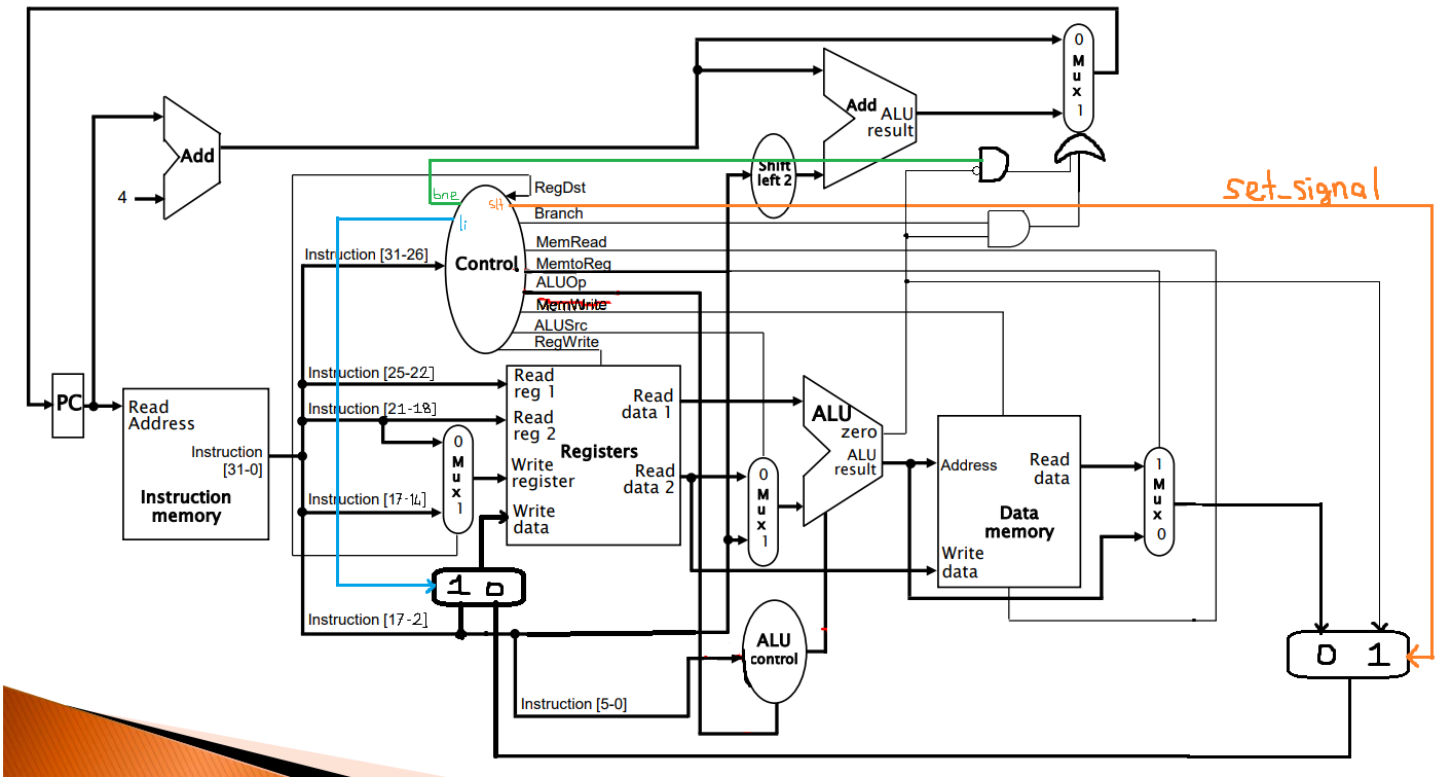
ERSEL CELAL EREN

1901042673

CSE341 – PROJE RAPORU



- Derste gösterilen Mips şemasını ödevde istenen instructionlara uygun olacak şekilde düzenlemeler yaptım.

• mips16_single_cycle modülü ilk olarak Instruction Memory'den 32-bit Instruction alır.Instruction Memeory'den ProgramCounter'ın gösterdiği adres ile Instruction alınır.

• Daha sonra Control Unit' Instruction'ın Opcode'u yani [31:26] bitleri gönderilir ve burada gerekli sinyaller oluşturulur.

