# 实验二 单周期CPU设计

1. 实验要求
2. 实验过程
   1. R型指令
      1. 指令分析，数据通路图。

指令：

31~26位：op

25~21位：rs

20~16位：rt

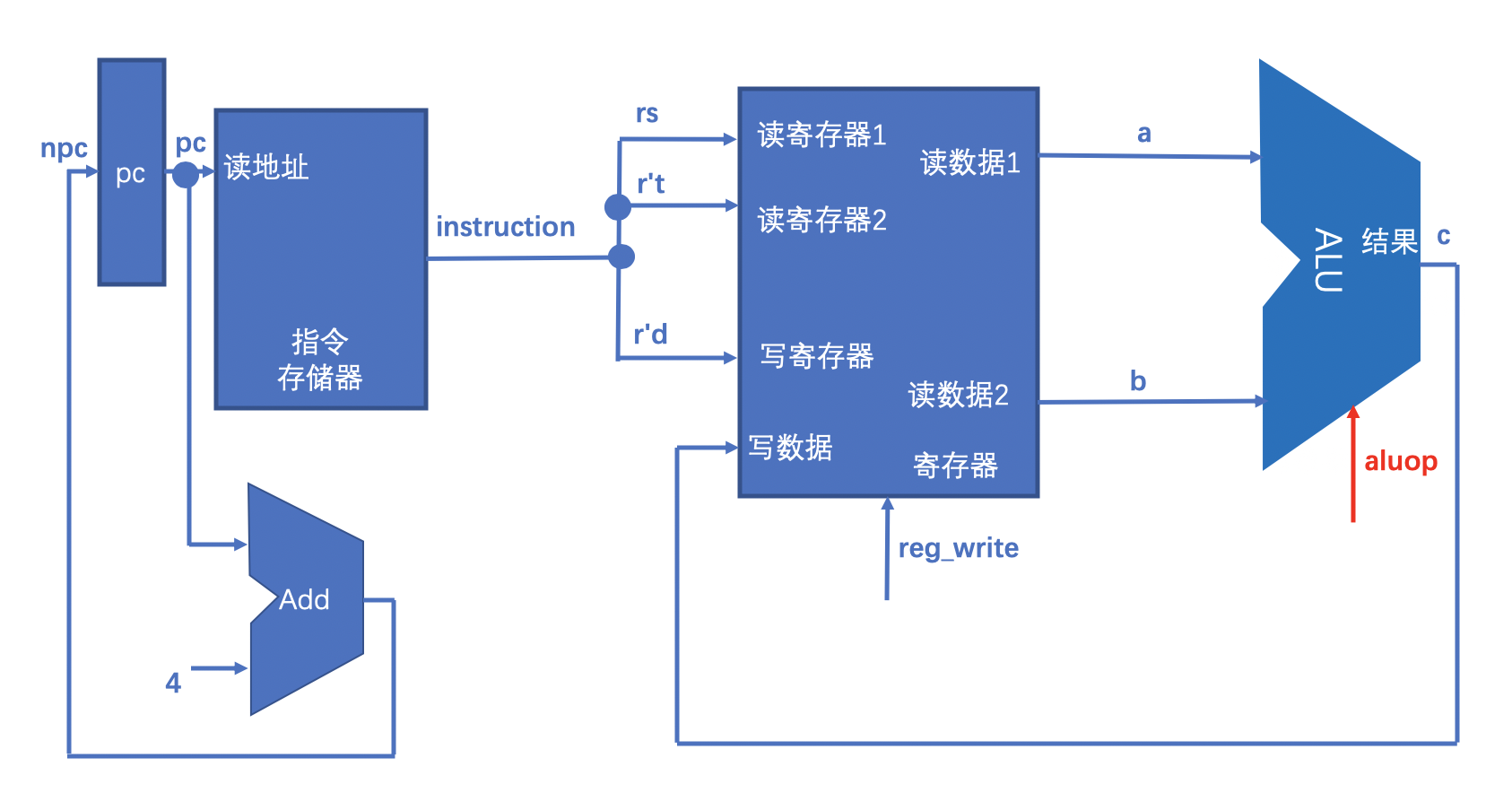
15~11位：rd

10~6位：shamt

5~0位：function

操作：GPR[rd]←GPR[rs] +（或其它操作） GPR[rt]

