浙江大学

本科实验报告

课程名称：	计算机组成
姓　名：	Parsa 帕萨
学　院：	计算机科学与技术学院
系：	计算机科学与技术系
专　业：	中加班
学　号：	3170300180
指导教师：	姜晓红

2019 年　　04 月　　22 日

浙江大学实验报告

课程名称： 计算机组成 实验类型： 综合

实验项目名称： Lab 3: Designing a single cycle mips cpu with datapath and controllers

学生姓名： Parsa 帕萨 专业： 中加班 学号： 3170300180

同组学生姓名： None 指导老师： 姜晓红

实验地点： 东 4-510 实验日期： 2019 年 04 月 22 日

1 – Purpose of the Experiments and Tasks

i. Design the controllers for each instruction set
ii. Design an ALU
iii. Design the cpu and datapath

2 – Instructions and Experiment Contents

Mips instruction sets has reserved 32 bits for each instruction. Instructions are divided into 3 different categories, I-Type, J-Type and R-Type. Each of these formats has a different structure when it comes to instructions.

R-Type or register format is used whenever we are dealing with variables that are stored in the registers. This type has the following format:

Operation code	rs	rt	rd	shift	function
6-bits	5-bits	5-bits	5-bits	5-bits	6-bits

i.e. let's assume we want to add to registers together. We have the following instruction:

add $t0, $t1, $t2

In this case, add is the operation we want to perform. It has an operation code of 0 and function code of 20 in hex. Rs and rt are our source registers and rd is our destination register; rs and rt are t1 and t2 and rd

is t0. Shift is used for sll and srl instructions which indicates how many bits we want to do the shifting.

I-Type: This instruction set is used when we are dealing with immediate values. It has a similar structure as R-type except we no longer need a secondary source register, instead we need an immediate. The structure of this type is as follows:

Operation code	rs	rt	immediate
6-bits	5-bits	5-bits	16-bits

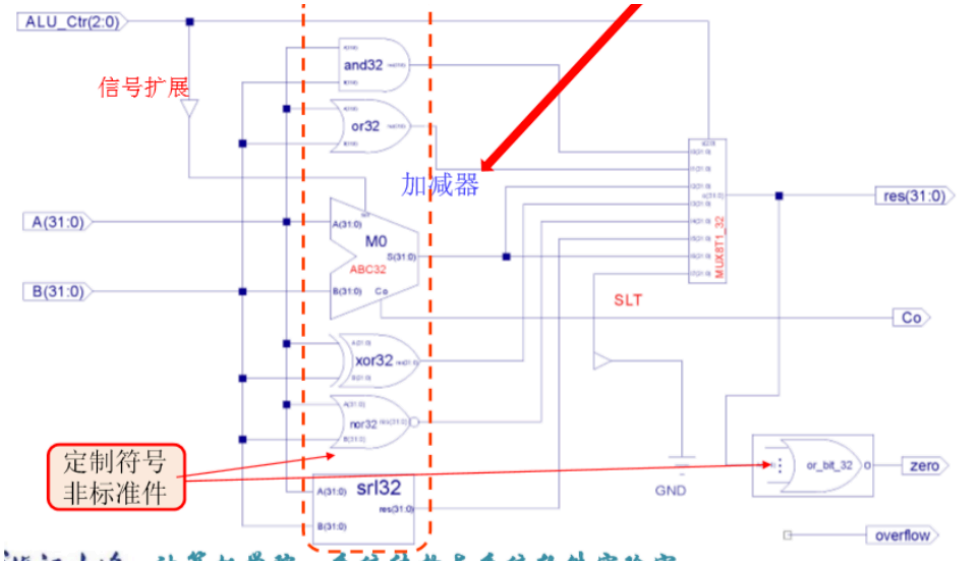
Rs is the source register and rt is the target register.

