CSC3050 Project4 Report: Design of a MIPS CPU

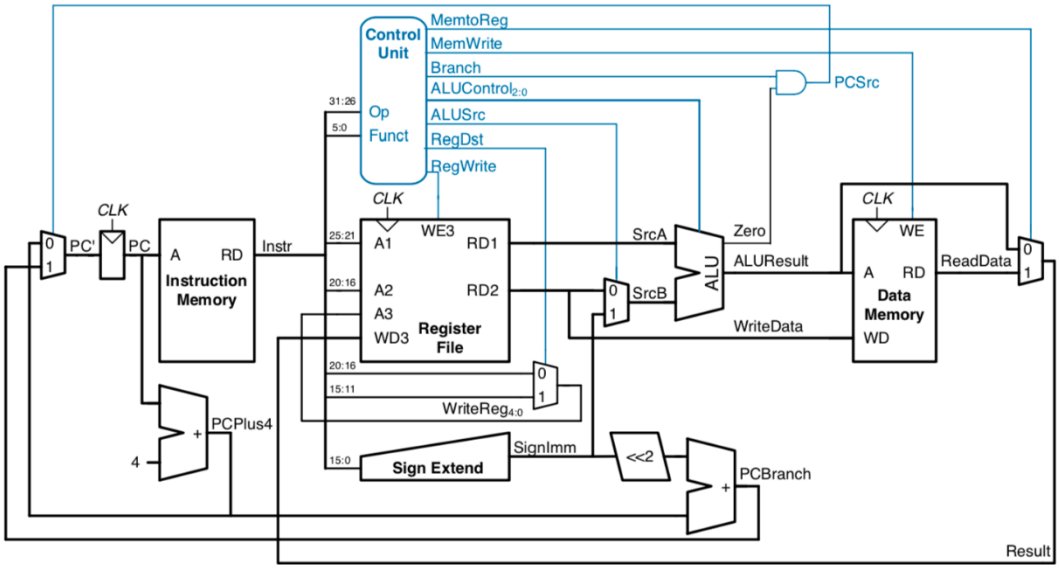
杨宇凡 117010349

Introduction:

For the project4, design of a MIPS CPU, I have designed two kinds of CPU. One is a simple single-cycled CPU and another is a pipeline CPU. I have submitted both of them for the checking. In this report, I will also introduce both of them, but mainly focus on the design of pipeline CPU. The design of a single-cycled CPU will be briefly introduced. The next part is about the design of a single-cycled CPU.

Single-Cycled CPU

Design



The basic block diagram of single-cycled CPU is just like the below picture. However, according to the Verilog language characteristic, some units are omitted in the real design. Then I'd like to introduce the design of my single-cycled CPU.

```
module CPU(
    input wire clock,
    input wire start,
    input [31:0] i_datain,
    input wire [31:0] d_datain,
    output wire [31:0] d_dataout
    );
```

```
reg [31:0]gr[31:0];
reg [15:0]pc = 16'h0000;

reg [31:0]reg_A;
reg [31:0]reg_B;
reg [31:0]reg_C;
reg [31:0]reg_C1;
reg [31:0]saveword;
```

```
reg [5:0]opcode;
reg [5:0]functcode;
reg [31:0]instruction;

reg overflow;
```

Below pictures show the main components of the single-cycled CPU. It has four inputs. Clock is used to control the clock cycle, start is used to trigger CPU, i_datain is used to represent the instruction in one cycle time, and d_datain is used to represent the input

memory data in one cycle time. It also has an output d_dataout which represents the data which will be stored to memory (only of sw instruction). Besides, there are many registers to store the contemporary data or control the whole CPU. Register gr represents 32 general registers, pc represents the counter, reg_A and reg_B are the inputs of the ALU, reg_C is the result of the ALU, reg_C1 is the result of the whole CPU, and saveword is used to store the data which will be transported to memory contemporarily. Opcode is used to store the real word from the instruction, as well as the functcode, and register instruction is used to store the instruction. Register overflow is used to store the overflow check result.

Logic

1) Step 1: fetch the opcode and function code

```
overflow=1'b0;
//control unit
opcode = i_datain[31:26];
functcode = i_datain[5:0];
```

This step gets the opcode and function code from the input instruction, which will in a key part in the following steps.

2) Step 2: get the input data for ALU unit

```
//reg_A and reg_B
//lw,sw,addi,addiu
if (opcode==6'b100011 || opcode==6'b101011 || opcode==6'b001000 ||opcode==6'b001001)
begin
    reg_A=gr[i_datain[25:21]];
    reg_B={{16{i_datain[15]}},i_datain[15:0]};
end
//andi,ori
else if (opcode==6'b001100 || opcode==6'b001101)
begin
    reg_A=gr[i_datain[25:21]];
    reg_B={16'b0,i_datain[15:0]};
end
//beq,bne
else if (opcode==6'b000100 || opcode==6'b000101)
begin
    reg_A=gr[i_datain[25:21]];
    reg_B=gr[i_datain[20:16]];
end
//R_type
```

This step gets the correct inputs for the ALU for further calculation according to the opcode and function code. The inputs are stored in reg_A or reg_B separately. Some of the inputs are from the general registers, and some are immediate number from the instruction directly.

3) Step3: ALU Execution

```
//R_type
else if (opcode == 6'b000000)
begin
    //add
    if (functcode==6'b100000)
```

```
//reg_C
//lw, sw
if (opcode==6'b100011 || opcode==6'b101011)
    reg_C=reg_A +reg_B;
//addi
else if (opcode==001000)
begin
    reg_C=$signed(reg_A)+$signed(reg_B);
    if ((reg_A[31]==reg_B[31]) && (reg_A[31]!=reg_C[31]))
        overflow=1'b1;
    else overflow=1'b0;
end
//addiu
else if (opcode==001001)
    reg_C=$unsigned(reg_A)+$unsigned(reg_B);
//andi
else if (opcode==001100)
    reg_C=reg_A&reg_B;
//ori
else if (opcode==001101)
    reg_C=reg_A|reg_B;
//beq
else if (opcode==000100)
begin
    if (reg_A==reg_B)
        reg_C=i_datain[15:0];
end
```

This step does the ALU execution according to the opcode and function code. The execution result is stored in reg_C.

