

**Budapesti Műszaki és Gazdaságtudományi Egyetem**

**Villamosmérnöki és Informatikai Kar**

**Beágyazott Információs Rendszerek főspecializáció (MIT)**

**Logikai Tervezés**

**1A : ALU egység 8 regiszterrel és 8 utasítással**

Készítették:

Nagy Tímea Csilla (O5D5VN)

Cseh Péter (DM5HMB)

Konzulens:

Fehér Béla

2017.05.18.

# ALU bemutatása

Az aritmetikai és logikai egység, röviden ALU, egy kombinációs logikát tartalmazó digitális áramkör.

Az ALU része a központi feldolgozó egységnek (CPU). Feladata műveleteket végezni a bemeneti egész típusú értékeken (operandusok), és előállítani a megfelelő kimenetet a művelet típusának függvényében.

A műveletek lehetnek aritmetikai műveletek, például: összeadás, összeadás carry-vel, kivonás, inkrementálás, szorzás. A műveletek másik csoportja a logikai műveletek, néhány gyakran megvalósított logikai művelet: AND, OR, XOR kapcsolat, egyes komplemens képzés. További művelet típus a léptető (shift)/forgató műveletek, ezek lehetnek például: logikai shiftelés, aritmetikai shiftelés, forgatás, forgatás carry-n keresztül.

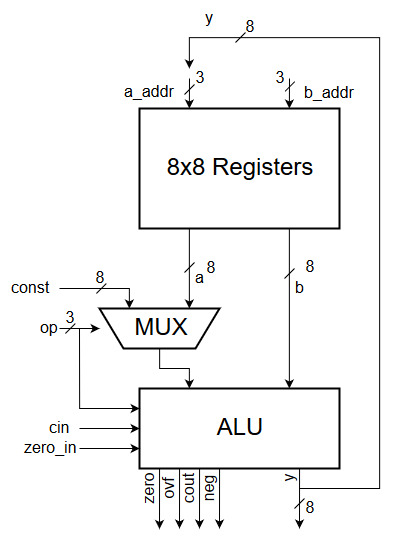
Másik feladata az ALU-nak a kimeneti feltétel biteket (flag-ek) állítása. A flag-ek kiegészítő információt hordoznak a kapott eredményről, vagy a műveletvégezésről. Segítségükkel történnek a feltételes ugrások a programban. A művelettől függ, hogy az ALU állítja-e az adott flag-et, vagy sem. Gyakran megvalósított flag-ek:

* Carry: jelzi, ha van átvitel a műveletvégzés során
* Half-carry: jelzi, ha az adatszélesség felénél történik átvitel (BCD számok aritmetikájánál hasznos)
* Overflow: Túlcsordulás, jelzi, ha a számítás eredménye nem fér el a számtartományon, ekkor az eredmény hibás
* Negative: jelzi, ha a műveletvégzés eredménye negatív
* Zero: jelzi, ha az eredmény nulla

# Specifikáció és Tervezés

A feladat specifikációja szerint az ALU-nak minimum az alábbi képességekkel kell rendelkeznie: összeadás, kivonás, shiftelés, komparálás, EXOR, AND, NAND, OR, NOR. Az operandusok 8 bitesek. Kaszkádosítható legyen az ALU. 8 utasítással és 8 regiszterrel kell rendelkeznie.

## Felépítés



. ábra ALU magas szintű blokkvázlata

Az ALU magas szintű blokkvázlata az 1. ábrán látható. Az architektúránál a kétoperandusú megoldást választottuk, tehát a visszaírt értéket valamelyik bemeneti operandus címére írja vissza az ALU.

Bemenetei a két operanduscím(*a\_addr* és *b\_addr*), amivel kiválasztható, hogy a 8 mély és 8 bit széles regiszter tömb melyik regiszterei lesznek az ALU operandusai. Ennek megfelelően ezek a vonalak 3bit szélesek, mivel így a teljes memória címezhető. Az így kiválasztott regiszterek értékét *a* és *b* jelöli.

A *op* bemeneti jellel kiválasztható, hogy az ALU egység milyen műveletet végezzen, illetve vezérli a multiplexert, amely kiválasztja, hogy az ALU egység egyes operandusa a bejövő konstans (*const*) vagy az *a* jel legyen. Az *op* jel 3bites, hogy mind a nyolc művelet kiválasztható legyen.

Az további két bemenő jel a bejövő átvitel (*cin*, azaz Carry In) és a bejövő zero flag (*zero\_in*) a kaszkádosítás miatt szükséges.