J-Type: This type is used when we are dealing with jump instructions. They take a operation code and an address. Its structure is as follows:

Operation code	Address
6-bits	26-bits

3 – Implementations

ALU Setup:



Alu is required to be able to perform the following operations: addition, subtraction, and, or, xor, nor, srl and slt. It is controlled by the signal coming from ALU_ctrl which controls which signal is passed through to the results. Co is the carryout of the addition, and zero is used for SLT operation. Here is the implementation of the top module and its children modules.

```
module alu32(input [2:0]ctrl,
             input [31:0] a,
             input [31:0] b,
             output co,
             output overflow,
             output zero,
             output [31:0]res
    );
   wire [31:0] and32out, or32out, xor32out,
               nor32out, srl32out, addsub32out;
   and32 mod0(a,b, and32out);
   or32 mod1(a,b, or32out);
   xor32 mod2(a,b, xor32out);
   nor32 mod3(a,b, nor32out);
   srl32 mod4(a,b, srl32out);
   addsub32 mod5(ctrl[2], a,b,co,addsub32out);
   mux8to1 mod6(.crtl(ctrl),.and32(and32out),.or32(or32out),
                .add32(addsub32out), .xor32(xor32out),
                .nor32(nor32out),.srl32(srl32out),
                .sub32(and32out),.slt({31'b0,addsub32out[31]}),.out(res));
   assign zero = (res==32'b0)?1:0;

endmodule
```

Figure 1 32-bit ALU top module

```
module and32(input [31:0]a,
             input [31:0]b,
             output [31:0]out
    );
   assign out = a & b;

endmodule
```

Figure 2 32-bit And module

```
module or32(input [31:0] a,b,
            output [31:0] out
    );
   assign out = a|b;

endmodule
```

Figure 3 32-bit Or module

```
module xor32(input [31:0]a,
             input [31:0]b,
             output [31:0]out
    );
   assign out = a ^ b;

endmodule
```

Figure 4 32-bit XOr module

```
module nor32(input [31:0]a,
             input [31:0]b,
             output [31:0]out
    );
   assign out = ~(a|b);


endmodule
```

Figure 5 32-bit NOr module

```
module srl32(input [31:0]a,
             input [31:0]b,
             output [31:0]out
    );

   assign out = b>>1;

endmodule
```

Figure 6 32-bit Srl module

```
module addsub32(input sub,
                input [31:0] a,
                input [31:0] b,
                output co,
                output [31:0] res
    );
    wire [31:0] operand = sub==1? b ^ 1 : b;
    add32 mod0(.a(a), .b(operand), .cin(sub),
               .co(co), .res(res));


endmodule
```

Figure 7 32-bit add/sub module

```
module add32(input [31:0] a,
             input [31:0] b,
             input cin,
             output [31:0] res,
             output co,
             output overflow
     );
    assign {co,res} = a+b+cin;

endmodule
```

Figure 8 32-bit 2's complement addition module

```
module mux8to1(input [2:0]crtl,
               input [31:0]and32,
               or32,
               add32,
               xor32,
               nor32,
               srl32,
               sub32,
               slt,
               output reg [31:0] out
    );

    always @* begin
      case(crtl)
         3'd0: out = and32;
         3'd1: out = or32;
         3'd2: out = add32;
         3'd3: out = xor32;
         3'd4: out = nor32;
         3'd5: out = srl32;
         3'd6: out = sub32;
         3'd7: out = slt;
      endcase
    end

endmodule
```

Figure 9 Multiplexer used for ALU

Once we have all the components of the ALU designed we can focus on the other components of the DataPath. These components are program counter, Sign extension, and registers. Program counter keeps track of the current instruction, sign extension is used for I-Type

instructions since our ALU only supports 32-bits operands. Registers module is a module that houses all the registers needed for our "CPU". Here are the implementations of these modules:

```
module p_counter(input clk,
                 input rst,
                 input en,
                 input [31:0] a,
                 output [31:0] res
    );

   reg [31:0] counter;
   always @(posedge clk or posedge rst)
      if (rst==1) counter = 32'b0;
      else if (en == 1) counter = a;

   assign res = counter;

endmodule
```

Figure 10 Program counter module

```
module sign_ext(input [15:0] a,
                output [31:0] res
    );
   assign res = a[15]==1?{16'hffff,a}:{16'h0,a};
endmodule
```