4) Step4: Decide the final result

```
//reg_C1(Result)
//lw
if (opcode == 6'b100011)
begin
    reg_C1 = d_datain[31:0];
end
//sw
else if (opcode==6'b101011)
    reg_C1=gr[i_datain[20:16]];
//R_type,addi,addiu,andi,ori
else if (opcode==6'b000000 || opcode==6'b001000 || opcode==6'b001001 || opcode==6'b001100 || opcode==6'b001101)
    reg_C1=reg_C;
```

The ALU result stored in reg_C may not be the final result of the CPU because some result needs to be fetched from the memory or somewhere else. Therefore, we have to select the final CPU result according to the opcode and function code. The selected final result is stored in reg_C1.

5) Step5: Write back the result to register

```
//write back to general registers
//lw
if (opcode == 6'b100011)
    gr[i_datain[20:16]] = reg_C1;
//sw
else if (opcode==6'b101011)
    saveword=reg_C1;
//addi,addiu,andi,ori
else if (opcode==6'b001000 || opcode==6'b001001 || opcode==6'b001100 || opcode==6'b001101)
    gr[i_datain[25:21]]=reg_C1;
//R_type
else if (opcode == 6'b000000 && functcode!=6'b001000)
    gr[i_datain[15:11]]=reg_C1;
```

Then we have to decide whether to write the result back to general registers according to the opcode and function code. The position of the general registers also depends on the opcode and function code. Specially, for the save word operation (sw), the result is stored in the saveword register contemporarily, and then it will be transported to d_dataout.

6) Step6: PC control signal update

```
    //pc
    //jr
    if (opcode==6'b001000)
        pc<=pc+4+reg_C*4;
    //j
    else if (opcode==6'b000010)
        pc<=pc+4+i_datain[25:0]*4;
    //jal
    else if (opcode==6'b000011)
    begin
        gr[31]=pc+4;
        pc<=pc+4+i_datain[25:0]*4;
    end
    //beq
    else if (opcode==000100)
    begin
        if (reg_A==reg_B)
            pc<=pc+4+reg_C*4;
    end
    //bne
    else if (opcode==000101)
    begin
        if (reg_A!=reg_B)
            pc<=pc+4+reg_C*4;
    end
    else pc <= pc + 16'h0004;
end
```

The final step is updating the PC control signal according to the opcode. For the jump and branch operation, PC control signal will be changed specially. For other operations, it is traditionally plus four.

Besides, the first general register, gr[0] will be initialized as zero at first.

```
always @(start)
    begin
        gr[0] = 32'h0000_0000;
    end
```

Test

```
    // Initialize Inputs
clock = 0;
start = 0;
d_datain = 0;
i_datain = 0;
$display("pc  :       instruction            : reg_A  : reg_B  : reg_C  : reg_C1 :d_datain:  gr0   :  gr1   :  gr2   :  gr3");
$monitor("%h:%b:%h:%h:%h:%h:%h:%h:%h:%h:%h",
    uut.pc, i_datain, uut.reg_A, uut.reg_B, uut.reg_C, uut.reg_C1, d_datain, uut.gr[0], uut.gr[1], uut.gr[2], uut.gr[3]);

/*Test:*/
//#5 start = 1;
d_datain <= 32'h0000_00ab;
i_datain <= {6'b100011, `gr0, `gr1, 16'h0001};

#period d_datain <= 32'h0000_3c00;
i_datain <= {6'b100011, `gr0, `gr2, 16'h0002};

#period i_datain <= {6'b000000, `gr1, `gr2, `gr3,  5'b00000, 6'b100000};

#period $finish;
end
```

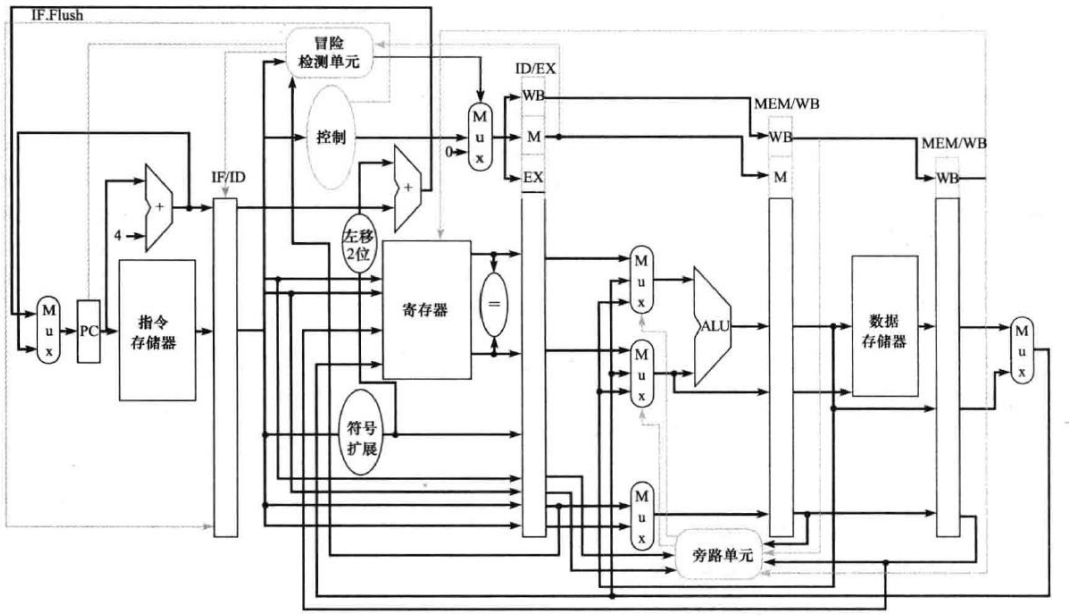
This is the test code for the single cycled CPU, and the result is shown as the following:

```
pc  :        instruction            : reg_A  : reg_B  : reg_C  : reg_C1 :d_datain:  gr0   :  gr1   :  gr2   :  gr3
0000:10001100000000010000000000000001:xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx:000000ab:00000000:xxxxxxxx:xxxxxxxx:xxxxxxxx
0004:10001100000000010000000000000001:00000000:00000001:00000001:000000ab:000000ab:00000000:000000ab:xxxxxxxx:xxxxxxxx
0004:10001100000000010000000000000010:00000000:00000001:00000001:000000ab:00003c00:00000000:000000ab:xxxxxxxx:xxxxxxxx
0008:10001100000000010000000000000010:00000000:00000002:00000002:00003c00:00003c00:00000000:000000ab:00003c00:xxxxxxxx
0008:00000000001000100001100000100000:00000000:00000002:00000002:00003c00:00003c00:00000000:000000ab:00003c00:xxxxxxxx
000c:00000000001000100001100000100000:000000ab:00003c00:00003cab:00003cab:00003c00:00000000:000000ab:00003c00:00003cab
```