• Control Unitte oluşan sinyallerden biri olan regDest sinyaline göre 5 bit multiplexer ile destination register seçilir.(rd veya rt)

• ALU Control'e Instruction'ın Opcode ve Function Code bitleri gönderilir ve ALU için ALUControlOP[2:0] bitleri oluşturulur.

• Instruction'ın [25:22] ve [21:18] bitleri Register'a gönderilir ve bu registerlardan 16-bitlik 2 değer okunur. Biri direkt ALU'ya girerken diğeri Instruction'ın [17:2] bitleri yani immediate field'ı ile multiplexer tarafından ALUSrc ile seçilir ve ALU'ya gönderilir.

• Instruction'ın [17:2] bitleri ve ALU'dan çıkan sonuç 'li' sinyali kontrolünde mux ile Register'a yazılır. 'li' sinyali Control Unit'ten gelir.

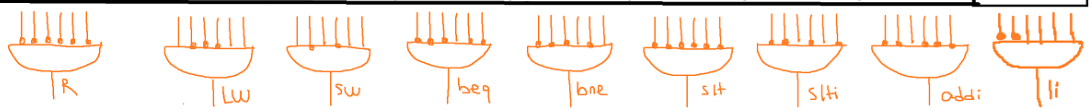
• ALUControlOP bitleri ile ALU'ya giren iki 16-bitlik data işleme sokulur. Result, Data Memory'nin address girişine ve Data Memory'den okunacak değerle arasında seçim yapılacak olan mux'a gider. Bu mux MemToReg sinyali ile kontrol edilir. Data Memory, MemWrite ve MemRead ile kontrol edilir. Bu mux'un çıkışı ControlUnit'ten gelen "set on less than"

instruction'ı için gelen set_signal ile kontrol edilen başka bir mux'a bağlanır. Bu mux'a ALU ve Data Memory seçiminden gelen result ve ALU'dan gelen "zero" biti bağlanır. Instruction "slt" ise zero bit'inin sonucu Registerlara iletilir ve istenilen adrese 1 ya da 0 sonucu yazılır.

• Control Unit'ten gelen "bne(branch not equal)" sinyali ALU'dan çıkan "zero" bitinin tersi ile and'lenir ve branch işleminin sonucuyla or'lanarak "branch not equal" instruction'ı gerçekleştirilmiş olur.

• Instruction'ın [17:2] bitleri register ve data memory 16-bitlik olduğu için sign extend olmadan ALU'ya girer.

opcodes →	000000	100011	101011	000100	000101	000000	001010	001000	001111
	R-type	lw	sw	beq	bne	slt	slti	addi	li
RegDst	1	0	X	X	X	1	1	0	1
AluSrc	0	1	1	0	0	0	1	1	1
MemtoReg	0	1	X	X	X	X	X	0	0
RegWrite	1	1	0	0	0	1	1	1	1
MemRead	0	1	0	0	0	0	0	0	0
MemWrite	0	0	1	0	0	0	0	0	0
Branch	0	0	0	1	0	0	0	0	0
AluOp1	1	0	0	0	0	0	0	0	0
AluOp0	0	0	0	1	1	1	1	0	0
AluOP	R-type	Add	Add	Subtract	Subtract	Subtract	Subtract	Add	
Set_signal	0	0	0	0	0	1	1	0	0
Bne_signal	0	0	0	0	1	0	0	0	0



Derste gösterilen Control Unit sinyallerinin tablosunun projede istenen instructionlar için genişletilmiş halini çizdim.

