

**本科实验报告**

|  |  |
| --- | --- |
| 课程名称： | 计算机体系结构 |
| 姓 名： | 蔡武威 |
| 学 院： | 计算机科学与技术学院 |
| 系： | 计算机科学与技术系 |
| 专 业： | 计算机科学与技术 |
| 学 号： | 3130101796 |
| 指导教师： | 姜晓红 |

2016 年 4月 18日

**浙江大学实验报告**

课程名称： 计算机体系结构 实验类型： 综合

实验项目名称： Lab2

学生姓名： 蔡武威 专业： 计算机科学与技术 学号： 3130101796

同组学生姓名： None 指导老师： 姜晓红

实验地点： 曹二期301 实验日期： 2016 年 4 月 18 日

1. 实验目的和要求
2. Experiment Purpose

* Understand the principles of Pipelined CPU
* Understand the basic units of Pipelined CPU
* Understand the working flow of 5-stages
* Master the method of simple Pipelined CPU
* Master the method of Pipelined CPU Stalls Detection and Stall the Pipeline
* Master methods of program verification of simple Pipelined CPU

1. Experiment Task and Requirements

* Design the CPU Controller, and the Datapath of 5-stages Pipelined CPU
* 5 Stages
* Register File
* Memory(Instruction and Data)
* Other basic units
* Implementing “stall” when have hazards
* Verify the Pipelined CPU with program and observe the execution of program

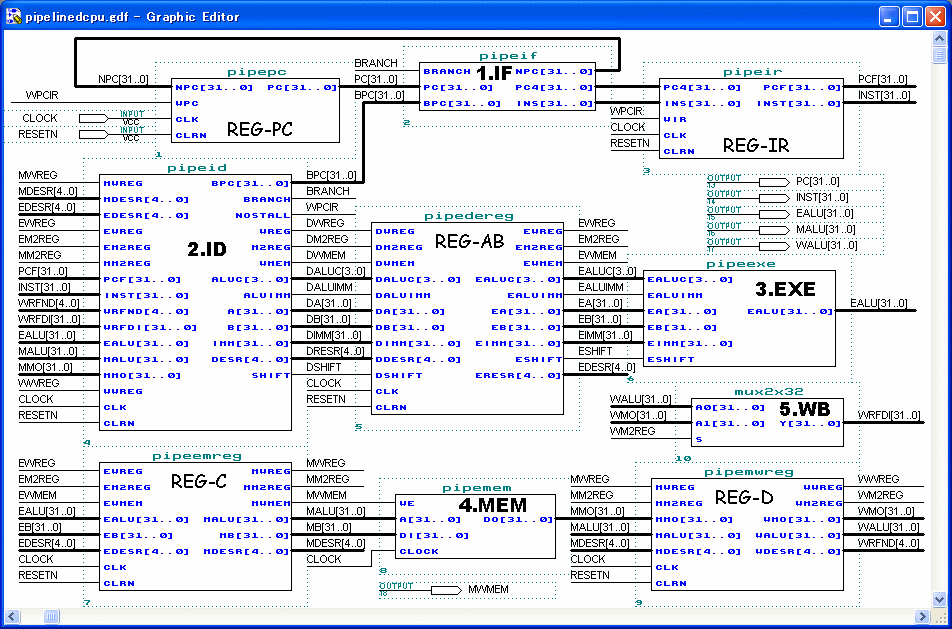
1. 实验内容和原理

* The Verilog code can be divided into the following modules:

1. mips\_top: The top module of the whole project.
2. clk\_gen: To generate different clocks.
3. anti\_jitter: To avoid the jitter of the button.
4. display: To display some information, includeing instruction address, instruction code..
5. mips: The main module to implement a MIPS CPU, which has also some sub-module, such as mips\_core, datapath, inst\_rom, data\_ram. Also, the datapath module has sub-module named regfile and alu.

* The circuit graph is shown as following:





* Control signals:

Here we define the following control signals:

1. For PC source(which is the next PC):

PC\_NEXT: no jump and branch, the next PC is just PC+4

PC\_JUMP: used when the instruction is j or jal

PC\_JR: used when the instruction is jr

PC\_BEQ: used when the instruction is beq

PC\_BNE: used when the instruction is bne

1. For ALU A source(the input a of ALU):

A\_RS: come from rs of the instruction

A\_SA: come from sa of the instruction

A\_LINK: used when the instruction is jal

A\_BRANCH: used when the instruction is beq or bne

1. For ALU B source(the input b of ALU):

B\_RT: come from rt of the instruction

B\_IMM: come from imm of the instruction

B\_LINK: used when the instruction is jal

B\_BRANCH: used when the instruction is beq or bne

1. For ALU operation: ALU\_ADD, ALU\_SUB, ALU\_SLT, ALU\_LUI, ALU\_AND, ALU\_OR, ALU\_SR
2. For WB(write back) address source

WB\_ADDR\_RD: to rd of the instruction

WB\_ADDR\_RT: to rt of the instruction

WB\_ADDR\_LINK: used when the instruction is jal

1. For WB data source

WB\_DATA\_ALU: from ALU result

WB\_ADDR\_MEM: from memory

1. Imm\_ext: Imm\_ext=1 when the immediate of the instruction should be sign-extension
2. Mem\_ren: memory read enable
3. Mem\_wen: memory write enable
4. WB\_wen:write back enable
5. Rs\_used: equals 1 when rs is used, otherwise equals 0
6. Rt\_used: equals 1 when rt is used, otherwise equals 0

