* rst: reset jel
* data\_in[7:0]: 8 bites adat bemenet
* opcode[6:0]: 7 bites opcode – 3 bit regiszter kiválasztó, 4 bit ALU opcode
* load: betöltést jelző bit
* cin: carry input
* cout: carry output
* ce: Chip Enable
* data\_out[7:0]: adat kimenet

A modul az operandus regiszter tömb definiálásával kezdődik:

reg [7:0] operands [0:7];

Ezt követi néhány belső regiszter és wire deklarálása, köztük a state-tel amely tárolni fogja az eszköz aktuális állapotát. Ehhez két paraméter is deklarálva van.

A logikát egy always blokk valósítja meg. Ennek az első része az összes belső regisztert nulla értékre állít reset jel esetén. A második része egy nagy if- else szerkezetet tartalmaz. Amennyiben IDLE az aktuális állapot, akkor történhet máveletvégzés, máskülönben, vagyis ha OPERATION az az állapot, akkor IDLE-be fog lépni az eszköz és a kimeneti regiszterek értéket kapnak. IDLE állapot esetén amennyiben a load bemenet 1-es az operands tömb opcode-ban leírt elemébe beíródik a data\_in bemenet:

if (state == IDLE)

begin

if (ce==1 && load ==1 )

operands[opcode[6:4]] <= data\_in;

Amennyiben a load 0-s az eszköz mintavételezi a bemeneteit és eltárolja a belső regiszterekben:

else if (ce == 1 && load == 0 )

begin

reg\_selector <= opcode[6:4];

cin\_reg <= cin;

cout\_reg <=cout;

operation <= opcode[3:0];

state <= OPERATION;

end

Mivel ezek a belső regiszterek rá vannak kötve az ALU bemenetére ekkor már elkezdődik a műveletvégzés, tehát a következő órajel ciklusban már kiolvasható az ALU kimenete:

else

begin

operands[0] <= alu\_output\_wire;

cout\_reg <= cout\_wire;

state <=IDLE;

end

A teljes kód CPU modul kódja a függelékben megtalálható.

# Szintézis és implementációs eredmények

A felhasznált modulok:

# Adders/Subtractors : 3

9-bit adder : 1

9-bit adder carry in : 1

9-bit subtractor : 1

# Registers : 74

Flip-Flops : 74

# Comparators : 2

8-bit comparator equal : 1

8-bit comparator greater : 1

# Multiplexers : 11

1-bit 2-to-1 multiplexer : 8

1-bit 4-to-1 multiplexer : 1

8-bit 4-to-1 multiplexer : 1

8-bit 8-to-1 multiplexer : 1

# Xors : 1

8-bit xor2 : 1

Slice Logic Utilization:

Number of Slice Registers: 79 out of 11,440 1%

Number used as Flip Flops: 78

Number used as Latches: 0

Number used as Latch-thrus: 0

Number used as AND/OR logics: 1

Number of Slice LUTs: 144 out of 5,720 2%

Number used as logic: 141 out of 5,720 2%

Number using O6 output only: 133

Number using O5 output only: 0

Number using O5 and O6: 8

Number used as ROM: 0

Number used as Memory: 0 out of 1,440 0%

Number used exclusively as route-thrus: 3

Number with same-slice register load: 3

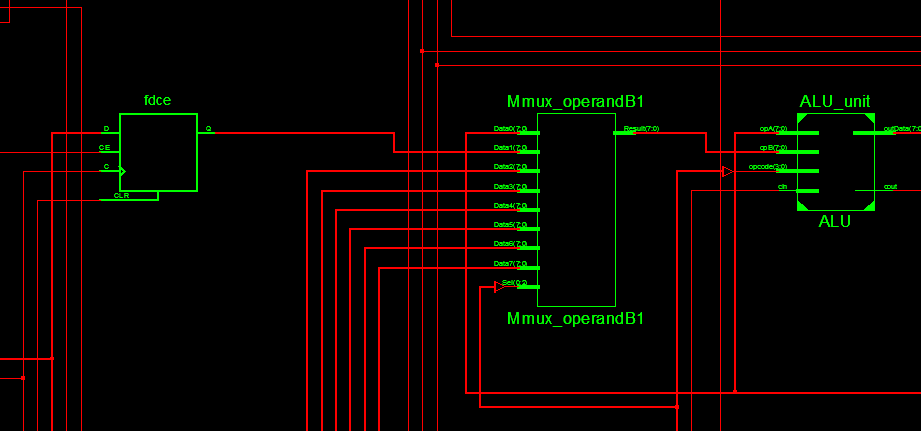
Number with same-slice carry load: 0

Number with other load: 0

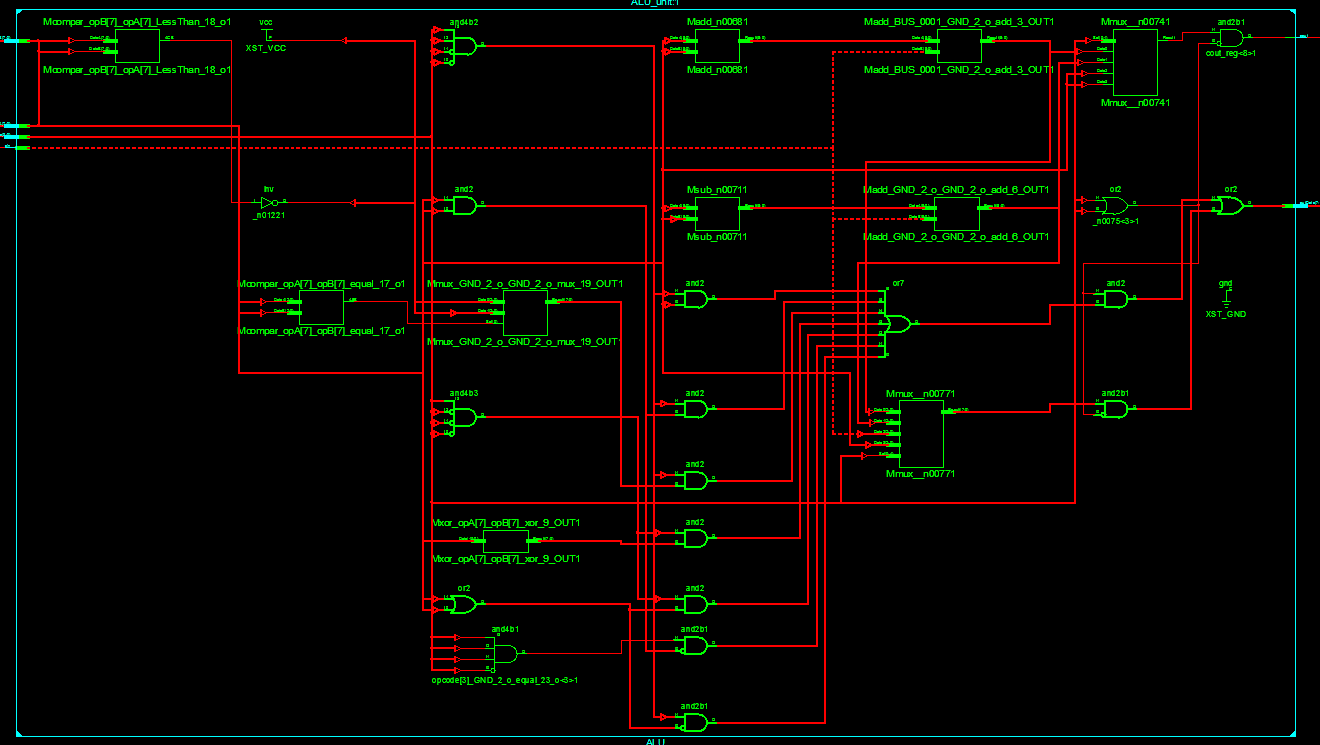
Ami látható, hogy 74 flip-flop található benne, különböző multiplexerek az opcode kiválasztásból adódóan, összeadók és kivonók a műveletvégzéshez, illetve komparátor a komparálás művelethez illetve egy 8-bites xor az XOR művelethez.

165,989 MHz maximális órajel frekvencia használata lehetséges.

Ezeket a primitíveket a schematicon meg is figyelhetjük:



Az ALU egység:



A teljes schematic a Függelékben megtalálható.