RegDest = R + li

AluSrc = lw + sw

MemToReg = lw

RegWrite = R + lw + slt + slti + addi + li

MemRead = lw

MemWrite = sw

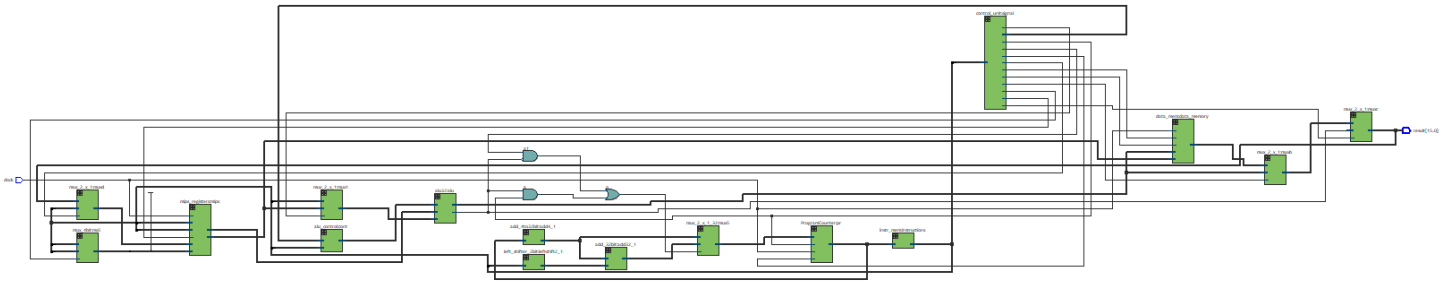
Branch = beq

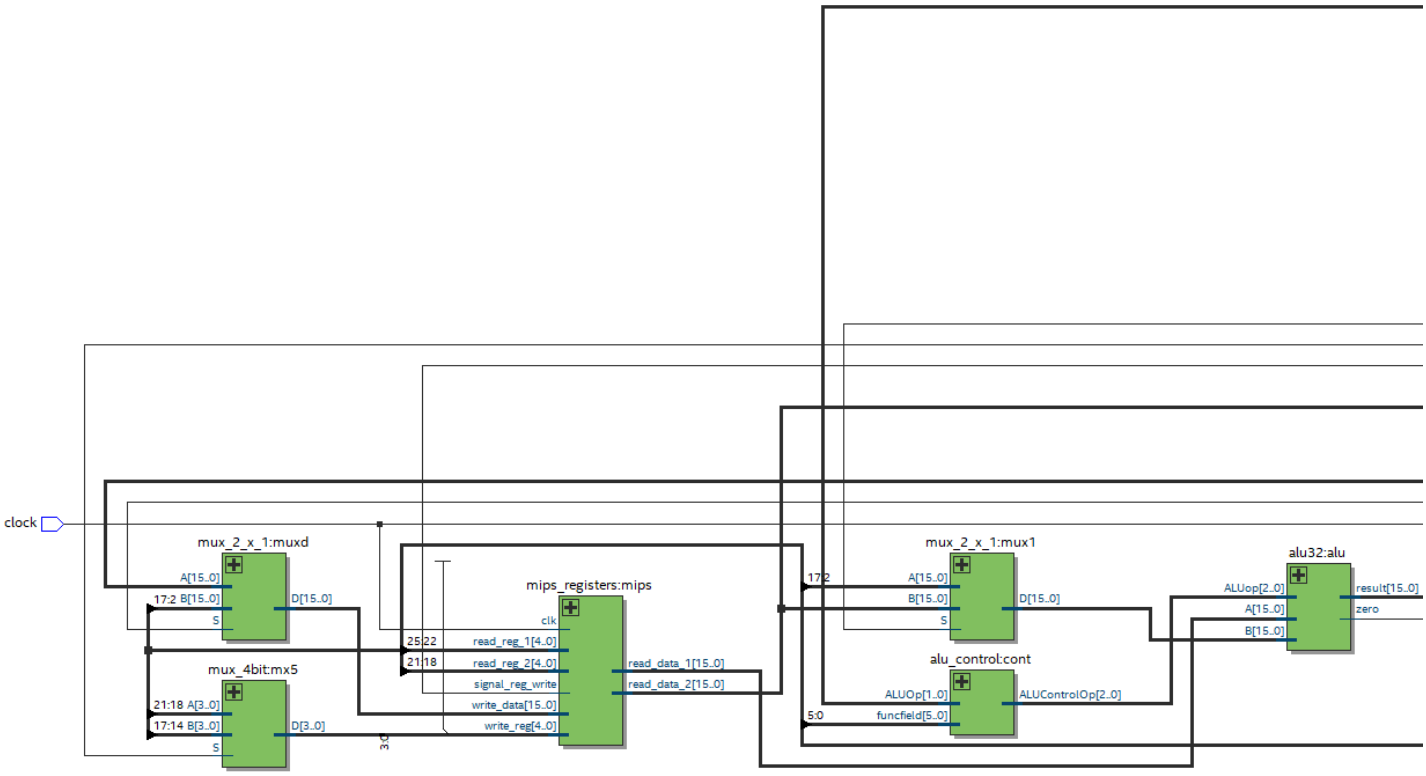
AluOp1 = R

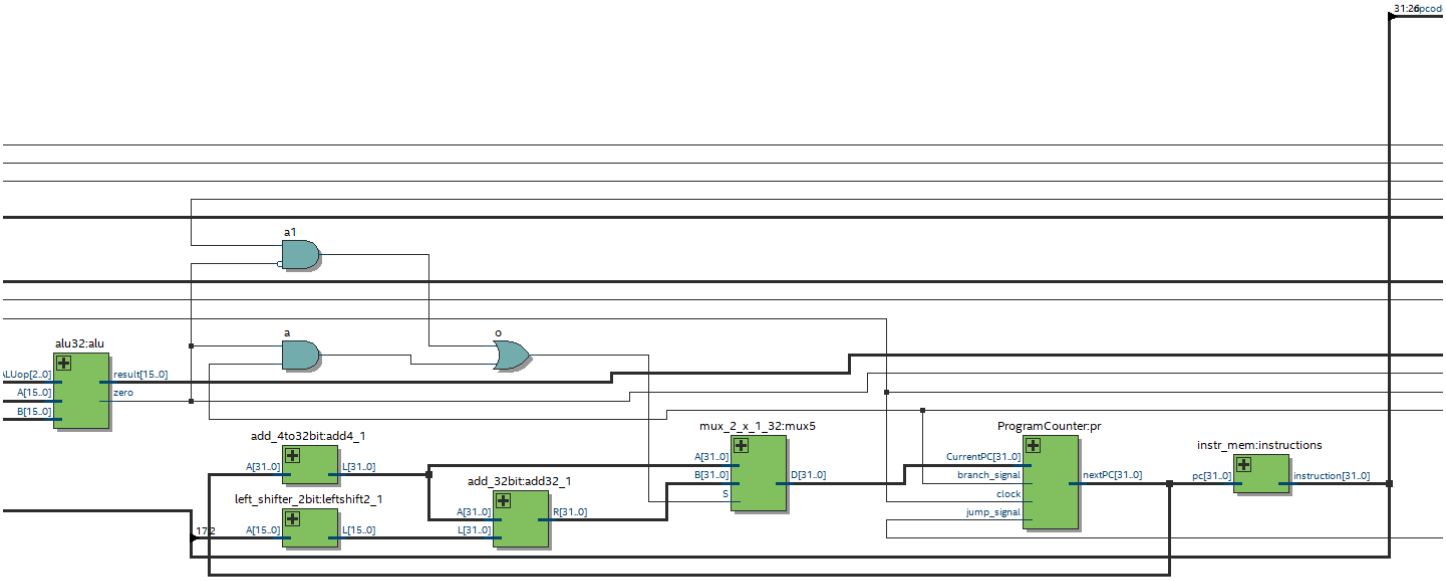
AluOp0 = beq + bne + slt + slti

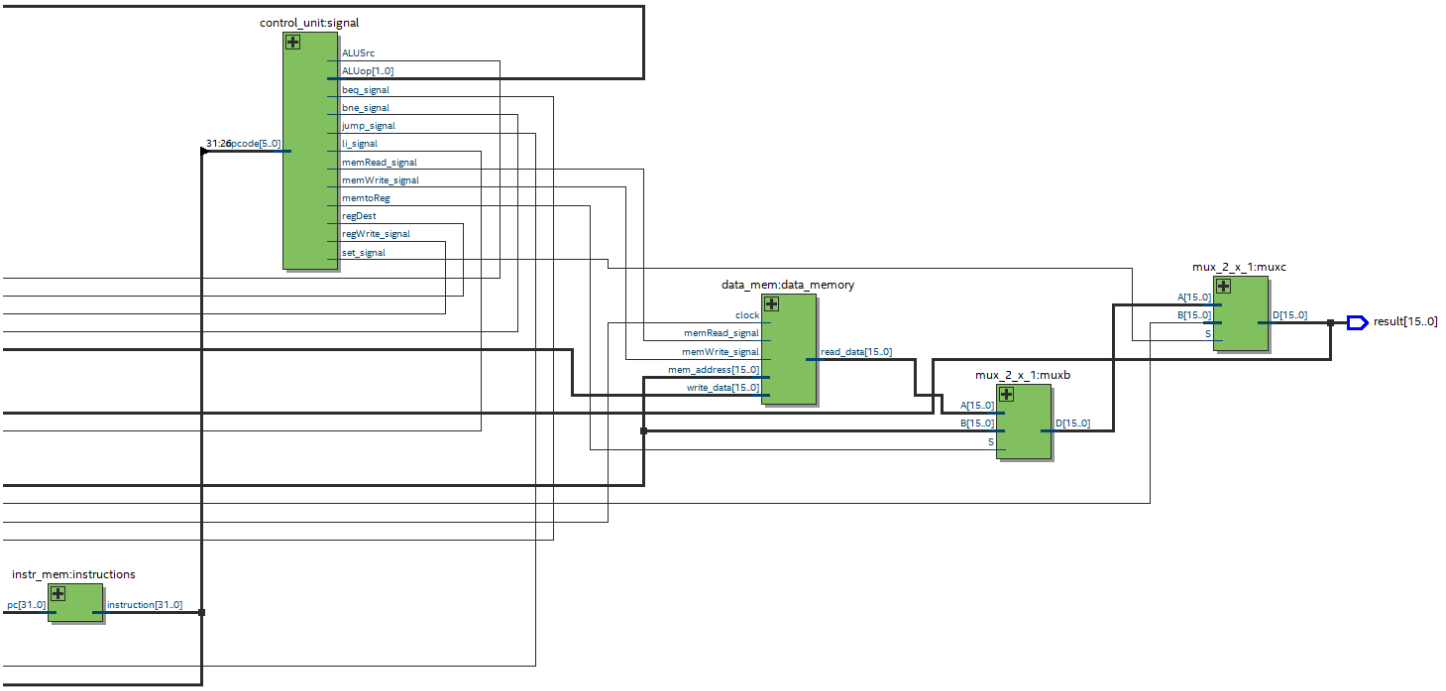
Set_signal = slt + slti

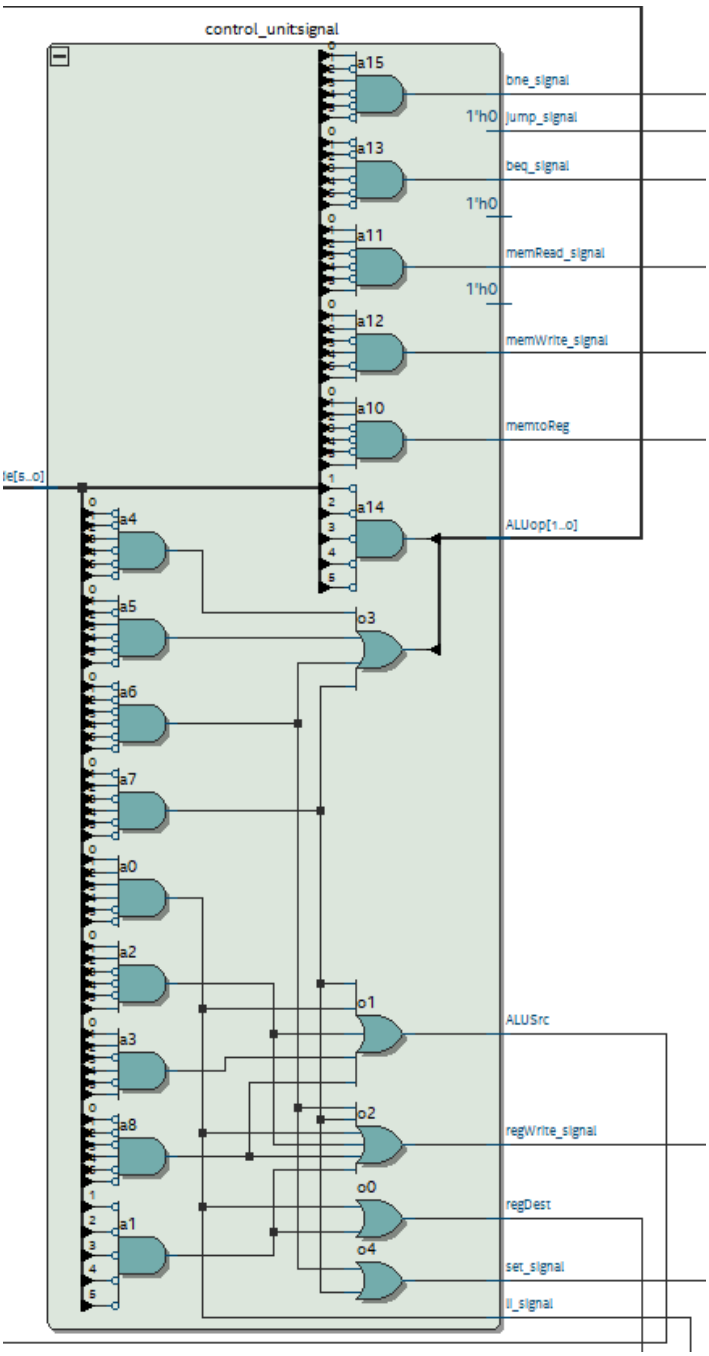
Bne_signal = bne

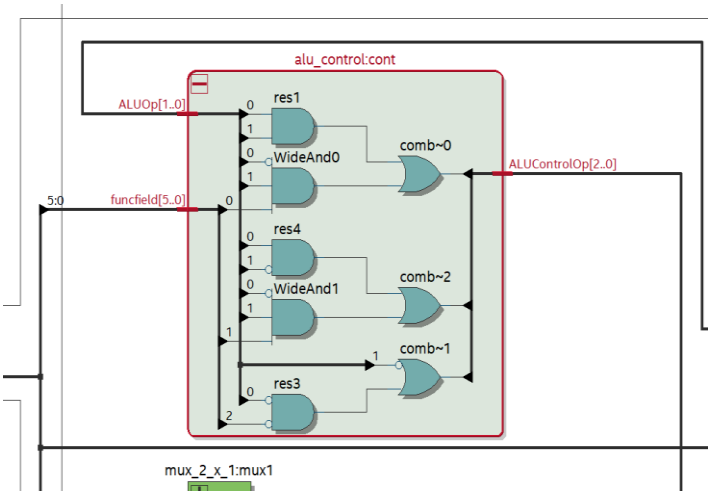



clock
mux_2_x_1:muxd
A[15..0]
17:2 B[15..0]
S
D[15..0]
mux_4bit:mx5