As you can see, the red circled data indicate the correctness of the calculation, which is the same as the content in the PowerPoint. Besides, it is welcomed to use other instruction to test.

Pipeline CPU

Design



Below picture indicated the basic block diagram of a pipeline CPU. According to the characteristic of Verilog Language, some units are omitted and some units are added as well. In this design part, the content is divided into the followings: basic components, five main modules, four link registers and control signals.

1) Basic Components

```
module CPU (clock,start,in_instruction);
    input wire clock;
    input wire start;
    input [319:0] in_instruction;

    reg [31:0]gr[31:0];
    reg [32:0]PC = 32'h0000_0000;
    reg [7:0] instruction_memory[1023:0];
    reg [7:0] data_memory[1023:0];

    reg [31:0]instruction;

    reg [5:0]opcode;
    reg [5:0]functcode;
```

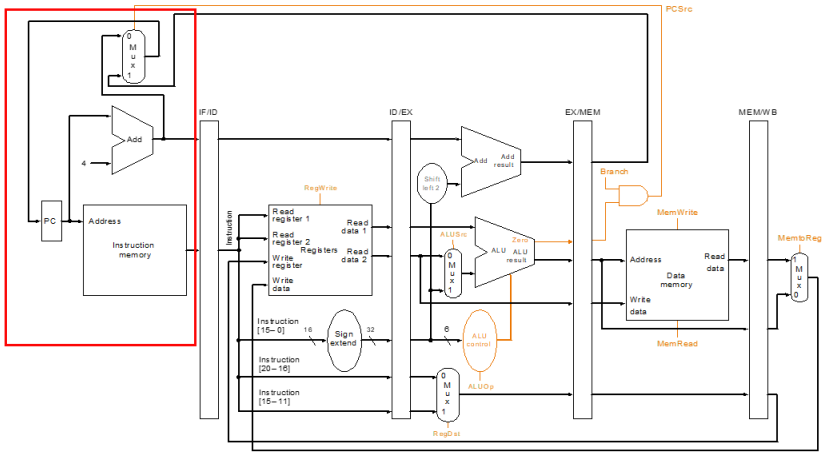
```
always @(start)
    begin
        gr[0] = 32'h0000_0000;
        instruction_memory[0]=in_instruction[319:312];
        instruction_memory[1]=in_instruction[311:304];
        instruction_memory[2]=in_instruction[303:296];
        instruction_memory[3]=in_instruction[295:288];
        instruction_memory[4]=in_instruction[287:280];
```

The pipeline CPU has three inputs. The clock is used to control the running of the whole CPU. The CPU will execute when the clock is up. The start is used to initialize the CPU. The in_insturction is the instruction transported to CPU from the outside. Besides, there are 32 registers representing the general registers, and a register represents the instruction memory with another register represents the data memory. The same PC controller as the single-cycled CPU. There are other small registers work for the five main modules to store contemporary data.

For each initialization, the first general register is set to zero, and the instruction memory will store the instruction from the in_instruction input.

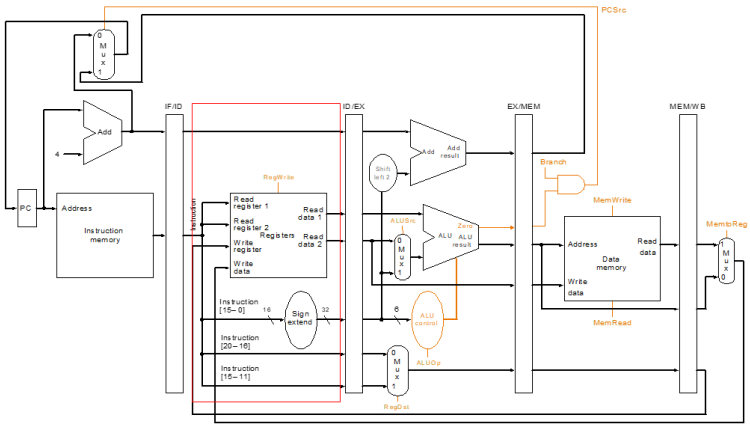
2) Five main Modules

- Instruction Fetch



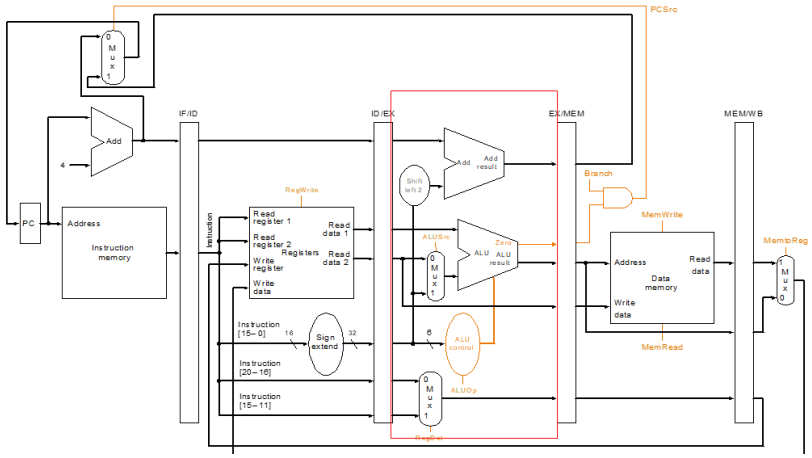
The instruction fetch module is responsible for fetching the instruction from the instruction memory according to the PC controller. Besides, based on the Verilog language characteristic, I also let it to calculate the next address earlier to escape the branch hazard.