Figure 11 Sign Extension module

```
module registers(input clk,
                 input rst,
                 input [4:0] Addr_A_R,
                 input [4:0] Addr_B_R,
                 input [4:0] Write_Addr,
                 input [31:0] Write_data,
                 input R_or_W,
                 output [31:0] A_data,
                 output [31:0] B_data
    );

   reg [31:0] regs [0:31];
   assign A_data = regs[Addr_A_R];
   assign B_data = regs[Addr_B_R];
   always @(posedge clk or posedge rst)begin
      if (rst == 1)
         for (integer i = 0; i<32; i = i+1)
            regs[i] = 32'b0;
      else
         if ( R_or_W == 1 && Write_Addr!=0 )
            regs[Write_Addr] = Write_data;
   end
endmodule
```

Figure 12 Registers module

In registers modules, we have 32 registers with each having a length of 32-bits. Depending on the incoming signals we can either read from the registers or write to them, or reset all 32 registers.

Following the diagram below we can implement the datapath needed for our single cycle cpu:

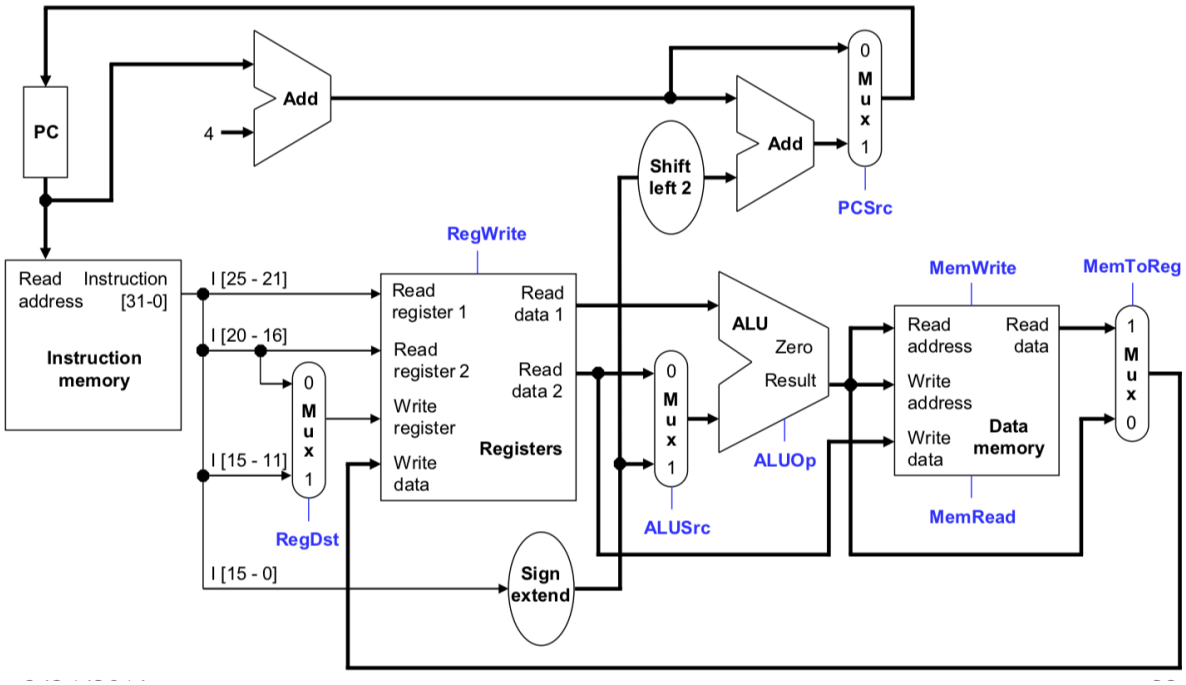


Figure 13 Datapath diagram (University of Pittsburgh)