数据通路图：



* + 1. 实现过程（主要部分）

1. 代码（加必要的注释），仿真截图，遇到的问题和解决办法。

alu.v

`include "header.v"

module alu(c,a,b,aluop);

output reg signed [31:0] c;

input signed [31:0] a;

input signed [31:0] b;

input [2:0] aluop;

always @(a or b or aluop)

case (aluop)

`ADD: c = a + b; //加

`SUB: c = a - b; //减

`AND: c = a & b; //与

`OR: c = a | b; //或

`SLT: c = (a < b) ? 32'd1 : 32'd0; //比较大小

default: c = a + b;

endcase

endmodule

ctrl.v

`include "header.v"

module ctrl(reg\_write,aluop,op,funct);

output reg\_write;

output [2:0] aluop;

input [5:0] op;

input [5:0] funct;

assign reg\_write = (op == `op\_R)? 1 : 0; //现在reg\_write都是1

assign aluop = (op == `op\_R) ? (funct == `funct\_addu) ? `ADD:

(funct == `funct\_subu) ? `SUB:

(funct == `funct\_add) ? `ADD:

(funct == `funct\_and) ? `AND:

(funct == `funct\_or) ? `OR:

(funct == `funct\_slt) ? `SLT:

`ADD:

`ADD;

//op都是R型指令，根据funct字段决定进行何种操作

endmodule

gpr.v

module gpr(a,b,clock,reg\_write,num\_write,rs,rt,data\_write);

output [31:0] a;

output [31:0] b;

input clock;

input reg\_write;

input [4:0] rs; //读寄存器1

input [4:0] rt; //读寄存器2

input [4:0] num\_write; //写寄存器

input [31:0] data\_write; //写数据

reg [31:0] gp\_registers[31:0]; //32个寄存器

assign a = rs ? gp\_registers[rs] : 0;

assign b = rt ? gp\_registers[rt] : 0;

always @(posedge clock)

begin

if (reg\_write) //有这个标识时候写入数据

gp\_registers[num\_write] <= data\_write;

end

endmodule

header.v //一个新增的头文件，用来保存op和funct字段的值和传入alu单元的操作

//aluop

`define ADD 3'b000

`define SUB 3'b001

`define AND 3'b010

`define OR 3'b011

`define SLT 3'b100

`define LUI 3'b101

//funct (when op=0)

`define funct\_addu 6'b100001

`define funct\_subu 6'b100011

`define funct\_add 6'b100000

`define funct\_and 6'b100100

`define funct\_or 6'b100101

`define funct\_slt 6'b101010

`define funct\_jr 6'b001000

//op

`define op\_R 6'b000000

im.v

module im(instruction,pc);

output [31:0] instruction;

input [31:0] pc;

reg [31:0] ins\_memory[1023:0]; //4k指令存储器

assign instruction = ins\_memory[pc[11:0]>>2];

endmodule

pc.v //和之前一样

module pc(pc,clock,reset,npc);

output [31:0] pc;

input clock;

input reset;

input [31:0] npc;

reg [31:0] data;

always @(posedge clock or negedge reset)

begin

if (reset == 0)

data <= 32'h00003000;

else

data <= npc;

end

assign pc = data;

endmodule

s\_cycle\_cpu.v

module s\_cycle\_cpu(clock,reset);