The following is the table which describes the control signal of each instruction:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Instruction | add | sub | and | or | slt | srl |
| PC source | PC\_NEXT | PC\_NEXT | PC\_NEXT | PC\_NEXT | PC\_NEXT | PC\_NEXT |
| ALU A source | A\_RS | A\_RS | A\_RS | A\_RS | A\_RS | A\_RS |
| ALU B source | B\_RT | B\_RT | B\_RT | B\_RT | B\_RT | B\_RT |
| ALU operation | ALU\_ADD | ALU\_SUB | ALU\_AND | ALU\_OR | ALU\_SLT | ALU\_SR |
| WB address source | WB\_ADDR\_RD | WB\_ADDR\_RD | WB\_ADDR\_RD | WB\_ADDR\_RD | WB\_ADDR\_RD | WB\_ADDR\_RD |
| WB data source | WB\_DATA\_ALU | WB\_DATA\_ALU | WB\_DATA\_ALU | WB\_DATA\_ALU | WB\_DATA\_ALU | WB\_DATA\_ALU |
| Imm\_ext | 0 | 0 | 0 | 0 | 0 | 0 |
| Mem\_ren | 0 | 0 | 0 | 0 | 0 | 0 |
| Mem\_wen | 0 | 0 | 0 | 0 | 0 | 0 |
| WB\_wen | 1 | 1 | 1 | 1 | 1 | 1 |
| Rs\_used | 1 | 1 | 1 | 1 | 1 | 0 |
| Rt\_used | 1 | 1 | 1 | 1 | 1 | 1 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Instruction | jr | j | jal | beq | bne | addi |
| PC source | PC\_JR | PC\_J | PC\_J | PC\_BEQ | PC\_BNE | PC\_NEXT |
| ALU A source | / | / | A\_LINK | A\_BRANCH | A\_ BRANCH | A\_RS |
| ALU B source | / | / | A\_LINK | B\_ BRANCH | B\_ BRANCH | B\_IMM |
| ALU operation | / | / | ALU\_ADD | ALU\_SUB | ALU\_SUB | ALU\_ADD |
| WB address source | / | / | WB\_ADDR\_LINK | WB\_ADDR\_RD | WB\_ADDR\_RD | WB\_ADDR\_RT |
| WB data source | / | / | WB\_DATA\_ALU | WB\_DATA\_ALU | WB\_DATA\_ALU | WB\_DATA\_ALU |
| Imm\_ext | 0 | 0 | 0 | 1 | 1 | 1 |
| Mem\_ren | 0 | 0 | 0 | 0 | 0 | 0 |
| Mem\_wen | 0 | 0 | 0 | 0 | 0 | 0 |
| WB\_wen | 0 | 0 | 1 | 0 | 0 | 1 |
| Rs\_used | 1 | 0 | 0 | 1 | 1 | 1 |
| Rt\_used | 0 | 0 | 0 | 1 | 1 | 0 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Instruction | andi | ori | slti | Lw | sw | lui |
| PC source | PC\_NEXT | PC\_NEXT | PC\_NEXT | PC\_NEXT | PC\_NEXT | PC\_NEXT |
| ALU A source | A\_RS | A\_RS | A\_RS | A\_RS | A\_RS | A\_RS |
| ALU B source | B\_IMM | B\_IMM | B\_IMM | B\_IMM | B\_IMM | B\_IMM |
| ALU operation | ALU\_AND | ALU\_OR | ALU\_SLT | ALU\_ADD | ALU\_ADD | ALU\_LUI |
| WB address source | WB\_ADDR\_RT | WB\_ADDR\_RT | WB\_ADDR\_RT | WB\_ADDR\_RT | / | WB\_ADDR\_RT |
| WB data source | WB\_DATA\_ALU | WB\_DATA\_ALU | WB\_DATA\_ALU | WB\_DATA\_MEM | / | WB\_DATA\_ALU |
| Imm\_ext | 0 | 0 | 1 | 1 | 1 | 1 |
| Mem\_ren | 0 | 0 | 0 | 1 | 0 | 0 |
| Mem\_wen | 0 | 0 | 0 | 0 | 1 | 0 |
| WB\_wen | 1 | 1 | 1 | 1 | 0 | 1 |
| Rs\_used | 1 | 1 | 1 | 1 | 1 | 0 |
| Rt\_used | 0 | 0 | 0 | 0 | 1 |  |

* Key code segment

1. Controller module

module controller (/\*AUTOARG\*/

input wire clk, // main clock

input wire rst, // synchronous reset

// debug

`ifdef DEBUG

input wire debug\_en, // debug enable

input wire debug\_step, // debug step clock

`endif

// instruction decode

input wire [31:0] inst, // instruction

input wire is\_branch\_exe, // whether instruction in EXE stage is jump/branch instruction

input wire [4:0] regw\_addr\_exe, // register write address from EXE stage

input wire wb\_wen\_exe, // register write enable signal feedback from EXE stage

input wire is\_branch\_mem, // whether instruction in MEM stage is jump/branch instruction