Az ALU-nak kimenő jele az *y*-al jelzett eredmény, amelyet az *a\_addr* regiszterbe ír bele az ALU. Illetve kimenő jelek a flag-ek, melyek a következők: *zero*, *ovf* (Overflow), *cout* (Carry Out), *neg* (Negative).

## Műveletek

A tervezés során az alábbi műveleteket választottuk az ALU-nak.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Op | Művelet | Leírás | Zero | Cout | Ovf | Neg |
| 000 | MOV | y <= const | - | - | - | - |
| 001 | AND | y <= a & b | + | - | - | + |
| 010 | OR | y <= a | b | + | - | - | + |
| 011 | XOR | y <= a ^ b | + | - | - | + |
| 100 | RRC | y <= {cin, a[7:1]} | + | + | - | + |
| 101 | CMP | y <= a | + | + | + | + |
| 110 | ADD | y <= a + b + cin | + | + | + | + |
| 111 | SUB | y <= a - b - cin | + | + | + | + |

1. táblázat ALU műveletei

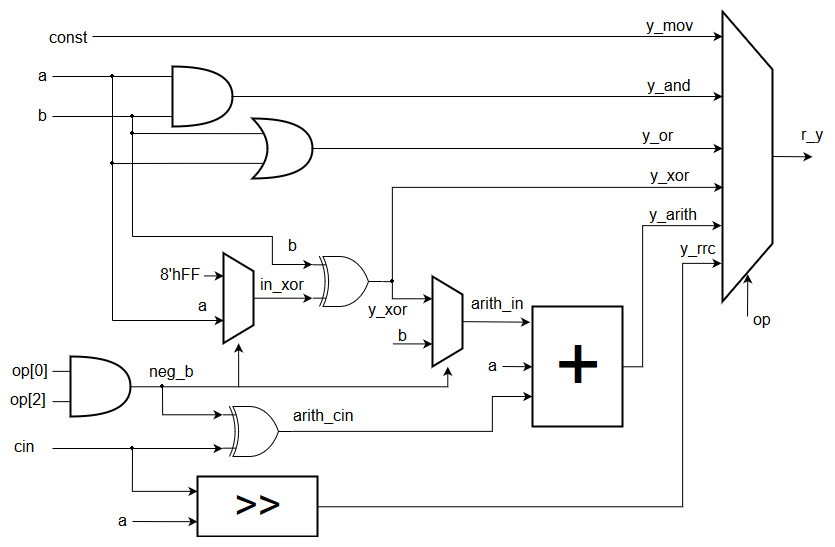
Az első oszlopban van a művelet kódja, a második oszlopban a rövid megnevezése, a harmadik oszlopban a leírása, hogy, hogy kapjuk meg a kimeneti eredményt. Az utolsó négy oszlopban látható, hogy a művelet hogyan változtatja a kimeneti flag-eket. A „+” jel arra utal, hogy a flag valamilyen számítás eredménye, ekkor az értéke lehet 1 vagy 0, „-” esetén mindenképp 0 a flag.

A műveletek kiválasztása során a legfontosabb szempont az volt, hogy maximum 8 legyen belőlünk, illetve a specifikációban leírt minimális képességek megvalósíthatók legyenek az ALU-val.

Speciális eset a MOV utasítás, erre mindenképp szükség volt, mivel ennek a segítségével lehet valamilyen értéket beírni a megcímzett regiszterbe. A jobbra shiftelés az RRC-vel oldható meg, amely jobbra lépteti az *a* operandust, és az MSB helyére a *cin*-t shifteli be. A balra shiftelést el lehet végezni az összeadás segítségével, ha ugyanazt a regisztert választjuk ki mindekét operandusnak, ekkor ugyanis a regiszter értékét összeadja önmagával, tehát a regiszter értékét kettővel szorozza, tehát lépteti balra. A NOR és NAND műveletek több iteráción keresztül valósíthatók meg egy OR vagy AND művelettel és egy XOR művelettel, ahol az egyik operandus csupa egy (mivel a XOR egy vezérelhető inverter).

# Implementáció

## Blokkvázlat



. ábra ALU belső felépítse

A 2. ábrán látható az ALU belső felépítése, a művelet eredménye szempontjából (a flag-ek számolása nincs az ábrán). Az egyes műveleti eredmények külön áramkörön haladnak keresztül, és a multiplexer választja ki ezek közül a kimenetet.