//输入

input clock;

input reset;

wire [31:0] npc;

wire [31:0] pc;

wire [31:0] instruction;

wire [4:0] rs; //读寄存器1

wire [4:0] rt; //读寄存器2

wire [4:0] rd;

wire reg\_write;

wire [31:0] a;

wire [31:0] b;

wire [31:0] c;

wire [2:0] aluop;

wire [5:0] op;

wire [5:0] funct;

pc PC(.pc(pc),.clock(clock),.reset(reset),.npc(npc));

assign npc = pc + 4;

im IM(.instruction(instruction),.pc(pc));

assign op = instruction [31:26];

assign rs = instruction [25:21];

assign rt = instruction [20:16];

assign rd = instruction [15:11];

assign funct = instruction [5:0];

ctrl CTRL(.reg\_write(reg\_write),.aluop(aluop),.op(op),.funct(funct));

gpr GPR(.a(a),.b(b),.clock(clock),.reg\_write(reg\_write),.num\_write(rd),.rs(rs),.rt(rt),.data\_write(c));

alu ALU(.c(c),.a(a),.b(b),.aluop(aluop));

endmodule

仿真截图统一写最下面。这里只放代码和分析。

上节课gpr写的不对，按照之前的代码无法通过，取消了赋初值，改变了部分语句的写法后正确运行通过。

1. 和老师讲解代码的区别，你的思考，疑问等。

自己写的gpr不对，按照老师讲解的代码做的，所以区别不大。

思考过s\_cycle\_cpu.v里语句的顺序会不会影响结果，先前以为必须按电流流过的顺序连线，后发现与顺序无关，而是会在每一个时钟上升沿做所有操作。

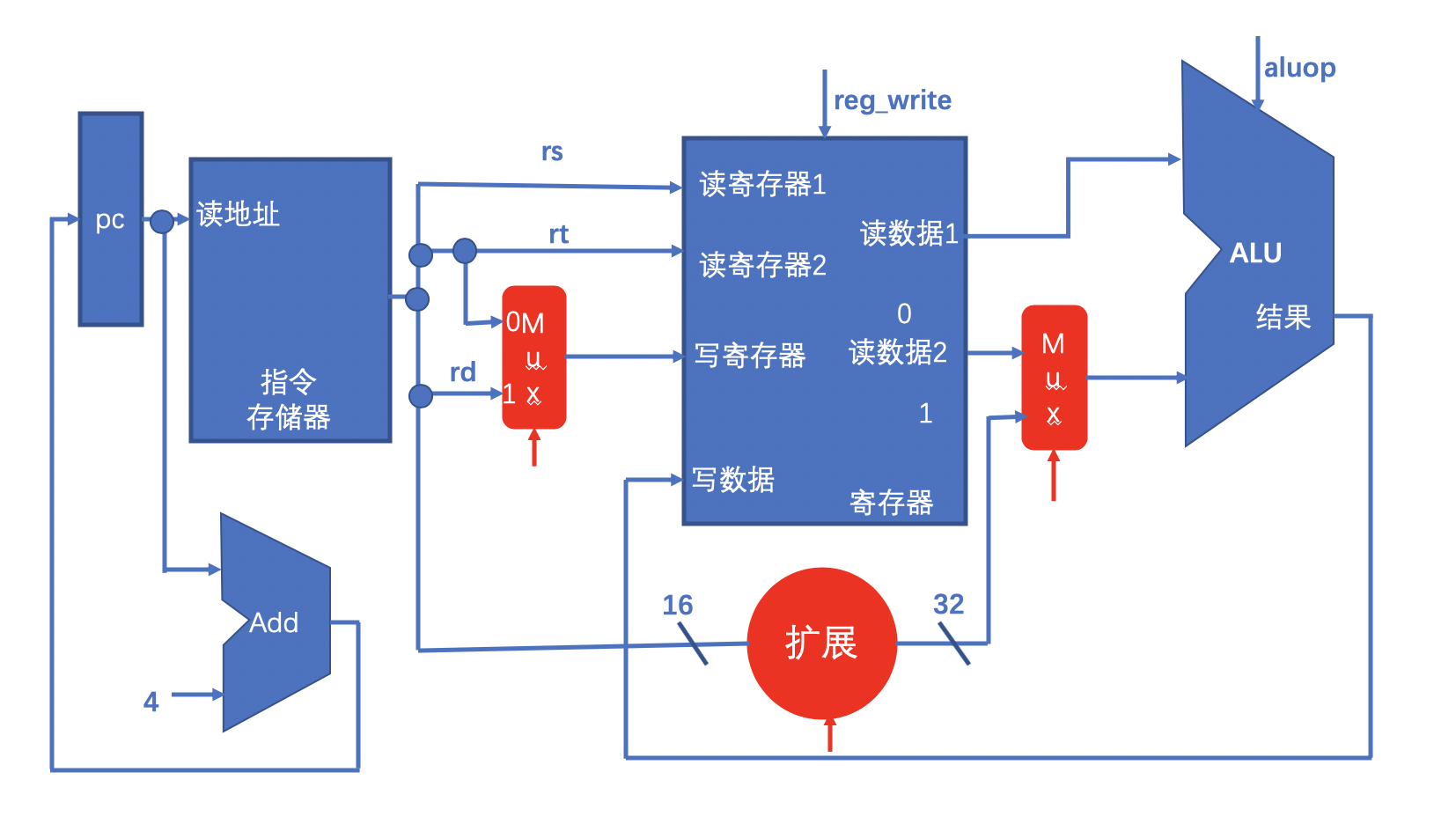
* 1. I型指令
     1. 指令分析、数据通路图

31~26位：op

25~21位：rs

20~16位：rt

15~0位：imm（立即数）



* + 1. 实现过程

1. 代码、遇到问题

alu.v

`include "header.v"

module alu(c,a,b,aluop);

output reg signed [31:0] c;

input signed [31:0] a;