# Teszt és szimulációs eredmények

A modulhoz készítettünk egy testbenchet amelyen szimuláltuk annak működését. A teszt, amelyet futtatunk először reset állapotba hozza az eszközt, majd feltölti a regisztereket különböző értékekkel majd azokkal műveleteket végez, hogy minden műveletnek a működését láthassuk.

A teszteléshez használt használt paraméterek:

parameter ADD = 4'h0;

parameter SUB = 4'h1;

parameter LSHIFT = 4'h2;

parameter RSHIFT = 4'h3;

parameter XOR = 4'h4;

parameter CMP = 4'h5;

parameter AND = 4'h6;

parameter NAND = 4'h7;

parameter OR = 4'h8;

parameter NOR = 4'h9;

parameter CPY = 4'ha;

parameter ACCU = 3'b000;

parameter REG0 = 3'b001;

parameter REG1 = 3'b010;

parameter REG2 = 3'b011;

parameter REG3 = 3'b100;

parameter REG4 = 3'b101;

parameter REG5 = 3'b110;

parameter REG6 = 3'b111;

A teszt első két initial blokkja amely reseteli az eszközt illetve órajelet generál:

initial begin

// Initialize Inputs

clk = 1;

rst = 1;

data\_in = 0;

opcode = 0;

cin = 0;

load = 0;

ce = 0;

end

initial #102 rst = 0;

always #5 clk <= ~clk;

Ezt követi a teszt initial blokkja, amely egy ideig resetben hagyja az eszközt majd feltölti a regisztereket:

initial

begin

//Loading registers with valid data

#1002

load = 1;

ce = 1;

opcode = {ACCU , 4'b0000};

data\_in = 8'b00000001;

#10

opcode = {REG0 , 4'b0000};

data\_in = 8'b00000010;

#10

opcode = {REG1 , 4'b0000};

data\_in = 8'b10101010;

#10

opcode = {REG2 , 4'b0000};

data\_in = 8'b00001111;

#10

opcode = {REG3 , 4'b0000};

data\_in = 8'b00000101;

#10

opcode = {REG4 , 4'b0000};

data\_in = 8'b00000100;

#10

opcode = {REG5 , 4'b0000};

data\_in = 8'b01000000;

#10

opcode = {REG6 , 4'b0000};

data\_in = 8'b10000000;

#10

load = 0;

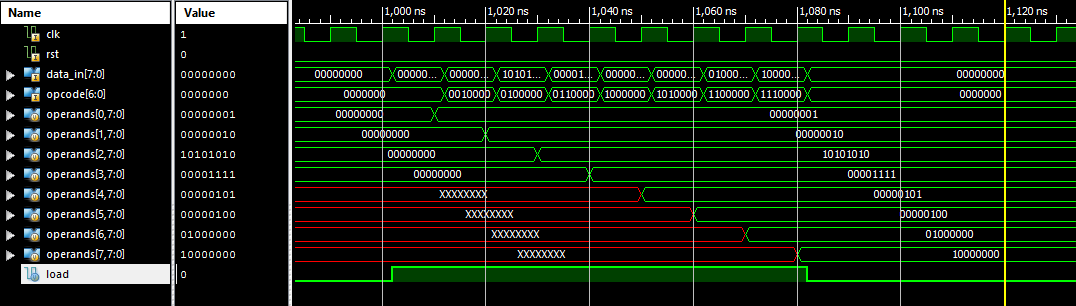
ce = 0;

opcode = 0;

data\_in = 0;

//End of register initialsation

A regiszterek feltöltése a szimulációban:



A teszt elsőrésze mutatja a CE működését:

#100

ce = 1;

opcode = {REG0,ADD}; // = 8'b00000011

#10

ce = 0;

#10

ce = 1;

opcode = {ACCU,SUB }; //Accu-Accu=8'b000000000

#10

ce = 0;