21:18 A[3..0]
17:14 B[3..0]
S
D[3..0]
3:0
mips_registers:mips
clk
25:22
21:18
read_reg_1[4..0]
read_reg_2[4..0]
signal_reg_write
write_data[15..0]
write_reg[4..0]
read_data_1[15..0]
read_data_2[15..0]
17:2
mux_2_x_1:mux1
A[15..0]
B[15..0]
S
D[15..0]
alu_control:cont
ALUOp[1..0]
funcfield[5..0]
ALUControlOp[2..0]
5:0
alu32:alu
ALUop[2..0]
A[15..0]
B[15..0]
result[15..0]
zero


31:26pcod
a1
a
o
alu32:alu
ALUop[2..0]
A[15..0]
B[15..0]
result[15..0]
zero
add_4to32bit:add4_1
A[31..0]
L[31..0]
left_shifter_2bit:leftshift2_1
A[15..0]
L[15..0]
172
add_32bit:add32_1
A[31..0]
L[31..0]
R[31..0]
mux_2_x_1_32:mux5
A[31..0]
B[31..0]
S
D[31..0]
ProgramCounter:pr
CurrentPC[31..0]
branch_signal
clock
jump_signal
nextPC[31..0]
instr_mem:instructions
pc[31..0]
instruction[31..0]


control_unit:signal
31:26pcode[5..0]
ALUSrc
ALUop[1..0]
beq_signal
bne_signal
jump_signal
li_signal
memRead_signal
memWrite_signal
memtoReg
regDest
regWrite_signal
set_signal
data_mem:data_memory
clock
memRead_signal
memWrite_signal
mem_address[15..0]
write_data[15..0]
read_data[15..0]
mux_2_x_1:muxb
A[15..0]
B[15..0]
S
D[15..0]
mux_2_x_1:muxc
A[15..0]
B[15..0]
S
D[15..0]
result[15..0]
instr_mem:instructions
pc[31..0]
instruction[31..0]


control_unit:signal
a15
bne_signal
1'h0
jump_signal
a13
beq_signal
1'h0
a11
memRead_signal
1'h0
a12
memWrite_signal
a10
memtoReg
le[5..0]
a14
ALUop[1..0]
a4
a5
o3
a6
a7
a0
a2
o1
ALUSrc
a3
a8
o2
regWrite_signal
o0
regDest
a1
o4
set_signal
ll_signal


alu_control:cont
ALUOp[1..0]
res1
WideAnd0
comb~0
ALUControlOp[2..0]
5:0
funcfield[5..0]
res4
WideAnd1
comb~2
comb~1
res3
mux_2_x_1:mux1