input signed [31:0] b;

input [2:0] aluop;

always @(a or b or aluop)

case (aluop)

`ADD: c = a + b;

`SUB: c = a - b;

`AND: c = a & b;

`OR: c = a | b;

`SLT: c = (a < b) ? 32'd1 : 32'd0;

`LUI: c = b << 16; //增加了这个左移16位的操作

default: c = a + b;

endcase

endmodule

ctrl.v

`include "header.v"

module ctrl(reg\_write,aluop,op,funct,regdst,extop,alusrc);

output reg reg\_write;

output reg [2:0] aluop;

input [5:0] op;

input [5:0] funct;

output reg extop; //0代表0扩展，1代表符号扩展

output reg alusrc;

output reg regdst;

/\*assign reg\_write = (op == `op\_R)? 1 : 0;

assign aluop = (op == `op\_R) ? (funct == `funct\_addu) ? `ADD:

(funct == `funct\_subu) ? `SUB:

(funct == `funct\_add) ? `ADD:

(funct == `funct\_and) ? `AND:

(funct == `funct\_or) ? `OR:

(funct == `funct\_slt) ? `SLT:

`ADD:

(op == `op\_addi) ? `ADD:

(op == `op\_addiu) ? `ADD:

(op == `op\_andi) ? `AND:

(op == `op\_ori) ? `OR:

(op == `op\_lui) ? `LUI:

`ADD; \*/

always @(\*)

begin

reg\_write = 0; regdst = 0; extop = 0; alusrc = 1; aluop = `ADD;

case(op)

`op\_R:

begin

reg\_write = 1; regdst = 1; extop = 0; alusrc = 0;

case (funct)

`funct\_addu: aluop = `ADD;

`funct\_subu: aluop = `SUB;

`funct\_add: aluop = `ADD;

`funct\_and: aluop = `AND;

`funct\_or: aluop = `OR;