- Instruction Decode



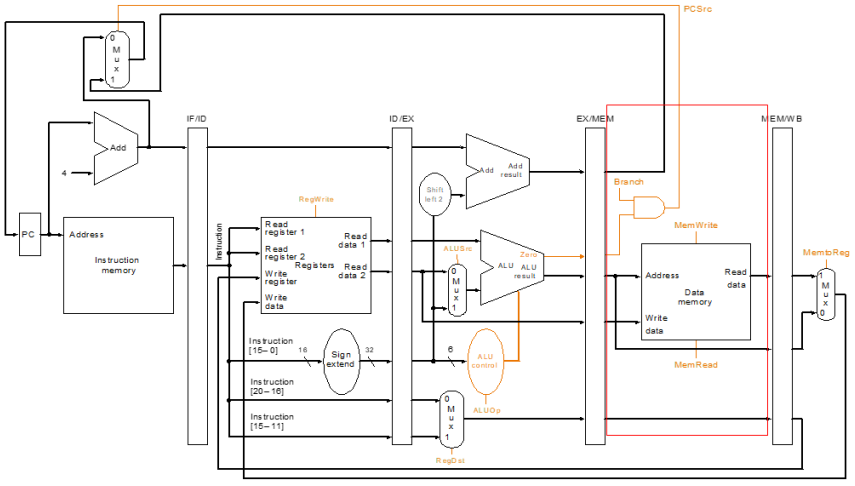
The instruction decode module is responsible for decoding the instruction to get opcode, function code and the address of register which will be used for ALU calculation later. The content of the used register or immediate number is prepared for transporting to ALU in this module. Besides, I also let it to produce and update the control signals because this module is the earliest one which can view the whole part of the instruction. What's more, a hazard detect unit is also designed here, which will be introduced later in details.

- ALU Execution



This module uses ALU to do the ALU calculation based on the two input registers' content and the control signals produced in the Instruction Decode module. The result of the calculation will be transported to the next module for further processing.

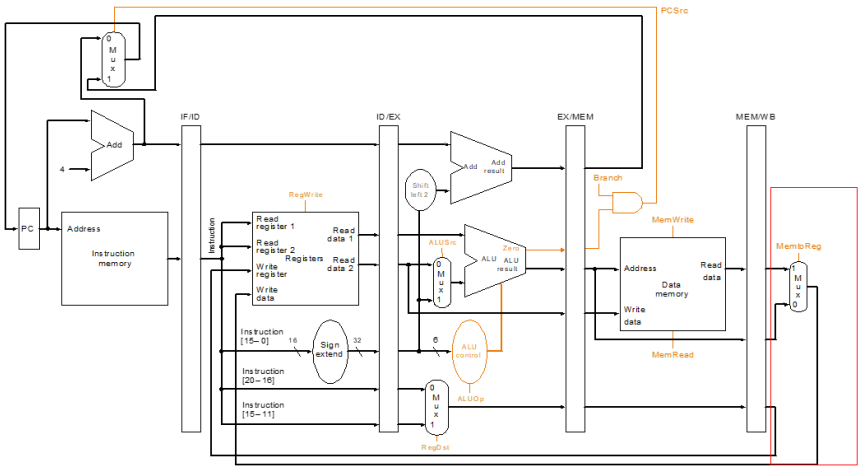
- Memory Operation



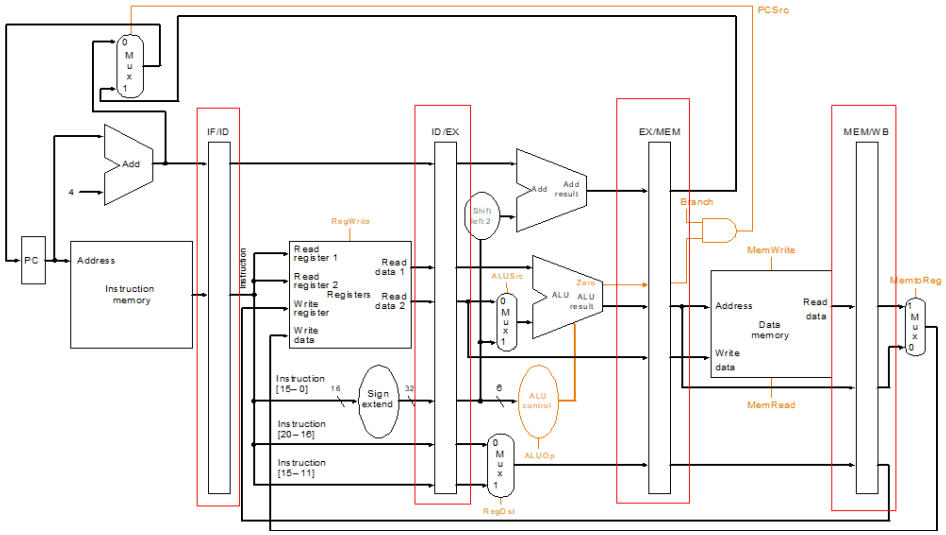
Memory Operation module will do the operation related to the memory (load and write) according to the result of ALU calculation (get address) and the control signals. The memory result and the ALU result will be both transported to the next module

- Write Back

The write back module will write the result from ALU or memory back to the general register according to the control signals. The control signals decide what data to write and where (which register) to write.



3) Four link Registers



In pipeline CPU we have four link registers totally, which are IF/ID, ID/EX, EX/MEM, MEM/WB. They are all responsible for storing the information from the previous module and transporting the information to the next modules. Then I will introduce them separately.

IF/ID is used to store the instruction from Instruction Fetch module and transport it to the Instruction Decode module.