Notice that in this diagram, we have a memory (RAM) module. We already have it implemented so there is no need to include it in our datapath, the output of our Read data 2 can be outputted as a bus called data_out; similarly, we can output the output of ALU as alu_out. The input of write is the output of a multiplexer with memtoreg as a switch and data_in (output of RAM) and alu_out. This diagram is missing J-Type instruction set. For that we are required to add two inputs: branch and jump. Branch is used for if statement-styled instructions and jump is used for J-Type instructions (Jr, Jl and etc.). We add a new multiplexer to this diagram that takes the jump address and if jump input is enabled then we set out pc to be the jump address otherwise it'd be pc+4 or address of branch. Here is the implementation of the Datapath:

```
module datapath(input jump,
                input clk,
                input rst,
                input wr,
                input branch,
                input [2:0]alu_ctrl,
                input alu_src_b,
                input reg_dst,
                input [25:0]inst_field,
                input memtoreg,
                input [31:0] data_in,
                output [31:0] pc_out,
                output [31:0] alu_out,
                output [31:0] data_out
    );
      wire z;
      wire [31:0] branch_offset, pc_plus_4, branch_pc, mux_1, mux_2;

      p_counter mod0(clk, rst, 1'b1, mux_2,pc_out);
      assign pc_plus_4 = 3'd4+ pc_out;

      sign_ext mod1(inst_field[15:0], branch_offset);
      assign branch_pc = pc_plus_4 + branch_offset;
      assign mux_1 = z&branch == 0?pc_plus_4:branch_pc;

      wire [31:0] jump_addr = {pc_plus_4[31:28], inst_field[25:0], 2'b0};
      assign mux_2 = jump == 0? mux_1: jump_addr;


      wire [4:0] write_addr = reg_dst==0?inst_field[20:16]:inst_field[15:11];
      wire [31:0] write_data = memtoreg==0?alu_out:data_in;
      wire [31:0] A_data, B_data;

      registers mod2(.clk(clk), .rst(rst), .Addr_A_R(inst_field[25:21]),
                     .Addr_B_R(inst_field[20:16]), .Write_Addr(write_addr),
                     .Write_data(write_data), .R_or_W(wr),.A_data(A_data),
                     .B_data(B_data));

      wire [31:0] inst_field_32, mux_3;
      sign_ext mod3(inst_field[15:0], inst_field_32);

      assign data_out = B_data;
      assign mux_3 = alu_src_b==0?B_data:inst_field_32;

      alu32 mod4(.ctrl(alu_ctrl), .a(A_data), .b(mux_3), .zero(z),
                 .res(alu_out));

endmodule
```

Figure 14 implementation of datapath

According to this diagram, controller module is the module that creates the all the signals required for our datapath module.