input wire [4:0] regw\_addr\_mem, // register write address from MEM stage

input wire wb\_wen\_mem, // register write enable signal feedback from MEM stage

output reg [2:0] pc\_src, // how would PC change to next

output reg imm\_ext, // whether using sign extended to immediate data

output reg [1:0] exe\_a\_src, // data source of operand A for ALU

output reg [1:0] exe\_b\_src, // data source of operand B for ALU

output reg [3:0] exe\_alu\_oper, // ALU operation type

output reg mem\_ren, // memory read enable signal

output reg mem\_wen, // memory write enable signal

output reg [1:0] wb\_addr\_src, // address source to write data back to registers

output reg wb\_data\_src, // data source of data being written back to registers

output reg wb\_wen, // register write enable signal

output reg unrecognized, // whether current instruction can not be recognized

// pipeline control

output reg if\_rst, // stage reset signal

output reg if\_en, // stage enable signal

input wire if\_valid, // stage valid flag

output reg id\_rst,

output reg id\_en,

input wire id\_valid,

output reg exe\_rst,

output reg exe\_en,

input wire exe\_valid,

output reg mem\_rst,

output reg mem\_en,

input wire mem\_valid,

output reg wb\_rst,

output reg wb\_en,

input wire wb\_valid

);

`include "mips\_define.vh"

// instruction decode

reg rs\_used, rt\_used;

always @(\*) begin

pc\_src = PC\_NEXT;

imm\_ext = 0;

exe\_a\_src = EXE\_A\_RS;

exe\_b\_src = EXE\_B\_RT;

exe\_alu\_oper = EXE\_ALU\_ADD;

mem\_ren = 0;

mem\_wen = 0;

wb\_addr\_src = WB\_ADDR\_RD;

wb\_data\_src = WB\_DATA\_ALU;

wb\_wen = 0;

rs\_used = 0;

rt\_used = 0;

unrecognized = 0;

case (inst[31:26])

INST\_R: begin

case (inst[5:0])

R\_FUNC\_JR: begin

pc\_src = PC\_JR;

rs\_used = 1;

end

R\_FUNC\_ADD: begin

exe\_alu\_oper = EXE\_ALU\_ADD;

wb\_addr\_src = WB\_ADDR\_RD;

wb\_data\_src = WB\_DATA\_ALU;

wb\_wen = 1;

rs\_used = 1;

rt\_used = 1;

end

R\_FUNC\_SUB: begin

exe\_alu\_oper = EXE\_ALU\_SUB;

wb\_addr\_src = WB\_ADDR\_RD;

wb\_data\_src = WB\_DATA\_ALU;

wb\_wen = 1;

rs\_used = 1;

rt\_used = 1;

end

R\_FUNC\_AND: begin

exe\_alu\_oper = EXE\_ALU\_AND;

wb\_addr\_src = WB\_ADDR\_RD;

wb\_data\_src = WB\_DATA\_ALU;

wb\_wen = 1;

rs\_used = 1;

rt\_used = 1;

end

R\_FUNC\_OR: begin

exe\_alu\_oper = EXE\_ALU\_OR;

wb\_addr\_src = WB\_ADDR\_RD;

wb\_data\_src = WB\_DATA\_ALU;

wb\_wen = 1;

rs\_used = 1;

rt\_used = 1;

end

R\_FUNC\_SLT: begin

exe\_alu\_oper = EXE\_ALU\_SLT;

wb\_addr\_src = WB\_ADDR\_RD;

wb\_data\_src = WB\_DATA\_ALU;

wb\_wen = 1;

rs\_used = 1;

rt\_used = 1;

end

default: begin

unrecognized = 1;

end

endcase

end

INST\_J: begin

pc\_src = PC\_JUMP;

end

INST\_JAL: begin

pc\_src = PC\_JUMP;

exe\_a\_src = EXE\_A\_LINK;

exe\_b\_src = EXE\_B\_LINK;

exe\_alu\_oper = EXE\_ALU\_ADD;

wb\_addr\_src = WB\_ADDR\_LINK;

wb\_data\_src = WB\_DATA\_ALU;

wb\_wen = 1;

end

INST\_BEQ: begin

pc\_src = PC\_BEQ;

exe\_a\_src = EXE\_A\_BRANCH;

exe\_b\_src = EXE\_A\_BRANCH;

exe\_alu\_oper = EXE\_ALU\_ADD;

imm\_ext = 1;

rs\_used = 1;

rt\_used = 1;

end

INST\_BNE: begin

pc\_src = PC\_BNE;

exe\_a\_src = EXE\_A\_BRANCH;

exe\_b\_src = EXE\_A\_BRANCH;

exe\_alu\_oper = EXE\_ALU\_ADD;

imm\_ext = 1;

rs\_used = 1;

rt\_used = 1;

end

INST\_ADDI: begin

imm\_ext = 1;

exe\_b\_src = EXE\_B\_IMM;

exe\_alu\_oper = EXE\_ALU\_ADD;

wb\_addr\_src = WB\_ADDR\_RT;

wb\_data\_src = WB\_DATA\_ALU;

wb\_wen = 1;

rs\_used = 1;

end

INST\_ANDI: begin

imm\_ext = 0;

exe\_b\_src = EXE\_B\_IMM;

exe\_alu\_oper = EXE\_ALU\_AND;

wb\_addr\_src = WB\_ADDR\_RT;

wb\_data\_src = WB\_DATA\_ALU;

wb\_wen = 1;

rs\_used = 1;

end

INST\_ORI: begin

imm\_ext = 0;

exe\_b\_src = EXE\_B\_IMM;