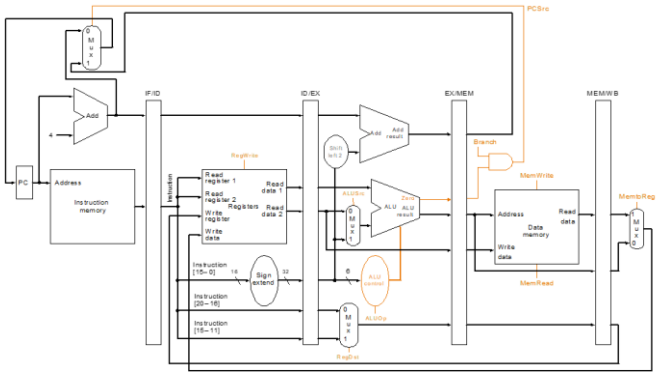
ID/EX is used to store the control signals, two registers' content and address and immediate number from Instruction Decode Module and transport it to ALU Execution module.

EX/MEM is used to store the control signals, ALU calculation result and the register address from the ALU Execution module and transport it to the Memory Operation Module.

MEM/WB is used to store the control signals, ALU calculation result, memory result and the register address from the Memory Operation module and transport it to Write Back Module.

4) Control Signals

To control the whole CPU and make it run stably, totally eight control signals are designed here, and they are represented as registers.



```
reg Regdst;
reg RegWrite;
reg ALUgate;
reg ALUSrc;
reg MemRead;
reg MemWrite;
reg MemtoReg;
reg PC_Control=1'b1;
```

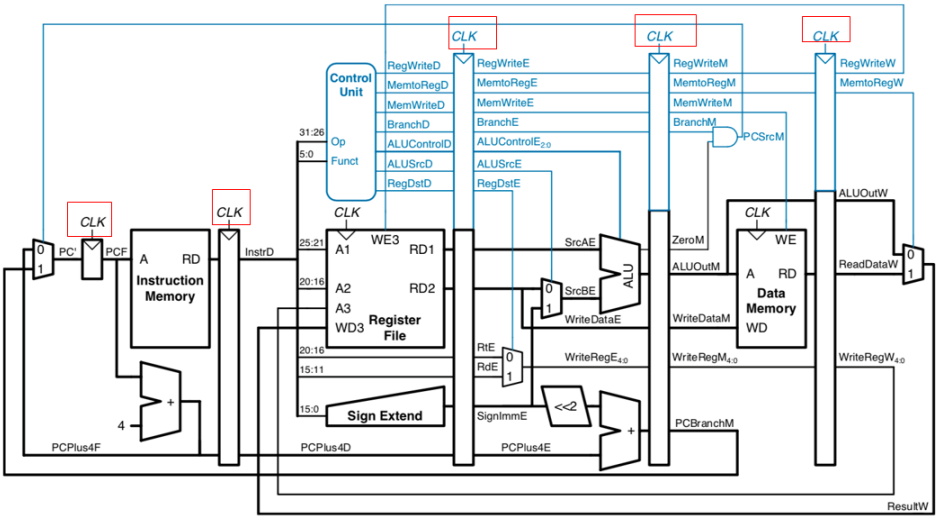
The traditional control signals are indicated as the yellow lines in the picture. However, based on the Verilog language, I used eight registers in the right. Their functions are shown in the following table:

信号名	置无效时的效果（0）	置有效时的效果（1）
RegDst	写入寄存器的目标号来自 rt 字段（20:16 位）	写入寄存器的目标号来自 rd 字段（15:11 位）
RegWrite	无	写入寄存器的源寄存器设置为输入的写入数据
ALUSrc	第二个 ALU 操作数来自第二个寄存器堆的输出（读数据 2）	第二个 ALU 操作数是指令低 16 位的符号扩展
PCSrc	PC 被 PC +4 替代	PC 被分支目标地址替代
MemRead	无	输入地址对应的数据存储器的内容为读数据的输出
MemWrite	无	输入地址对应的数据存储器的内容替换为写数据的输入
MemtoReg	ALU 提供寄存器写数据的输入	数据存储器提供寄存器写数据的输入

For the PC control signal, it is used to control the PC register and the Instruction Fetch Module. When it is 0, Instruction Fetch module is triggered and PC register will be refreshed. When it is 1, Instruction Fetch module is inactive and PC register will stay.

Logic

1) Basic Logic



As you can see, all the five components are all connected to the clock. Therefore, they will be all triggered when the clock is up. We used the five always modules to represents the five main components, and they will be triggered when the clock is

up as well. The five always modules are shown as followings:

```
//Instruction Fetch
always @(posedge clock)
```

```
//Instruction Decoding
always @(posedge clock)
```

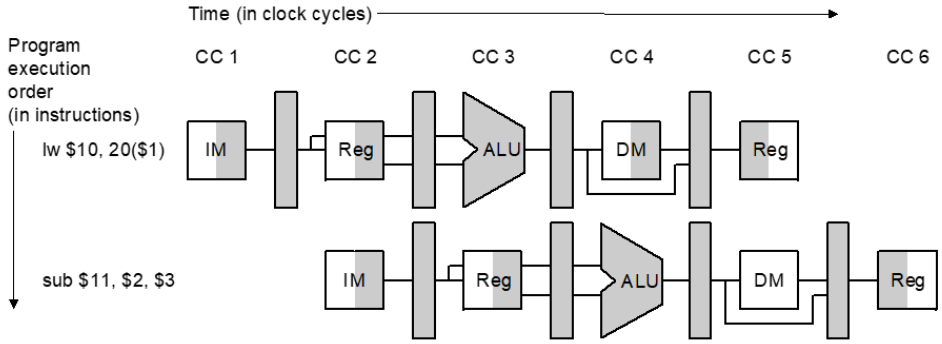
```
//Excution
always @(posedge clock)
```

```
//Memory
always @(posedge clock)
```

```
//Write Back
always @(posedge clock)
```

When the five modules are all triggered, they will fetch the data they need from the four link registers and do the operation. After that, they will store the information to the next link register, and then the whole clock cycle is finished. The data and operation are transmitted in the way like this.

```
reg [63:0] IF_ID;
reg [129:0] ID_EX;
reg [53:0] EX_MEM;
reg [78:0] MEM_WB;
```



2) How to Detect the Hazards

- Structure Hazards

  There will be no structure hazards in this pipeline CPU because all the instructions should follow the operation in the five main modules and the order is the same. Even though some of the operation is useless for them, but they have to wait there to escape the structure hazards.