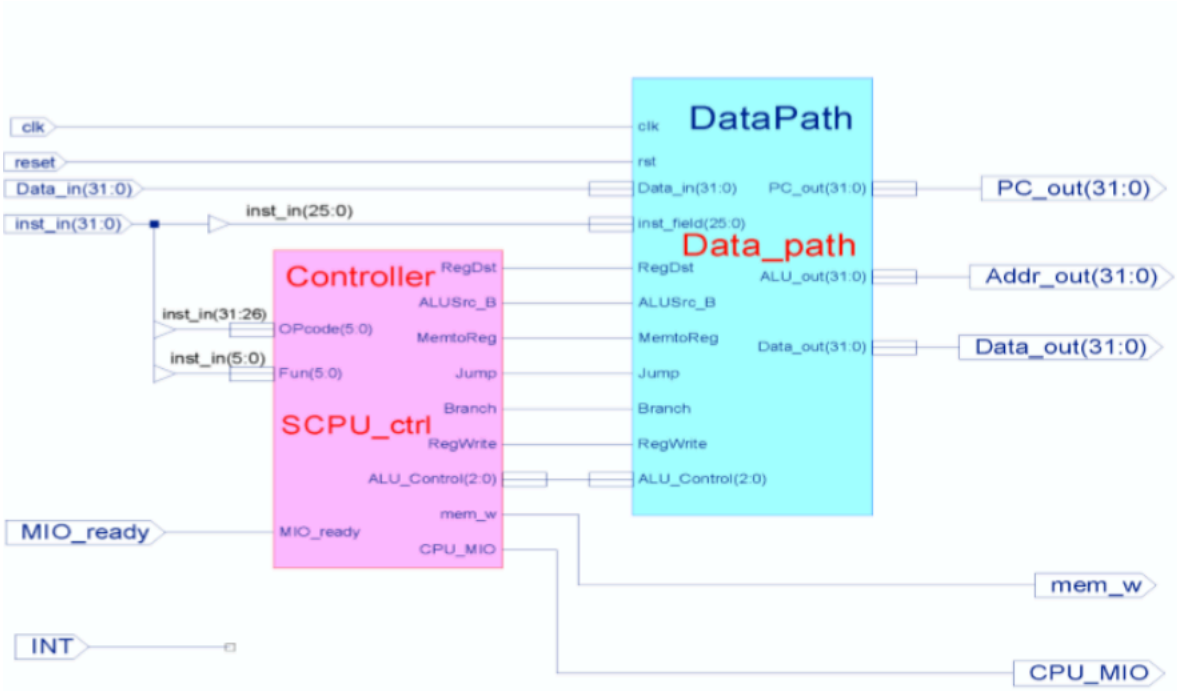


Figure 15 Schematics of the single cycle cpu

The implementation of controller module is pretty straight-forward. Branch is enabled if all 6 bits of the instructions except the second one are 0s. Regdst is the signal that chooses where data is written to and is enabled if all 6 bits are not enabled. For the rest, they can be seen in the controller module and are pretty easy to understand. Here is the implementation of this module:

```
module controller(input [5:0] op,
                  input [5:0] func,
                  input MIO_ready,
                  output [2:0] alu_crtl,
                  output branch,
                  output regw,
                  output regdst,
                  output memtoreg,
                  output alusrc_b,
                  output mem_w,
                  output jump
    );
    assign branch = ~op[5] & ~op[3] & op[2] & ~op[1] & ~op[0];
    assign regdst = ~(op[0]|op[1]|op[2]|op[3]|op[4]|op[5]);
    assign mem_w = op[0]& op[1]&~op[2] &op[3] &op[5];
    assign jump = op[0]& ~op[1]&~op[2] &~op[3] &~op[5];
    assign memtoreg = op[0]& op[1]&~op[2] &~op[3] &op[5];
    assign regw = memtoreg|regdst;
    assign alusrc_b = mem_w|memtoreg;
    assign alu_crtl[2] = branch | (regdst&func[1]);
    assign alu_crtl[1] = ~(regdst&func[2]);
    assign alu_crtl[0] = regdst&(func[3]&(~func[1]&func[0]));

endmodule
```