exe\_alu\_oper = EXE\_ALU\_OR;

wb\_addr\_src = WB\_ADDR\_RT;

wb\_data\_src = WB\_DATA\_ALU;

wb\_wen = 1;

rs\_used = 1;

end

INST\_LW: begin

imm\_ext = 1;

exe\_a\_src = EXE\_A\_RS;

exe\_b\_src = EXE\_B\_IMM;

exe\_alu\_oper = EXE\_ALU\_ADD;

mem\_ren = 1;

wb\_addr\_src = WB\_ADDR\_RT;

wb\_data\_src = WB\_DATA\_MEM;

wb\_wen = 1;

rs\_used = 1;

end

INST\_SW: begin

imm\_ext = 1;

exe\_a\_src = EXE\_A\_RS;

exe\_b\_src = EXE\_B\_IMM;

exe\_alu\_oper = EXE\_ALU\_ADD;

mem\_wen = 1;

rs\_used = 1;

rt\_used = 1;

end

default: begin

unrecognized = 1;

end

endcase

end

// pipeline control

reg reg\_stall;

reg branch\_stall;

wire [4:0] addr\_rs, addr\_rt;

assign

addr\_rs = inst[25:21],

addr\_rt = inst[20:16];

always @(\*) begin

reg\_stall = 0;

if (rs\_used && addr\_rs != 0) begin

if (regw\_addr\_exe == addr\_rs && wb\_wen\_exe) begin

reg\_stall = 1;

end

else if (regw\_addr\_mem == addr\_rs && wb\_wen\_mem) begin

reg\_stall = 1;

end

end

if (rt\_used && addr\_rt != 0) begin

if (regw\_addr\_exe == addr\_rt && wb\_wen\_exe) begin

reg\_stall = 1;

end

else if (regw\_addr\_mem == addr\_rt && wb\_wen\_mem) begin

reg\_stall = 1;

end

end

end

always @(\*) begin

branch\_stall = 0;

if (pc\_src != PC\_NEXT || is\_branch\_exe || is\_branch\_mem)

branch\_stall = 1;

end

`ifdef DEBUG

reg debug\_step\_prev;

always @(posedge clk) begin

debug\_step\_prev <= debug\_step;

end

`endif

always @(\*) begin

if\_rst = 0;

if\_en = 1;

id\_rst = 0;

id\_en = 1;

exe\_rst = 0;

exe\_en = 1;

mem\_rst = 0;

mem\_en = 1;

wb\_rst = 0;

wb\_en = 1;

if (rst) begin

if\_rst = 1;

id\_rst = 1;

exe\_rst = 1;

mem\_rst = 1;

wb\_rst = 1;

end

`ifdef DEBUG

// suspend and step execution

else if ((debug\_en) && ~(~debug\_step\_prev && debug\_step)) begin

if\_en = 0;

id\_en = 0;

exe\_en = 0;

mem\_en = 0;

wb\_en = 0;

end

`endif

// this stall indicate that ID is waiting for previous instruction, should insert NOPs between ID and EXE.

else if (reg\_stall) begin

if\_en = 0;

id\_en = 0;

exe\_rst = 1;

end

// this stall indicate that a jump/branch instruction is running, so that 3 NOP should be inserted between IF and ID

else if (branch\_stall) begin

id\_rst = 1;

end

end

endmodule

1. Datapath module

module datapath (

input wire clk, // main clock

// debug

`ifdef DEBUG

input wire [5:0] debug\_addr, // debug address

output wire [31:0] debug\_data, // debug data

`endif

// control signals

output reg [31:0] inst\_data\_id, // instruction

output reg is\_branch\_exe, // whether instruction in EXE stage is jump/branch instruction

output reg [4:0] regw\_addr\_exe, // register write address from EXE stage

output reg wb\_wen\_exe, // register write enable signal feedback from EXE stage

output reg is\_branch\_mem, // whether instruction in MEM stage is jump/branch instruction

output reg [4:0] regw\_addr\_mem, // register write address from MEM stage

output reg wb\_wen\_mem, // register write enable signal feedback from MEM stage

input wire [2:0] pc\_src\_ctrl, // how would PC change to next

input wire imm\_ext\_ctrl, // whether using sign extended to immediate data

input wire [1:0] exe\_a\_src\_ctrl, // data source of operand A for ALU

input wire [1:0] exe\_b\_src\_ctrl, // data source of operand B for ALU

input wire [3:0] exe\_alu\_oper\_ctrl, // ALU operation type

input wire mem\_ren\_ctrl, // memory read enable signal

input wire mem\_wen\_ctrl, // memory write enable signal

input wire [1:0] wb\_addr\_src\_ctrl, // address source to write data back to registers

input wire wb\_data\_src\_ctrl, // data source of data being written back to registers

input wire wb\_wen\_ctrl, // register write enable signal

// IF signals

input wire if\_rst, // stage reset signal

input wire if\_en, // stage enable signal

output reg if\_valid, // working flag

output reg inst\_ren, // instruction read enable signal

output reg [31:0] inst\_addr, // address of instruction needed

input wire [31:0] inst\_data, // instruction fetched

// ID signals

input wire id\_rst,

input wire id\_en,

output reg id\_valid,

// EXE signals

input wire exe\_rst,

input wire exe\_en,

output reg exe\_valid,

// MEM signals

input wire mem\_rst,

input wire mem\_en,

output reg mem\_valid,

output wire mem\_ren, // memory read enable signal

output wire mem\_wen, // memory write enable signal

output wire [31:0] mem\_addr, // address of memory

output wire [31:0] mem\_dout, // data writing to memory

input wire [31:0] mem\_din, // data read from memory

// WB signals

input wire wb\_rst,

input wire wb\_en,

output reg wb\_valid

);