- Branch Hazards

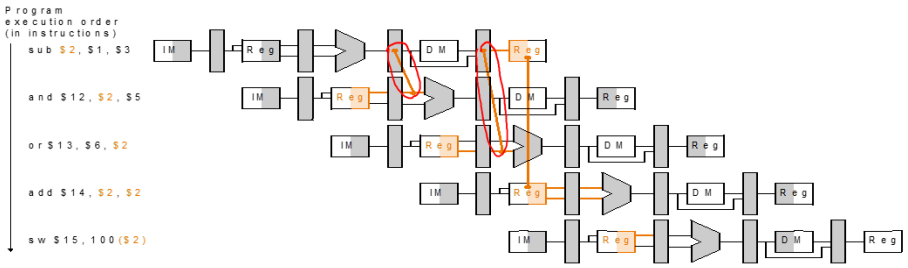
  In this pipeline CPU, the method to solve branch hazards is to make the address calculation earlier. In the traditional pipeline CPU, the calculation of the address is after the ALU calculation, and it's too late and greatly increases the risk of branch hazards. I add a small ALU in the Instruction Fetch module, and it can calculate the address directly and there's no need for waiting. Therefore, it greatly reduces the risk of branch hazards.

```
//Instruction Fetch
always @(posedge clock)
    begin
    if (PC_Control!=1'b0)
    begin
        instruction={instruction_memory[PC],instruction_memory[PC+1],instruction_memory[PC+2],instruction_memory[PC+3]};
        if (instruction[31:26]==6'b000010)
            PC<=PC+4+instruction[25:0]*4;
        else if (instruction[31:26]==6'b000011)
        begin
            gr[31]<=PC+4;
            PC<=PC+4+instruction[25:0]*4;
        end
        else if (instruction[31:26]==6'b000000 && instruction[5:0]==6'b001000)
            PC<=PC+4+gr[instruction[25:21]]*4;
        else if (instruction[31:26]==6'b000100)
        begin
            if (instruction[25:21]==instruction[20:16])
                PC<=PC+4+instruction[15:0]*4;
        end
        else if (instruction[31:26]==6'b000101)
```

(The instruction fetch module can calculate the address by itself)

- Data Hazards

There are two units in this pipeline CPU to detect data hazards. The first one is bypassing unit:



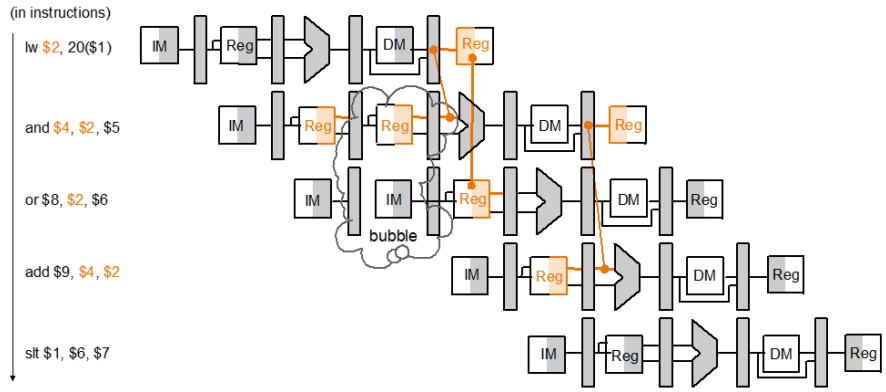

```
//bypassing
//EX_MEM rd = ID_EX rs
if (EX_MEM[41:37]==ID_EX[129:125] && ID_EX[112]==1'b1 && EX_MEM[52]==1'b0)
    ALU_reg1=EX_MEM[31:0];
//EX_MEM rd = ID_EX rt
if (EX_MEM[41:37]==ID_EX[124:120] && ID_EX[112]==1'b1 && ID_EX[111]==1'b0 && EX_MEM[52]==1'b0)
    ALU_reg2=EX_MEM[31:0];
//MEM_WB rd = ID_EX rs
if (MEM_WB[71:67]==ID_EX[129:125] && ID_EX[112]==1'b1 && MEM_WB[77]==1'b0)
    ALU_reg1=MEM_WB[31:0];
//MEM_WB rd = ID_EX rt
if (MEM_WB[71:67]==ID_EX[124:120] && ID_EX[112]==1'b1 && ID_EX[111]==1'b0 && MEM_WB[77]==1'b0)
    ALU_reg2=MEM_WB[31:0];
//EX_MEM rt = ID_EX rs
if (EX_MEM[46:42]==ID_EX[129:125] && ID_EX[112]==1'b1 && EX_MEM[52]==1'b1)
    ALU_reg1=EX_MEM[31:0];
//EX_MEM rt = ID_EX rt
if (EX_MEM[46:42]==ID_EX[124:120] && ID_EX[112]==1'b1 && ID_EX[111]==1'b0 && EX_MEM[52]==1'b1)
    ALU_reg2=EX_MEM[31:0];
//MEM_WB rt = ID_EX rs (ALU)
if (MEM_WB[76:72]==ID_EX[129:125] && ID_EX[112]==1'b1 && MEM_WB[77]==1'b1 && MEM_WB[64]==1'b0)
    ALU_reg1=MEM_WB[31:0];
//MEM_WB rt = ID_EX rt (ALU)
if (MEM_WB[76:72]==ID_EX[124:120] && ID_EX[112]==1'b1 && ID_EX[111]==1'b0 && MEM_WB[77]==1'b1 && MEM_WB[64]==1'b0)
    ALU_reg2=MEM_WB[31:0];
//MEM_WB rt = ID_EX rs (Memory)
```

The bypassing units is triggered when the present operation needs the data from the register previous used but the data is not already stored there. Therefore, we should set some conditions to trigger it, otherwise it will cause more chaos. Most of the conditions are showed in the below picture, and the simplified logic is showed in the following pictures:

1a. EX/MEM. RegisterRd = ID/EX. RegisterRs
1b. EX/MEM. RegisterRd = ID/EX. RegisterRt
2a. MEM/WB. RegisterRd = ID/EX. RegisterRs
2b. MEM/WB. RegisterRd = ID/EX. RegisterRt

When the bypassing units is triggered, it will use the result from ALU directly to replace the data from register.