Modules

```
module ProgramCounter(nextPC,clock,jump_signal,branch_signal,CurrentPC);
  input clock,branch_signal,jump_signal;
  input [31:0] CurrentPC;
  output reg [31:0] nextPC;

  always @ (posedge clock) begin
      if(jump_signal==1) begin
         nextPC = CurrentPC;
      end

      else if(branch_signal == 1) begin
         nextPC = nextPC + CurrentPC;
      end
      else begin
         nextPC = nextPC+4;
      end
  end
endmodule
```

Input olarak 32 bit ProgramCounter adresi alır. Ek olarak “jump” ve “branch” sinyalleri alır. Jump ise input olarak gelen adresi nextPc adresi yapar.(Jump ise curentpc mips16_single_cycle modülünden jump yapılmak istenen adres olarak gönderilir.) Branch ise nextPC adresi branch adresidir.Bu yüzden currentpc ile toplanarak nextPC adresine atanır branch.Program counter instructionun atlamak istediği yeri gösterir. Bu sinyaller modülde var ama instructionları implement edemediğim için işlevleri yok.

```
module instr_mem(instruction ,pc);
input [31:0] pc;
output reg [31:0] instruction;


reg [31:0] instr_mem [255:0];

   always @(*) begin
      instruction = instr_mem[pc];
   end

endmodule

```

Instruction Memory modülü.Input olarak ProgramCounter alır ve output olarak ProgramCounter’ın gösterdiği adresteki 32 bit Instruction’ı verir.

```
module data_mem (read_data, mem_address, write_data, memRead_signal, memWrite_signal,clock);
output reg [15:0] read_data;
input [15:0] mem_address,write_data;
input memRead_signal,memWrite_signal,clock;

reg [15:0] data_mem  [255:0];


always @(*) begin
   if (memRead_signal) begin
       read_data[15:0] = data_mem[mem_address];
   end
end


always @(posedge clock) begin
   if (memWrite_signal) begin
      data_mem[mem_address] = write_data[15:0];
   end
end

endmodule


```

Input olarak yazılacak ya da okunacak adres, clock ve memRead ve memWrite sinyalleri alır. Output verme durumu yalnızca load instructionlarında olacağından yani okunan datanın dışarı verileceği zamanda olur.Bu yüzden bir tane output vardır.

```
module mips_registers( read_data_1, read_data_2, write_data, read_reg_1, read_reg_2, write_reg, signal_reg_write,clk);

   output [15:0] read_data_1, read_data_2;
   input [15:0] write_data;
   input [4:0] read_reg_1, read_reg_2, write_reg;
   input signal_reg_write, clk;

   reg [15:0] registers [15:0];


   assign read_data_1 = registers[read_reg_1];
   assign read_data_2 = registers[read_reg_2];

   always@(posedge clk)
   if( signal_reg_write && write_reg!=5'b0) begin
       registers[write_reg] = write_data;
      end

endmodule