`include "mips\_define.vh"

// control signals

reg [2:0] pc\_src\_exe, pc\_src\_mem;

reg [1:0] exe\_a\_src\_exe, exe\_b\_src\_exe;

reg [3:0] exe\_alu\_oper\_exe;

reg mem\_ren\_exe, mem\_ren\_mem;

reg mem\_wen\_exe, mem\_wen\_mem;

reg wb\_data\_src\_exe, wb\_data\_src\_mem, wb\_data\_src\_wb;

// IF signals

wire [31:0] inst\_addr\_next;

// ID signals

reg [31:0] inst\_addr\_id;

reg [31:0] inst\_addr\_next\_id;

reg [4:0] regw\_addr\_id;

wire [4:0] addr\_rs, addr\_rt, addr\_rd;

wire [31:0] data\_rs, data\_rt, data\_imm;

// EXE signals

reg [31:0] inst\_addr\_exe;

reg [31:0] inst\_addr\_next\_exe;

reg [31:0] inst\_data\_exe;

reg [31:0] data\_rs\_exe, data\_rt\_exe, data\_imm\_exe;

reg [31:0] opa\_exe, opb\_exe;

wire [31:0] alu\_out\_exe;

wire rs\_rt\_equal\_exe;

// MEM signals

reg [31:0] inst\_addr\_mem;

reg [31:0] inst\_addr\_next\_mem;

reg [31:0] inst\_data\_mem;

reg [4:0] data\_rs\_mem;

reg [31:0] data\_rt\_mem;

reg [31:0] alu\_out\_mem;

reg [31:0] branch\_target\_mem;

reg rs\_rt\_equal\_mem;

// WB signals

reg wb\_wen\_wb;

reg [31:0] alu\_out\_wb;

reg [31:0] mem\_din\_wb;

reg [4:0] regw\_addr\_wb;

reg [31:0] regw\_data\_wb;

initial begin

pc\_src\_exe = PC\_NEXT;

pc\_src\_mem = PC\_NEXT;

mem\_ren\_exe = 0;

mem\_ren\_mem = 0;

mem\_wen\_exe = 0;

mem\_wen\_mem = 0;

is\_branch\_exe = 0; // whether instruction in EXE stage is jump/branch instruction

is\_branch\_mem = 0; // whether instruction in MEM stage is jump/branch instruction

wb\_wen\_exe = 0; // register write enable signal feedback from EXE stage

wb\_wen\_mem = 0; // register write enable signal feedback from MEM stage

wb\_wen\_wb = 0;

end

// debug

`ifdef DEBUG

wire [31:0] debug\_data\_reg;

reg [31:0] debug\_data\_signal;

always @(posedge clk) begin

case (debug\_addr[4:0])

0: debug\_data\_signal <= inst\_addr;

1: debug\_data\_signal <= inst\_data;

2: debug\_data\_signal <= inst\_addr\_id;

3: debug\_data\_signal <= inst\_data\_id;

4: debug\_data\_signal <= inst\_addr\_exe;

5: debug\_data\_signal <= inst\_data\_exe;

6: debug\_data\_signal <= inst\_addr\_mem;

7: debug\_data\_signal <= inst\_data\_mem;

8: debug\_data\_signal <= {27'b0, addr\_rs};

9: debug\_data\_signal <= data\_rs;

10: debug\_data\_signal <= {27'b0, addr\_rt};

11: debug\_data\_signal <= data\_rt;

12: debug\_data\_signal <= data\_imm;

13: debug\_data\_signal <= opa\_exe;

14: debug\_data\_signal <= opb\_exe;

15: debug\_data\_signal <= alu\_out\_exe;

16: debug\_data\_signal <= 0;

17: debug\_data\_signal <= 0;

18: debug\_data\_signal <= {19'b0, inst\_ren, 7'b0, mem\_ren, 3'b0, mem\_wen};

19: debug\_data\_signal <= mem\_addr;

20: debug\_data\_signal <= mem\_din;

21: debug\_data\_signal <= mem\_dout;

22: debug\_data\_signal <= {27'b0, regw\_addr\_wb};

23: debug\_data\_signal <= regw\_data\_wb;

default: debug\_data\_signal <= 32'hFFFF\_FFFF;

endcase

end

assign

debug\_data = debug\_addr[5] ? debug\_data\_signal : debug\_data\_reg;