Az ALU belső felépítése során törekedtünk arra, hogy minimális erőforrást használjunk. Ezt úgy értük el, hogy az egyes áramköröket újra felhasználtuk ott ahol lehetett. Az ADD, CMP, és SUB utasítások ugyanazt az összeadót használják, csak az összeadó bemenetén más adat jelenik meg. Ahhoz, hogy a kivonást visszavezethessük összeadásra elő kell állítani a *b* operandus kettes komplemensét. Ehhez a már meglévő XOR kaput használja az ALU.

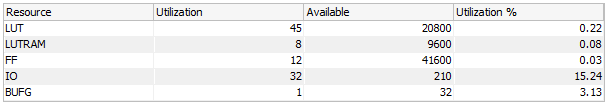
A kimeneti flag-ek meghatározásakor ahol kellett multiplexerekkel választjuk ki a megfelelő kimenetet, hiszen általános szabályt sok esetben nem tudunk meghatározni, mert például a MOV utasítás esetén minden flag 0. A flag-ek értékét a következő képpen határoztuk meg:

* Zero: Az ALU saját zero flag-e a bitredukciós NOR kapcsolata az eredménynek. Tehát ha akár egyetlen bit értéke 1 a kimeneten akkor a Zero flag 0, egyébként 1. Ezt az értéket a kaszkádosítás miatt ÉS kapcsolatba hozzuk a bemeneti *zero\_in* jellel. Tehát ha nem szeretnénk kaszkádosítani, akkor a *zero\_in*-t logikai 1 szintre kell kötni, hogy helyesen működjön a Zero flag
* Cout: a művelet végzés során a kimeneti adatot 9 bitesre bővítettük, ennek a 8. bitje van kivezetve kimenő átvitelként
* Ovf: túlcsordulás akkor történik ha a bemeneti értékek ellenkező előjelűek, mint a kimeneti érték
* Neg: A negatív flag a kimeneti eredmény MSB-je, azaz az előjel bit

## Időzítés analízis

(\*\*\*Ezt kérlek csináld meg Timi\*\*\*)

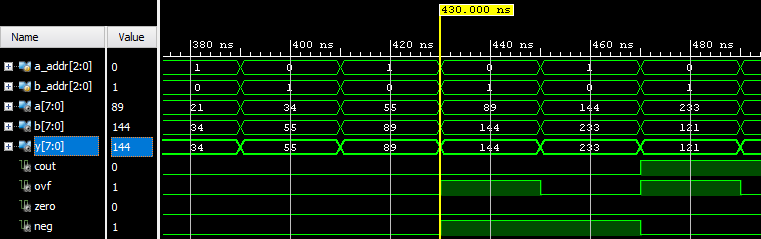
## Erőforrások

A leírt hardver erőforrás szintéziséhez Artix-7 FPGA-t (pontos típus: xc7a35tcsg324-2) használtunk. Az implementált hardver által felhasznált erőforrásokat a következő ábrán láthatjuk.  


. ábra Architektúra erőforrásai

# Verifikáció

## Fibonacci sorozat

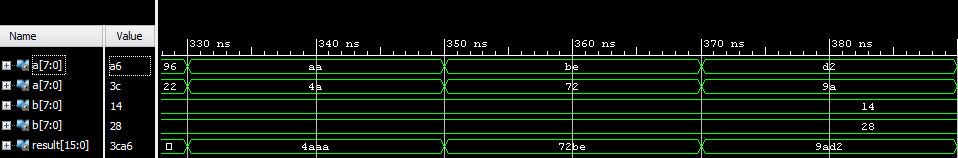


. ábra Szimuláció: Fibonacci sorzat

A szimulációs ábrán látható, hogy a regisztercímek, folyamatosan megcserélődnek. A kimeneten megjelenik az *a* és *b* összege (az ábrázolásmód előjel nélküli decimális).

## Kaszkádosítás

A kaszkádoítás során a test bench-ben két alut példányosítunk, az egyik az alsó 8 biten számol, a másik a felső 8 biten. Az alsó egység *zero\_in* jelét logikai egyre kötjük, a *cout* jelét a felső egység *cin*-jével kötjük össze. A kimeneti zero flag-et összekötjük a felső egység *zero\_in*-jével. A szimulációt egy összeadásra végeztük el. Az ábrán a jelek sorrendben: a low, a high, b low, b high, eredmény.