```

Registerlar arasından input olarak gelen “read_data” adreslerinden 16-bit content output olarak verilir. “signal_reg_write” 1 ise ve “write register” zero registerı değilse input olarak gelen write data clock değişiminde yine input olarak gelen “write_reg” adresine yazılır.

```
module control_unit(opcode,memRead_signal,memWrite_signal,regWrite_signal,signExtend_signal,zeroExtend_signal,
jump_signal,beq_signal,regDest,memtoReg,ALUSrc,set_signal,bne_signal,ALUop,li_signal);

    input [5:0] opcode;
    output memRead_signal,memWrite_signal,regWrite_signal,signExtend_signal,zeroExtend_signal,regDest,jump_signal,
    beq_signal,memtoReg,ALUSrc,set_signal,bne_signal,li_signal;
    output [1:0] ALUop;
    wire sw_signal;

    wire n0,n1,n2,n3,n5;
    wire t1,t2,t3,t4,t5;
    not(n0,opcode[0]);
    not(n1,opcode[1]);
    not(n2,opcode[2]);
    not(n3,opcode[3]);
    not(n4,opcode[4]);
    not(n5,opcode[5]);

    wire R_and, lw_and, sw_and, beq_and,bne_and, slt_and,slti_and,addi_and;
    and a0(li_signal,n5,n4,opcode[3],opcode[2],opcode[1],opcode[0]);
    and a1(R_and,n5,n4,n3,n2,n1);
    and a2(lw_and,opcode[5],n4,n3,n2,opcode[1],opcode[0]);
    and a3(sw_and,opcode[5],n4,opcode[3],n2,opcode[1], opcode[0]);
    and a4(beq_and,n5,n4,n3,opcode[2],n1,n0);
    and a5(bne_and,n5,n4,n3,opcode[2],n1,opcode[0]);
    and a6(slt_and,n5,n4,n3,n2,n1,n0);
    and a7(slti_and,n5,n4,opcode[3],n2,opcode[1],n0);
    and a8(addi_and,n5,n4,opcode[3],n2,n1,n0);

    or o0(regDest,R_and,li_signal);
    or o1(ALUSrc,lw_and,sw_and,slti_and,addi_and,li_signal);
    and a10(memtoReg,opcode[5],n4,n3,n2,opcode[1],opcode[0]);
    or o2(regWrite_signal,R_and,lw_and,slt_and,slti_and,addi_and,li_signal);
    and a11(memRead_signal,opcode[5],n4,n3,n2,opcode[1],opcode[0]);
    and a12(memWrite_signal,opcode[5],n4,opcode[3],n2,opcode[1], opcode[0]);
    and a13(beq_signal,n5,n4,n3,opcode[2],n1,n0);
    and a14(ALUop[1],n5,n4,n3,n2,n1);
    or o3(ALUop[0],beq_and,bne_and,slt_and,slti_and);
    or o4(set_signal,slt_and,slti_and);
    and a15(bne_signal,n5,n4,n3,opcode[2],n1,opcode[0]);

    endmodule
```

```
module alu_control(input [1:0] ALUOp, input [5:0] funcfield, output [2:0] ALUControlOp);
   wire ALUOp0n, ALUOp1n, funcfield2n, res1, res2, res3, res4, res5;
   not(ALUOp0n, ALUOp[0]);
   and(res1, ALUOp[1], ALUOp[0]);
   and(res2, ALUOp[1], ALUOp0n, funcfield[0]);
   or(ALUControlOp[0], res1, res2);

   not(ALUOp1n, ALUOp[1]);
   not(funcfield2n, funcfield[2]);
   and(res3, ALUOp0n, funcfield2n);
   or(ALUControlOp[1], res3, ALUOp1n);

   and(res4, ALUOp1n, ALUOp[0]);
   and(res5, ALUOp[1], ALUOp0n, funcfield[1]);
   or(ALUControlOp[2], res4, res5);
endmodule
```

```
module alu32(input [15:0] A,input [15:0] B, input [2:0] ALUop, output reg [15:0] result, output zero);
    wire [15:0] A_inv, B_inv, A_xor_B, A_and_B, A_or_B;
    reg [15:0] A_slt_B; // Declare A_slt_B as reg type
    // Invert A and B
    not (A_inv, A);
    not (B_inv, B);

    // Perform XOR operation
    xor (A_xor_B, A, B);

    // Perform AND operation
    and (A_and_B, A, B);

    // Perform OR operation
    or (A_or_B, A, B);

    // Perform SLT operation
    always @(*) begin
        A_slt_B = (A < B) ? 16'b1 : 16'b0;
    end
    // Multiplexer to select the operation
    always @(*) begin
        case (ALUop)
            3'b000: result <= A + B;
            3'b001: result <= A - B;
            3'b011: result <= A_xor_B;
            3'b100: result <= A_and_B;
            3'b101: result <= A_or_B;
            3'b110: result <= A_slt_B;
            3'b111: result <= A_inv & B_inv;
        endcase
    end

    // Output zero signal
    assign zero = (result == 16'b0);
endmodule
```

***Not**: Çeşitli sebeplerden dolayı zamanında yetiştiremediğim bu ödevden kısmen puan alabilmek için bazı eksiklerle yolladım. Bilgisayarımda kurulu olan Quartus programındaki sıkıntıdan dolayı testbench yazmakta ve RAM portu kullanmakta sorun yaşadım bu yüzden raporumda bunlara ait testler ve screenshot'lar mevcut değil. Yine aynı sebeplerle projenin pdf'inde verilen instructionlardan jump instructionlarını implement edemedim.*