`endif

// IF stage

assign

inst\_addr\_next = inst\_addr + 4;

always @(\*) begin

if\_valid = ~if\_rst & if\_en;

inst\_ren = ~if\_rst;

end

always @(posedge clk) begin

if (if\_rst) begin

inst\_addr <= 0;

end

else if (if\_en) begin

if (is\_branch\_mem)

inst\_addr <= branch\_target\_mem;

else

inst\_addr <= inst\_addr\_next;

end

end

// IF/ID

always @(posedge clk) begin

if (id\_rst) begin

id\_valid <= 0;

inst\_addr\_id <= 0;

inst\_data\_id <= 0;

inst\_addr\_next\_id <= 0;

end

else if (id\_en) begin

id\_valid <= if\_valid;

inst\_addr\_id <= inst\_addr;

inst\_data\_id <= inst\_data;

inst\_addr\_next\_id <= inst\_addr\_next;

end

end

// ID Stage

assign

addr\_rs = inst\_data\_id[25:21],

addr\_rt = inst\_data\_id[20:16],

addr\_rd = inst\_data\_id[15:11],

data\_imm = imm\_ext\_ctrl ? {{16{inst\_data\_id[15]}}, inst\_data\_id[15:0]} : {16'b0, inst\_data\_id[15:0]};

always @(\*) begin

regw\_addr\_id = inst\_data\_id[15:11];

case (wb\_addr\_src\_ctrl)

WB\_ADDR\_RD: regw\_addr\_id = addr\_rd;

WB\_ADDR\_RT: regw\_addr\_id = addr\_rt;

WB\_ADDR\_LINK: regw\_addr\_id = 32'h31;

endcase

end

regfile REGFILE (

.clk(clk),

`ifdef DEBUG

.debug\_addr(debug\_addr[4:0]),

.debug\_data(debug\_data\_reg),

`endif

.addr\_a(addr\_rs),

.data\_a(data\_rs),

.addr\_b(addr\_rt),

.data\_b(data\_rt),

.en\_w(wb\_wen\_wb),

.addr\_w(regw\_addr\_wb),

.data\_w(regw\_data\_wb)

);

// ID/EXE

always @(posedge clk) begin

if (exe\_rst) begin

exe\_valid <= 0;

inst\_addr\_exe <= 0;

inst\_data\_exe <= 0;

inst\_addr\_next\_exe <= 0;

regw\_addr\_exe <= 0;

pc\_src\_exe <= 0;

exe\_a\_src\_exe <= 0;

exe\_b\_src\_exe <= 0;

data\_rs\_exe <= 0;

data\_rt\_exe <= 0;

data\_imm\_exe <= 0;

exe\_alu\_oper\_exe <= 0;

mem\_ren\_exe <= 0;

mem\_wen\_exe <= 0;

wb\_data\_src\_exe <= 0;

wb\_wen\_exe <= 0;

end

else if (exe\_en) begin

exe\_valid <= id\_valid;

inst\_addr\_exe <= inst\_addr\_id;

inst\_data\_exe <= inst\_data\_id;

inst\_addr\_next\_exe <= inst\_addr\_next\_id;

regw\_addr\_exe <= regw\_addr\_id;

pc\_src\_exe <= pc\_src\_ctrl;

exe\_a\_src\_exe <= exe\_a\_src\_ctrl;

exe\_b\_src\_exe <= exe\_b\_src\_ctrl;

data\_rs\_exe <= data\_rs;

data\_rt\_exe <= data\_rt;

data\_imm\_exe <= data\_imm;

exe\_alu\_oper\_exe <= exe\_alu\_oper\_ctrl;

mem\_ren\_exe <= mem\_ren\_ctrl;

mem\_wen\_exe <= mem\_wen\_ctrl;

wb\_data\_src\_exe <= wb\_data\_src\_ctrl;

wb\_wen\_exe <= wb\_wen\_ctrl;

end

end

// EXE Stage

always @(\*) begin

is\_branch\_exe <= (pc\_src\_exe != PC\_NEXT);

end

assign

rs\_rt\_equal\_exe = (data\_rs\_exe == data\_rt\_exe);

always @(\*) begin

opa\_exe = data\_rs\_exe;

opb\_exe = data\_rt\_exe;

case (exe\_a\_src\_exe)

EXE\_A\_RS: opa\_exe = data\_rs\_exe;

EXE\_A\_LINK: opa\_exe = 0;

EXE\_A\_BRANCH: opa\_exe = inst\_addr\_next\_exe;

endcase

case (exe\_b\_src\_exe)

EXE\_B\_RT: opb\_exe = data\_rt\_exe;

EXE\_B\_IMM: opb\_exe = data\_imm\_exe;

EXE\_B\_LINK: opb\_exe = inst\_addr\_next\_exe; // linked address is the next one of current instruction

EXE\_B\_BRANCH: opb\_exe = {data\_imm\_exe[29:0],2'b00};

endcase

end

alu ALU (

.a(opa\_exe),

.b(opb\_exe),

.oper(exe\_alu\_oper\_exe),

.result(alu\_out\_exe)

);

// EX/MEM

always @(posedge clk) begin

if (mem\_rst) begin

mem\_valid <= 0;

pc\_src\_mem <= 0;

inst\_addr\_mem <= 0;

inst\_data\_mem <= 0;

inst\_addr\_next\_mem <= 0;

regw\_addr\_mem <= 0;

data\_rs\_mem <= 0;

data\_rt\_mem <= 0;

alu\_out\_mem <= 0;

mem\_ren\_mem <= 0;

mem\_wen\_mem <= 0;

wb\_data\_src\_mem <= 0;

wb\_wen\_mem <= 0;

rs\_rt\_equal\_mem <= 0;

end

else if (mem\_en) begin

mem\_valid <= exe\_valid;

pc\_src\_mem <= pc\_src\_exe;

inst\_addr\_mem <= inst\_addr\_exe;

inst\_data\_mem <= inst\_data\_exe;

inst\_addr\_next\_mem <= inst\_addr\_next\_exe;

regw\_addr\_mem <= regw\_addr\_exe;

data\_rs\_mem <= data\_rs\_exe;

data\_rt\_mem <= data\_rt\_exe;

alu\_out\_mem <= alu\_out\_exe;

mem\_ren\_mem <= mem\_ren\_exe;

mem\_wen\_mem <= mem\_wen\_exe;

wb\_data\_src\_mem <= wb\_data\_src\_exe;

wb\_wen\_mem <= wb\_wen\_exe;

rs\_rt\_equal\_mem <= rs\_rt\_equal\_exe;

end

end

// MEM State

always @(\*) begin

is\_branch\_mem <= (pc\_src\_mem != PC\_NEXT);

end

always @(\*) begin

case (pc\_src\_mem)