# Kódok

## ALU architektúra

|  |  |
| --- | --- |
| `timescale 1ns / 1ps  //`default\_nettype none  //////////////////////////////////////////////////////////////////////////////////  // Company: BME  // Engineer: Nagy Tímea Csilla(O5D5VN), Cseh Péter(DM5HMB)  //  // Create Date: 17.04.2017 12:39:55  // Design Name: alu  // Module Name: alu  //////////////////////////////////////////////////////////////////////////////////  module alu  (  input clk,    input [2:0] a\_addr,  input [2:0] b\_addr,  input [7:0] const,  input [2:0] op,  input cin,  input zero\_in,    output reg [7:0] y,  output reg cout,  output reg ovf,  output reg zero,  output reg neg  );    //Operation Codes  parameter MOV = 3'b000;  parameter AND = 3'b001;  parameter OR = 3'b010;  parameter XOR = 3'b011;  parameter RRC = 3'b100;  parameter CMP = 3'b101;  parameter ADD = 3'b110;  parameter SUB = 3'b111;    //8x8 Register Array(Block RAM)  reg [7:0] regs[7:0];  //A and B operands  wire [7:0] a = regs[a\_addr];  wire [7:0] b = regs[b\_addr];    //MOV  wire [7:0] y\_mov = const;  //AND  wire [7:0] y\_and = a & b;  //OR  wire [7:0] y\_or = a | b;  //XOR  //At opcode = 1X1(CMP and SUB) -> negates the b operand  wire neg\_b = op[0] & op[2];  wire [7:0] in\_xor = neg\_b == 1'b1 ? {8{1'b1}} : a;  wire [7:0] y\_xor = in\_xor ^ b;  //RRC : Rotates "a" to right over carry in  wire [7:0] y\_rrc = {a[0], cin, a[7:1]};  //ARITH  //ADD: a + b + cin  //CMP, SUB: a - b - cin = a + (~b + 1'b1) - cin = a + ~b + ~cin | wire [7:0] arith\_in = (neg\_b == 1'b1) ? y\_xor : b;  wire arith\_cin = neg\_b ^ cin; //negated carry in for CMP and SUB  wire [8:0] y\_arith = a + arith\_in + arith\_cin;  wire ovf\_arith = (a[7] & b[7] & ~y\_arith[7]) | (~a[7] & ~b[7] & y\_arith[7]);  reg [8:0] r\_y;  reg r\_cout;  reg r\_ovf;  reg r\_zero;  reg r\_neg;  //MUXES  always @ (\*)  begin  //Y  case (op)  MOV : r\_y <= {1'b0, y\_mov};  AND : r\_y <= {1'b0, y\_and};  OR : r\_y <= {1'b0, y\_or};  XOR : r\_y <= {1'b0, y\_xor};  RRC : r\_y <= y\_rrc;  CMP : r\_y <= {y\_arith[8], a}; //result not changed  ADD : r\_y <= y\_arith;  SUB : r\_y <= {~y\_arith[8], y\_arith[7:0]}; //~carry  endcase    //COUT  r\_cout <= r\_y[8];    //OVF  case (op)  CMP : r\_ovf <= ovf\_arith;  ADD : r\_ovf <= ovf\_arith;  SUB : r\_ovf <= ovf\_arith;  default: r\_ovf <= 1'b0;  endcase    //ZERO  case (op)  MOV : r\_zero <= 1'b0;  default: r\_zero <= ~|r\_y[7:0];  endcase    //NEG  case (op)  MOV : r\_neg <= 1'b0;  default: r\_neg <= r\_y[7];  endcase    end  //Write the result back to a\_addr  always @ (posedge clk)  regs[a\_addr] <= r\_y[7:0];    //Output regs  always @ (posedge clk)  begin  y <= r\_y;  cout <= r\_cout;  ovf <= r\_ovf;  zero <= (r\_zero & zero\_in);  neg <= r\_neg;  end  endmodule |