The second unit is stall unit. When the ALU want to use the data from the load word operation, the bypassing operation is not enough because the load word result is from the Memory Operation module instead of ALU Execution Module, and it is later. Therefore, facing such situation, we have to stall the operation by stall units.




```
if (MemRead==1'b1 && (ID_EX[124:120]==IF_ID[25:21] || ID_EX[124:120]==IF_ID[20:16]))
begin
    Regdst=1'b0;
    RegWrite=1'b0;
    ALUgate=1'b0;
    ALUSrc=1'b0;
    MemRead=1'b0;
    MemWrite=1'b0;
    MemtoReg=1'b0;
    PC_Control<=1'b0;
end
```

Stall means delay all the operation one period. Therefore, in this pipeline my method is to set all the control signals to 0, therefore, all the operation including instruction fetching will stop.

3) Details

```
else
begin
instruction={instruction_memory[PC-8],instruction_memory[PC-7],instruction_memory[PC-6],instruction_memory[PC-5]};
PC<=PC-4;
IF_ID[31:0]<=instruction;
IF_ID[63:32]<=PC;
end
```

It is about the Instruction Fetch module. Because of the clock delaying, if the stall happens, the PC should minus four to do the previous operation again. As well as the instruction, it should catch the original instruction from the instruction memory again and store it in to IF/ID. What's more, here is a problem that for the beq and bne operation, if the register is used by the previous two instructions, it may result in data and branch hazards

```
//store control signals to ID_EX
ID_EX[114]<=Regdst;
ID_EX[113]<=RegWrite;
ID_EX[112]<=ALUgate;
ID_EX[111]<=ALUSrc;
ID_EX[110]<=MemRead;
ID_EX[109]<=MemWrite;
ID_EX[108]<=MemtoReg;
```

```
//store control signals to EX_MEM
EX_MEM[36]<=ID_EX[114];
EX_MEM[35]<=ID_EX[113];
EX_MEM[34]<=ID_EX[110];
EX_MEM[33]<=ID_EX[109];
EX_MEM[32]<=ID_EX[108];
EX_MEM[53]<=ID_EX[112];
EX_MEM[52]<=ID_EX[111];
```

```
//store control signals to MEM_WB
MEM_WB[66]<=EX_MEM[36];
MEM_WB[65]<=EX_MEM[35];
MEM_WB[64]<=EX_MEM[32];
MEM_WB[78]<=EX_MEM[53];
MEM_WB[77]<=EX_MEM[52];
```

The below three pictures show how control signals is transported between the four link registers.

```
//Write Back
always @(posedge clock)
    begin
    //Memory to rd
    if (MEM_WB[66]==1'b1 && MEM_WB[65]==1'b1 && MEM_WB[64]==1'b1)
        gr[MEM_WB[71:67]]<=MEM_WB[63:32];
    //Memory to rt
    else if (MEM_WB[66]==1'b0 && MEM_WB[65]==1'b1 && MEM_WB[64]==1'b1)
        gr[MEM_WB[76:72]]<=MEM_WB[63:32];
    //ALU_result to rd
    else if (MEM_WB[66]==1'b1 && MEM_WB[65]==1'b1 && MEM_WB[64]==1'b0)
        gr[MEM_WB[71:67]]<=MEM_WB[31:0];
    //ALU_result to rt
    else if (MEM_WB[66]==1'b0 && MEM_WB[65]==1'b1 && MEM_WB[64]==1'b0)
        gr[MEM_WB[76:72]]<=MEM_WB[31:0];
    end
endmodule
```

The logic of the writeback operation is shown as the below picture.
For more details, you can check the reference in my program. I think they are very detailed and tell a lot.

Test

```
18  CPU a(.clock(clock),.start(start),.in_instruction(in_instruction));
19
20  initial begin
21  clock=0;
22  start=0;
23  in_instruction[319:288]={6'b001000,`gr0,`gr1,16'b11};
24  in_instruction[287:256]={6'b000000,`gr1,`gr1,`gr2,5'b0,6'b100000};
25  in_instruction[255:224]={6'b001000,`gr2,`gr3,16'b1};
26  in_instruction[223:192]={6'b000000,`gr2,`gr3,`gr4,5'b0,6'b100000};
27  in_instruction[191:160]={6'b000010,26'b1};
28  in_instruction[159:128]={6'b001000,`gr1,`gr5,16'b1};
29  in_instruction[127:96]={6'b001000,`gr1,`gr5,16'b11};
30  in_instruction[95:64]={6'b000000,`gr2,`gr3,`gr6,5'b0,6'b101010};
31  in_instruction[63:32]={6'b000000,`gr4,`gr6,`gr7,5'b0,6'b100011};
32  #period $display("PC=%h gr0=%h gr1=%h gr2=%h gr3=%h gr4=%h gr5=%h gr6=%h gr7=%h instruction=%b\n",a.PC,a.gr[0],a.gr[1],a.gr[2],a.gr[3],a.gr[4],a.gr[
33  #period $display("PC=%h gr0=%h gr1=%h gr2=%h gr3=%h gr4=%h gr5=%h gr6=%h gr7=%h instruction=%b\n",a.PC,a.gr[0],a.gr[1],a.gr[2],a.gr[3],a.gr[4],a.gr[
```