Figure 16 Controller module

4 – Results

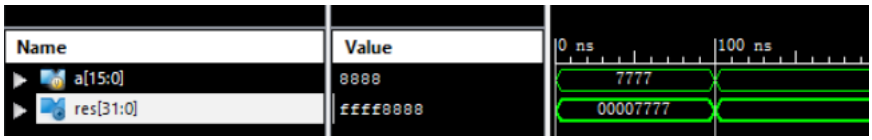


Figure 17 results of sign extension

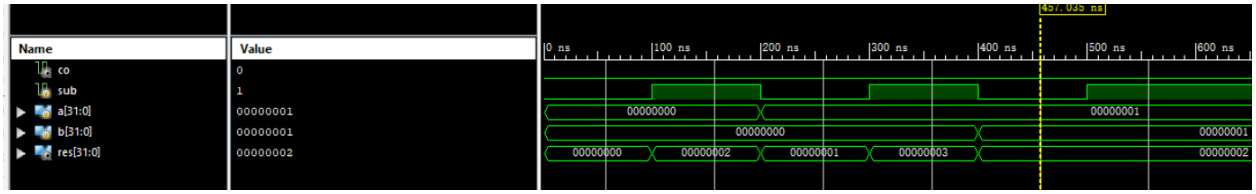


Figure 18 Adder and subtraction modules

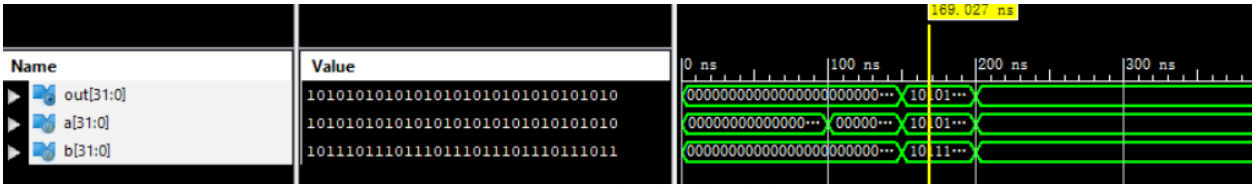


Figure 19 results of And module

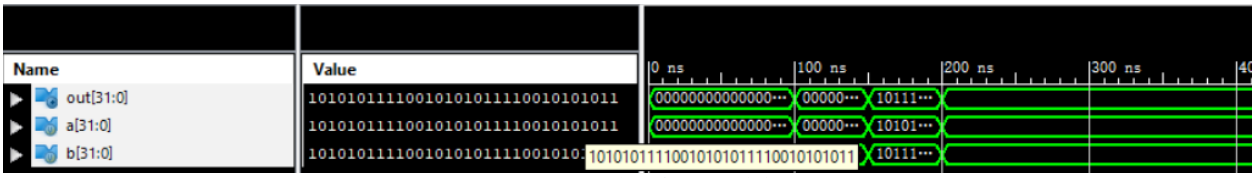


Figure 20 results of OR module

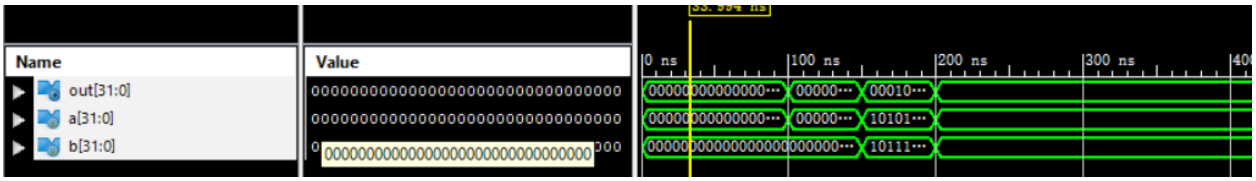


Figure 21 Results of XOR module

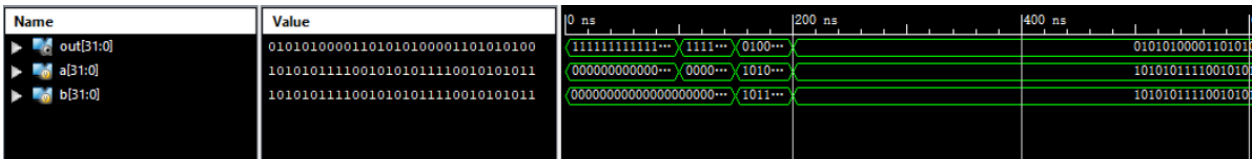


Figure 22 Results of Nor module

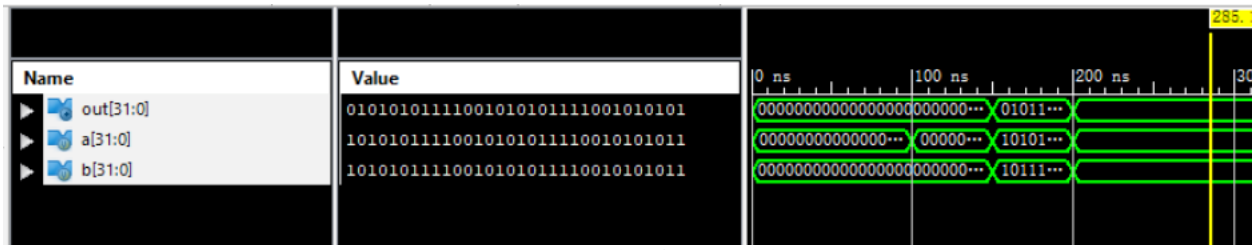


Figure 23 Results of SRL module