#30

ce=1;

opcode = {REG0,ADD}; // = 8'b00000011

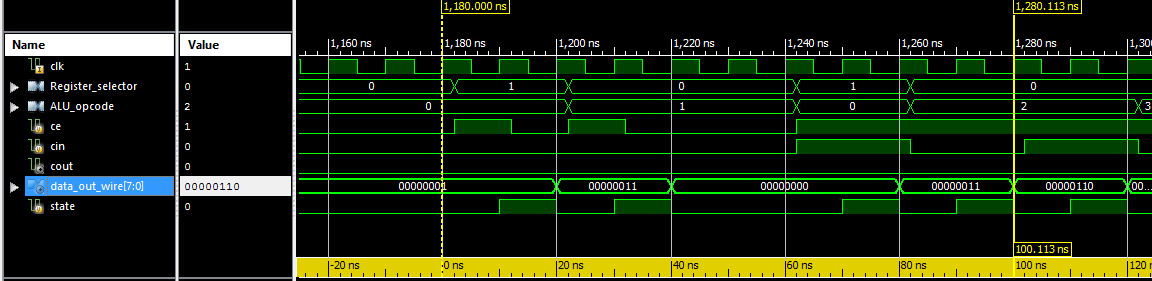
cin = 1; //

#20

opcode = {ACCU,LSHIFT}; // 8'b00000110

cin = 0;

A teszt első része mutatja a ce működését, illetve összeadás kivonást a cin használatával, illetve anélkül, valamint egy balra shiftelést cin=0-val, vagyis egy nullás beshiftelését a legkisebb helyiértékre. A műveletek után található a várt eredmény, amely meg is figyelhető a data\_out\_wire kimeneten.



Ezt követően cin=1-gyel való shiftelést is teszteltünk, majd egy jobbra shiftelést ami a cout-ra ki is írta az előzőleg beshiftelt egyest. Ezt követi egy CPY és két XOR művelet (ugyan azt az értéket kell visszakapni, ez a 8'b10101010), ezt követi egy bitenkénti ÉS és egy negált ÉS művelet.

#20

cin = 1; //8'b00001101

#20

opcode = {ACCU,RSHIFT}; //8'b10000110

cin = 0; //~1320 ns!

#20

opcode = {REG1,CPY}; //ACCU =8'b10101010

#20

opcode = {REG2,XOR}; //8'b10100101

#20

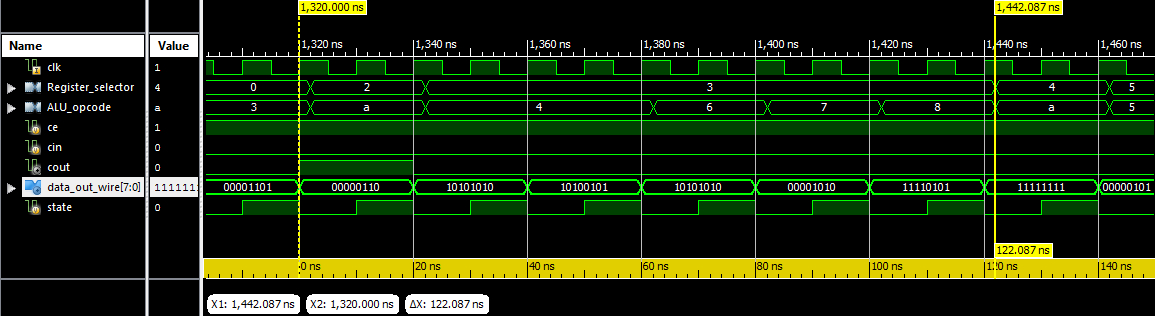
opcode = {REG2,XOR}; //8'b10101010

#20

opcode = {REG2,AND}; //8'b00001010

#20

opcode = {REG2,NAND}; //8'b11110101



Ezt követi egy VAGY művelet, még egy másolás majd egy komparálás, amely a 8'h00000010 értékkel jelzi, hogy az AKKU-ban tárolt 8'b00000101 nagyobb mint a 4-es regiszter 8'b00000100 értéke.