PC\_JUMP: branch\_target\_mem <= {inst\_addr\_exe[31:28],inst\_data\_mem[25:0],2'b00};

PC\_JR: branch\_target\_mem <= data\_rs\_mem;

PC\_BEQ: branch\_target\_mem <= rs\_rt\_equal\_mem? alu\_out\_mem : inst\_addr\_next\_mem;

PC\_BNE: branch\_target\_mem <= rs\_rt\_equal\_mem? inst\_addr\_next\_mem : alu\_out\_mem;

default: branch\_target\_mem <= inst\_addr\_next\_mem; // will never used

endcase

end

assign

mem\_ren = mem\_ren\_mem,

mem\_wen = mem\_wen\_mem,

mem\_addr = alu\_out\_mem,

mem\_dout = data\_rt\_mem;

// WB/MEM

always @(posedge clk) begin

if (wb\_rst) begin

wb\_valid <= 0;

wb\_wen\_wb <= 0;

wb\_data\_src\_wb <= 0;

regw\_addr\_wb <= 0;

alu\_out\_wb <= 0;

mem\_din\_wb <= 0;

end

else if (wb\_en) begin

wb\_valid <= mem\_valid;

wb\_wen\_wb <= wb\_wen\_mem;

wb\_data\_src\_wb <= wb\_data\_src\_mem;

regw\_addr\_wb <= regw\_addr\_mem;

alu\_out\_wb <= alu\_out\_mem;

mem\_din\_wb <= mem\_din;

end

end

//WB Stage

always @(\*) begin

regw\_data\_wb = alu\_out\_wb;

case (wb\_data\_src\_wb)

WB\_DATA\_ALU: regw\_data\_wb = alu\_out\_wb;

WB\_DATA\_MEM: regw\_data\_wb = mem\_din\_wb;

endcase

end

endmodule

1. 实验过程和数据记录及结果分析
2. Input data sets and output results
   * + Instruction memory

|  |  |  |  |
| --- | --- | --- | --- |
| Line number | Assembly code | Machine code | result |
| 0 | Lw r1, 20(r0) | 8c010014, | R1=4 |
| 1 | Lw r2, 24(r0) | 8c020018 | R2=1 |
| 2 | Add r3,r2,r1 | 00221820 | R3=5 //LW-ALU:forwarding:1 stall |
| 3 | Sub r4,r3,r1 | 00612022 | R4=1 //ALU-ALU |
| 4 | And r5,r3,r1 | 00232824 | R5=4 //no hazards |
| 5 | Or r6,r3,r1 | 00223025 | R6=5 //no hazards |
| 6 | addi r6,r3,4 | 20660004 | r6=9 //no hazards |
| 7 | Add r7, r0, r1 | 00203820 | R7=4 //no hazards |
| 8 | Lw r8,0(r7) | 8ce80000 | R8=8 //ALU-LW |
| 9 | Sw r8,8(r7) | Ace80008 | LW-SW：forwarding to solve |
| 10 | Lw r9, 8(r7) | 8ce90008 | R9=8 |
| 11 | Sw r7, 0(r9) | ad270000 | LW-SW: forwarding:1 stall |
| 12 | Lw r10,0(r9) | 8d2a0000 | r10=4 |
| 13 | Add r10, r1, r1 | 00215020 | R10=8 |
| 14 | Add r11, r2,r2 | 00425820 | R11=2 |
| 15 | Add r10,r1,r2 | 00415020 | R10=5 |
| 16 | Beq r10, r11 ,8 | 10220008 | not taken//ALU-BEQ; branch |
| 17 | Lw r1, 8(r7) | 8ce10008 | R1=8 |
| 18 | Lw r2, 24(r0) | 8c020018 | R2=1 |
| 19 | Add r3,r2,r1Lw r2, 24(r0) | 00221820 | R3=9 //LW-ALU |
| 20 | Sub r4,r3,r1 | 00612022 | R4=1 |
| 21 | Addi r20, r4, 1 | 20940001 | R20=2 //ALU-addi |
| 22 | Ori r20, r4, 1 | 34940001 | R20=1 //no hazards |
| 23 | Bne r1, r2, 4 | 14220004 | taken |
| 24 | Lw r1, (20)r0 | 8c010014 | Not execute |
| 25 | Lw r2, (24)r0 | 8c020018 | R2=1 |
| 26 | Add r3,r2,r1 | 00221820 | R3=9 |
| 27 | Sub r4,r3,r1 | 00412022 | R4=1 |
| 28 | J 0 | 08000000 | j |