## Fibonacci Test

|  |  |
| --- | --- |
| `timescale 1ns / 1ps  //`default\_nettype none  //////////////////////////////////////////////////////////////////////////////////  // Company: BME  // Engineer: Nagy Tímea Csilla(O5D5VN), Cseh Péter(DM5HMB)  //  // Create Date: 17.04.2017 12:39:55  // Design Name: alu  // Module Name: alu\_sim  //////////////////////////////////////////////////////////////////////////////////  module alu\_sim;  // Inputs  reg clk;  reg [2:0] a\_addr;  reg [2:0] b\_addr;  reg [7:0] const;  reg [2:0] op;  reg cin;  reg zero\_in;    //Outputs  wire [7:0] y;  wire cout;  wire ovf;  wire zero;  wire neg;  // Instantiate the Unit Under Test (UUT)  alu uut  (  .clk(clk),  .a\_addr(a\_addr),  .b\_addr(b\_addr),  .const(const),  .op(op),  .cin(cin),  .zero\_in(zero\_in),    .y(y),  .cout(cout),  .ovf(ovf),  .zero(zero),  .neg(neg)  ); | initial begin  clk = 0;  a\_addr <= 0;  b\_addr <= 1;  const = 0;  op = MOV;  cin = 1'b0;  zero\_in = 1'b1;  end  //Operation Codes  parameter MOV = 3'b000;  parameter AND = 3'b001;  parameter OR = 3'b010;  parameter XOR = 3'b011;  parameter RRC = 3'b100;  parameter CMP = 3'b101;  parameter ADD = 3'b110;  parameter SUB = 3'b111;  always #10 clk = ~clk;  //Init: clear all reg, write 1 to addr 0; a\_addr = 0, b\_addr = 1  initial #20 a\_addr = 0;  initial #40 a\_addr = 1;  initial #60 a\_addr = 2;  initial #80 a\_addr = 3;  initial #100 a\_addr = 4;  initial #120 a\_addr = 5;  initial #140 a\_addr = 6;  initial #160 a\_addr = 7;  initial #180 a\_addr = 0;  initial #180 const = 1;  reg init;  initial #10 init = 0;  initial #200 init = 1;  //Fibonacci  always @ (posedge clk)  begin  if(init == 1)  begin  op <= ADD;  a\_addr <= b\_addr;  b\_addr <= a\_addr;  end  end  endmodule |

## Kaszkádosítás teszt

|  |  |
| --- | --- |
| `timescale 1ns / 1ps  //`default\_nettype none  //////////////////////////////////////////////////////////////////////////////////  // Company: BME  // Engineer: Nagy Tímea Csilla(O5D5VN), Cseh Péter(DM5HMB)  //  // Create Date: 17.04.2017 12:39:55  // Design Name: alu  // Module Name: alu\_sim  //////////////////////////////////////////////////////////////////////////////////  module alu\_cascade\_sim;  // Inputs  reg clk;  reg [2:0] a\_addr[1:0];  reg [2:0] b\_addr[1:0];  reg [7:0] const[1:0];  reg [2:0] op;  reg cin;  reg zero\_in;    //Outputs  wire [7:0] y[1:0];  wire cout[1:0];  wire ovf[1:0];  wire zero[1:0];  wire neg[1:0];  // Instantiate the Unit Under Test (UUT)  //ALU1  alu uut1  (  .clk(clk),  .a\_addr(a\_addr[0]),  .b\_addr(b\_addr[0]),  .const(const[0]),  .op(op),  .cin(cin),  .zero\_in(zero\_in),    .y(y[0]),  .cout(cout[0]),  .ovf(ovf[0]),  .zero(zero[0]),  .neg(neg[0])  );    //ALU2  alu uut2  (  .clk(clk),  .a\_addr(a\_addr[1]),  .b\_addr(b\_addr[1]),  .const(const[1]),  .op(op),  .cin(cout[0]),  .zero\_in(zero[0]),    .y(y[1]),  .cout(cout[1]),  .ovf(ovf[1]),  .zero(zero[1]),  .neg(neg[1])  ); | wire [15:0] result = {y[1], y[0]};    //Operation Codes  parameter MOV = 3'b000;  parameter AND = 3'b001;  parameter OR = 3'b010;  parameter XOR = 3'b011;  parameter RRC = 3'b100;  parameter CMP = 3'b101;  parameter ADD = 3'b110;  parameter SUB = 3'b111;  reg[1:0] i;  initial begin  clk = 0;  cin <= 0;  zero\_in <= 0;  op <= MOV;  for(i = 0; i < 2; i = i + 1)  begin  a\_addr[i] <= 0;  b\_addr[i] <= 1;  const[i] <= 0;  end  end  always #10 clk = ~clk;  initial #20 a\_addr[0] = 1;  initial #40 const[0] = 20;  initial #60 a\_addr[0] = 0;  initial #80 const[0] = 30;  initial #120 a\_addr[1] = 1;  initial #140 const[1] = 40;  initial #160 a\_addr[1] = 0;  initial #180 const[1] = 50;  initial #200 op <= ADD;  initial #400 op <= SUB;  endmodule |