#20

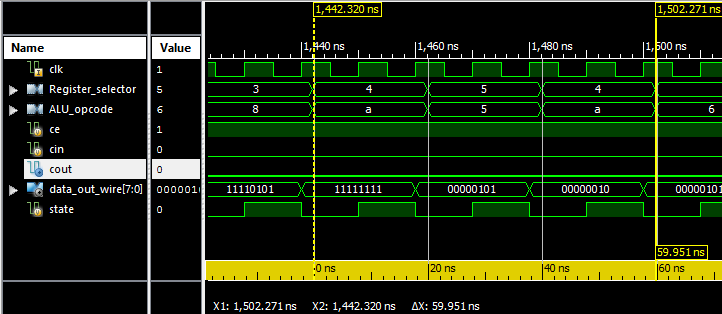
opcode = {REG2,OR}; //8'b11111111

#20

opcode = {REG3,CPY}; //8'b00000101

#20

opcode = {REG4,CMP}; //8'h02



Ezt még egy másolás egy ÉS művelet és ismételten egy komparálás történik. Mivel most megegyezik a két operandus, a kimeneten 8'b00000001 jelenik meg.

#20

opcode = {REG3,CPY}; //8'b00000101

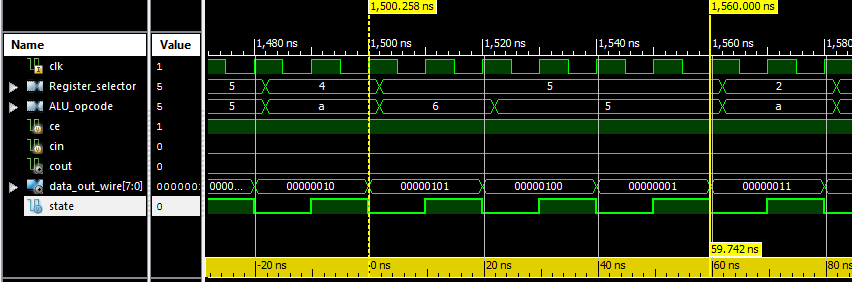
#20

opcode = {REG4,AND}; //8'b00000100

#20

opcode = {REG4,CMP}; //8'h01

#20



Ezt követi még egy komparálás ami most azt jelzi, hogy a REG4 értéke nagyobb mint az aktuális AKKU. Ezt még egy másolás illetve egy negált VAGY művelet követ:

#20

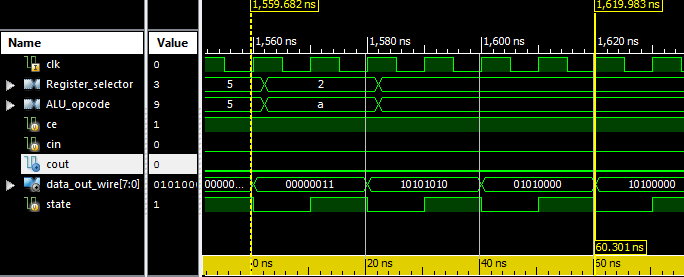
opcode = {REG4,CMP}; //8'h03

#20

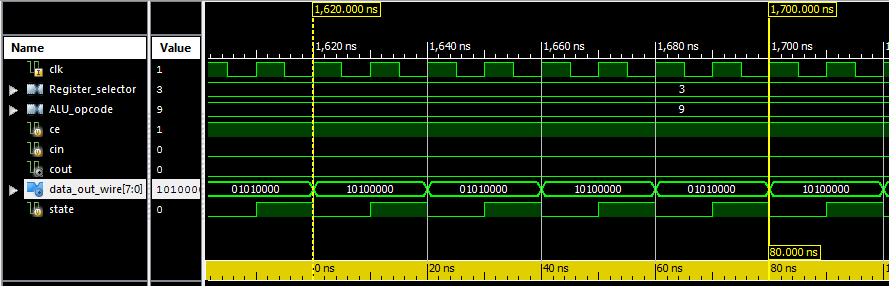
opcode = {REG1,CPY}; //ACCU =8'b10101010

#20

opcode = {REG2,NOR}; //8'b01010000



Ezt már más művelet nem követ a testbenchben, azonban itt figyelhető meg a CE jel használatának szüksége, ugyanis ha hagyjuk tovább futni a szimulációt az adott beállításokkal, akkor a kimenet változik. Tehát, amennyiben nem húzzuk 0-ra a CE-t egy valós működésben a bármi is van a bemeneteken az eszköz műveleteket fog végezni, amellyel energiát fogyaszt, illetve az AKKU értéke kiszámíthatatlanná válik:



# Függelék

ALU\_unit.v :

`timescale 1ns / 1ps

module ALU\_unit(

input [7:0] opA,

input [7:0] opB,

output [7:0] outData,

input [3:0] opcode,

input cin,

output cout

);

reg [7:0] outDataReg;

reg cout\_reg;

assign cout = cout\_reg;

assign outData = outDataReg ;

parameter ADD = 4'h0;

parameter SUB = 4'h1;

parameter LSHIFT = 4'h2;

parameter RSHIFT = 4'h3;

parameter XOR = 4'h4;

parameter CMP = 4'h5;

parameter AND = 4'h6;

parameter NAND = 4'h7;

parameter OR = 4'h8;

parameter NOR = 4'h9;

parameter CPY = 4'ha; //Copies value to the output (Copy to the ACCU)

always @ ( opA, opB, cin, opcode)

if (opcode == ADD)

{cout\_reg,outDataReg} <= opA + opB + cin;

else if (opcode == SUB)

{cout\_reg, outDataReg} <= opA - opB + cin;

else if (opcode == LSHIFT)

{cout\_reg, outDataReg} <= {opA[7:0],cin} ;

else if (opcode == RSHIFT)

{outDataReg, cout\_reg} <= { cin, opB[7:0] };

else

begin

cout\_reg <= 0;

case (opcode)

XOR: outDataReg <= opA^opB;

AND: outDataReg <= opA&opB;

NAND: outDataReg <= ~(opA&opB);

OR: outDataReg <= opA|opB;

NOR: outDataReg <= ~(opA|opB);

CPY: outDataReg <= opB;

CMP:

begin

if(opA == opB) outDataReg <= 8'h01;

else if (opA>opB)outDataReg <= 8'h02;

else outDataReg <= 8'h03;

end

default: outDataReg <= 8'h0;

endcase

end

endmodule

CPU.v:

`timescale 1ns / 1ps

module CPU(

input clk,

input rst,

input [7:0] data\_in,

input [6:0] opcode, // Valójában 8 bites az operandus, a 8. bit a 'load'

input cin,

output cout,

input load,

input ce,

output [7:0] data\_out

);

//Regiszterek az értékek tárolására

reg [7:0] operands [0:7];

//regiszterezendő bemeneti értékek

reg cin\_reg, cout\_reg;

reg [3:0] operation;

reg [2:0] reg\_selector;

wire [7:0] operandA, operandB;

wire [7:0] alu\_output\_wire;

wire [3:0] op\_wire;

wire cin\_wire, cout\_wire;

//Aktuális állapotot tároló regiszter

parameter IDLE = 1'b0;

parameter OPERATION = 1'b1;

reg state;

integer i;

assign op\_wire = operation;

assign operandA = operands[0]; // Az egyik operandus mindig az Akku

assign operandB = operands[reg\_selector]; // Ez választja ki, hogy melyik regiszter a másik operandus

assign data\_out = operands[0];

assign cout = cout\_reg;

assign cin\_wire = cin\_reg;

always @ (posedge clk, posedge rst)

if (rst)

begin

for (i = 0 ; i<4 ; i= i+1)

begin

operands[i] <= 0;

end

reg\_selector <= 0;

state <= IDLE;

cin\_reg <= 0;

cout\_reg <= 0;

operation <= 0;

end

else

begin

if (state == IDLE)

begin

if (ce==1 && load ==1 ) // Ebben a helyzetben a regiszterek töltődnek fel adatokkal

operands[opcode[6:4]] <= data\_in;

else if (ce == 1 && load == 0 )

begin

reg\_selector <= opcode[6:4];

cin\_reg <= cin;

cout\_reg <=cout;

operation <= opcode[3:0];

state <= OPERATION;

end

end

else

begin

operands[0] <= alu\_output\_wire;

cout\_reg <= cout\_wire;

state <=IDLE;

end

end

//tisztán kombinációs hálózat ALU példányosítása

ALU\_unit ALU(

.opA (operandA),

.opB (operandB),

.outData (alu\_output\_wire),

.opcode (op\_wire),

.cin (cin\_wire),

.cout (cout\_wire)

);

endmodule