* + - Data memory

00000000

00000008

00000000

00000000

00000000

00000004

00000001

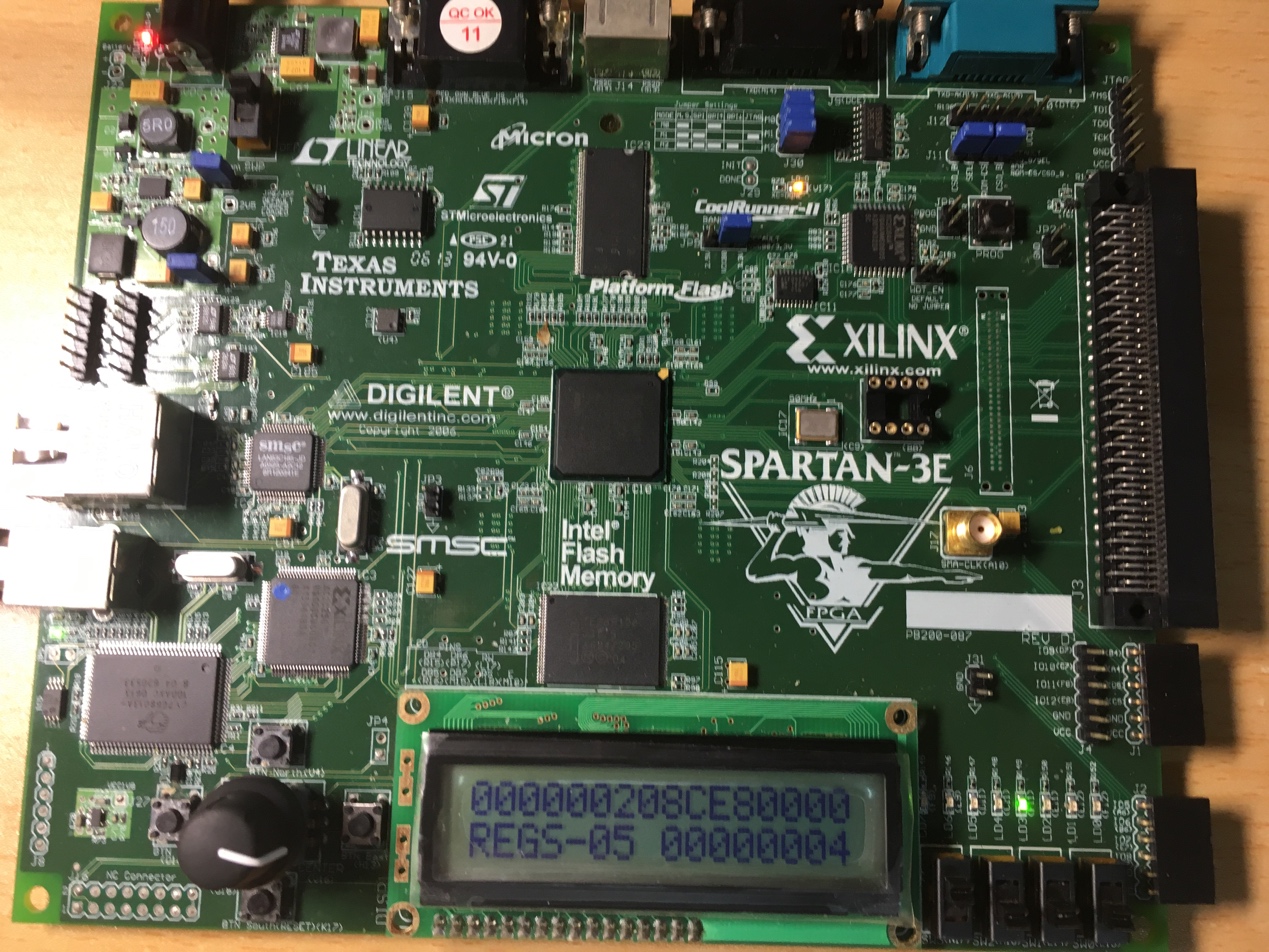
1. Screen shots and pictures of the running result on Spartan3E
   * + The left 8 digits of the top is PC and the right 8 digits is the code of the instruction which is in the IF stage. The bottom is some information. In this case, it shows the data in register 0($zero). The following picture is taken when starting.



* + - The following is taken when the 4th instruction ”Sub r4, r4, r1” is taken, which will arise a ALU-ALU hazard.



* + - The following is taken when “lw r8,0(r7)” is taken, which will arise a ALU-LW hazard.



* + - The following pictures are some others taken when execute the testing code.







1. Steps to do this experiment
   * + Review and expand the previous code in the lab1.
     + Modify the display module to make it suitable for pipelined CPU debugging
     + Modify the MIPS module to implement pipelined CPU, which is the most important step.
     + Get the testing code of MIPS
     + Compile and download the bit file to the board
     + Execute the testing code and debug
2. 讨论与心得

After “lab 1 warming up”, I feel it may be easy to complete the later labs, which is to implement pipelined CPU. However, when practically coding pipelined CPU, I begin to feel embarrassed that it is not easy not all with so many bugs existing. To implement the pipelined CPU well, I review the textbook and courseware and then do the coding, simulating and debugging. The main problem that disturbed me is that there is always some bug existing in IF and ID stages. Delightedly, with much effort, we succeed to solve the problem and implement the pipelined CPU at last. Thanks to this lab, I has opportunity to strengthen my knowledge about pipelined CPU and to practically implement it, which is a process with sweet and harvest.