This is main part of my test bench. For testing, you can change content circled by the red, which is the content of instruction. I have prepared two groups of data for testing. The two groups are included in the submitted file as a txt file:

test_code.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
//Group 1
in_instruction[319:288]={6'b001000,`gr0,`gr1,16'b11};
in_instruction[287:256]={6'b000000,`gr1,`gr1,`gr2,5'b0,6'b100000};
in_instruction[255:224]={6'b001000,`gr2,`gr3,16'b1};
in_instruction[223:192]={6'b000000,`gr2,`gr3,`gr4,5'b0,6'b100000};
in_instruction[191:160]={6'b000010,26'b1};
in_instruction[159:128]={6'b001000,`gr1,`gr5,16'b1};
in_instruction[127:96]={6'b001000,`gr1,`gr5,16'b11};
in_instruction[95:64]={6'b000000,`gr2,`gr3,`gr6,5'b0,6'b101010};
in_instruction[63:32]={6'b000000,`gr4,`gr6,`gr7,5'b0,6'b100011};


//Group 2
in_instruction[319:288]={6'b101011,26'b0};
in_instruction[287:256]={6'b100011,`gr0,`gr1,16'b0};
in_instruction[255:224]={6'b001000,`gr1,`gr2,16'b11};
in_instruction[223:192]={6'b001000,`gr2,`gr3,16'b11};
in_instruction[191:160]={6'b000000,`gr2,`gr3,`gr4,5'b0,6'b100010};
in_instruction[159:128]={6'b000000,`gr1,`gr4,`gr5,5'b0,6'b100100};
in_instruction[127:96]={6'b001101,`gr4,`gr6,16'b11111};
in_instruction[95:64]={6'b001000,`gr6,`gr7,16'b111111};
```

Windows (CRLF) 第 19 行，第 67 列 100%

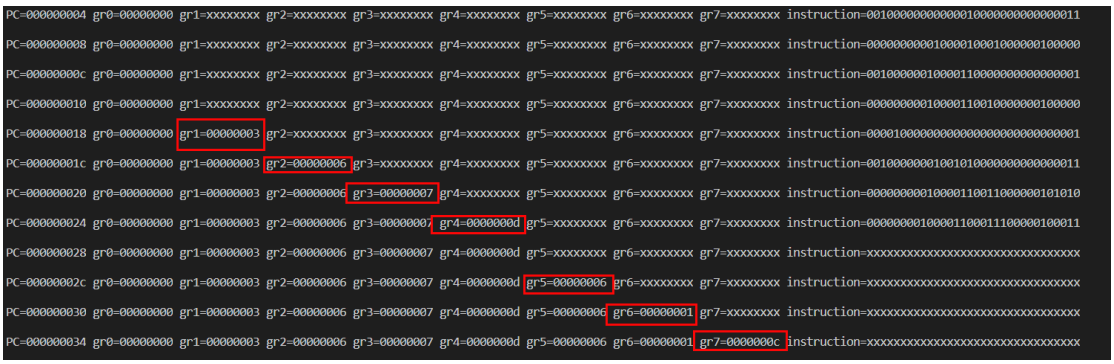
Then I'd like to explain the result of the two groups in details.

1) Group 1

- The Meaning of Each Instruction

  gr1=gr0+3
  gr2=gr1+gr1
  gr3=gr2+1
  gr4=gr2+gr3
  jump 1
  gr5=gr1+1
  gr5=gr1+3
  gr6=slt (gr2<gr3)
  gr7=gr4-gr6

- Result

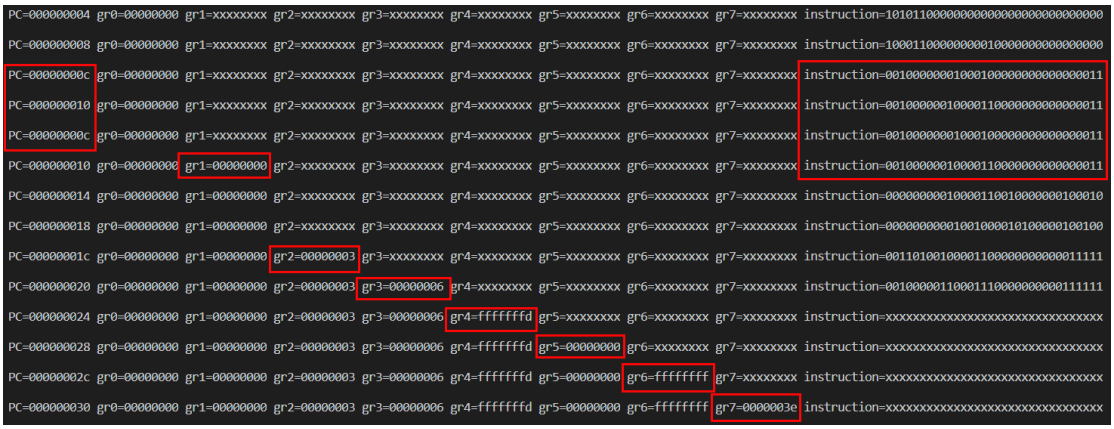


- Explanation

  Firstly, all the calculated result is correct and obey the MIPS operation. In this test, the biggest challenge is that data hazard is almost in every instruction. For example, gr2 needs to use gr1, gr3 needs to use gr2, gr4 needs to use gr3 one by one and so on. The correct result indicates that the bypassing unit is designed successfully to detect data hazards. What's more, the data is arranged in a ladder shape, which indicates the greatest feature of pipeline CPU.

2) Group 2

- The Meaning of Each Instruction
  sw gr0 to data memory [0]
  lw data memory [0] to gr1
  gr2=gr1+3
  gr3=gr2+3
  gr4=gr2-gr3(signed)
  gr5=gr4 & gr1
  gr6=gr4 | 16'b11111
  gr7=gr6+ 16'b111111
- Result



- Explanation
  The result circled by the red is all correct according to the meaning of the instructions. Comparing with group1, the challenge of group2 increases because of the load word operation. We have to do stalling to escape the data hazards. The PC in the left and the instruction in the right circled by the red indicates that the stalling operation is successful. The instruction is wait until it is prepared to get the data from gr1. Besides, almost the same as group1, almost all the operations face the data hazards. The result indicate that the stall unit and bypassing unit are successful.