`funct\_slt: aluop = `SLT;

endcase

end

`op\_addi: begin reg\_write = 1; regdst = 0; extop = 1; alusrc = 1; aluop = `ADD; end

`op\_addiu: begin reg\_write = 1; regdst = 0; extop = 1; alusrc = 1; aluop = `ADD; end

`op\_andi: begin reg\_write = 1; regdst = 0; extop = 0; alusrc = 1; aluop = `AND; end

`op\_ori: begin reg\_write = 1; regdst = 0; extop = 0; alusrc = 1; aluop = `OR; end

`op\_lui: begin reg\_write = 1; regdst = 0; extop = 0; alusrc = 1; aluop = `LUI; end

endcase

end

endmodule

ext.v //新增了扩展单元

module ext (

input [15:0] immediate,

input ExtSel,

output [31:0] extended\_immediate

);

assign extended\_immediate = (ExtSel)?{{16{immediate[15]}}, immediate[15:0]} //符号扩展

:{{16{1'b0}}, immediate[15:0]}; //零扩展

endmodule

gpr.v //不变

module gpr(a,b,clock,reg\_write,num\_write,rs,rt,data\_write);

output [31:0] a;

output [31:0] b;

input clock;

input reg\_write;

input [4:0] rs; //读寄存器1

input [4:0] rt; //读寄存器2

input [4:0] num\_write; //写寄存器

input [31:0] data\_write; //写数据

reg [31:0] gp\_registers[31:0]; //32个寄存器

assign a = rs ? gp\_registers[rs] : 0;

assign b = rt ? gp\_registers[rt] : 0;

always @(posedge clock)

begin

if (reg\_write)

gp\_registers[num\_write] <= data\_write;

end

endmodule

header.v //增加了一些aluop和op

//aluop

`define ADD 3'b000

`define SUB 3'b001

`define AND 3'b010

`define OR 3'b011

`define SLT 3'b100

`define LUI 3'b101

//funct (when op=0)

`define funct\_addu 6'b100001

`define funct\_subu 6'b100011

`define funct\_add 6'b100000

`define funct\_and 6'b100100

`define funct\_or 6'b100101

`define funct\_slt 6'b101010

`define funct\_jr 6'b001000

//op

`define op\_R 6'b000000

`define op\_addi 6'b001000

`define op\_addiu 6'b001001

`define op\_andi 6'b001100

`define op\_ori 6'b001101

`define op\_lui 6'b001111

im.v

module im(instruction,pc);

output [31:0] instruction;

input [31:0] pc;

reg [31:0] ins\_memory[1023:0]; //4k指令存储器

assign instruction = ins\_memory[pc[11:0]>>2];

endmodule

pc.v

module pc(pc,clock,reset,npc);

output [31:0] pc;

input clock;

input reset;

input [31:0] npc;

reg [31:0] data;

always @(posedge clock or negedge reset)

begin

if (reset == 0)

data <= 32'h00003000;

else

data <= npc;

end

assign pc = data;

endmodule

s\_cycle\_cpu.v

`include "header.v"

module s\_cycle\_cpu(clock,reset);

//输入

input clock;

input reset;

wire [31:0] npc;

wire [31:0] pc;

wire [31:0] instruction;

wire [4:0] rs; //读寄存器1

wire [4:0] rt; //读寄存器2

wire [4:0] rd;

wire reg\_write;

wire [31:0] a;

wire [31:0] b;

wire [31:0] b1;

wire [31:0] c;

wire [2:0] aluop;

wire [5:0] op;

wire [5:0] funct;

wire regdst;

wire extop;

wire alusrc;

wire [15:0] imm;

wire [31:0] eximm;

wire [4:0] num\_write;

pc PC(.pc(pc),.clock(clock),.reset(reset),.npc(npc));

assign npc = pc + 4;

im IM(.instruction(instruction),.pc(pc));

assign op = instruction [31:26];

assign rs = instruction [25:21];

assign rt = instruction [20:16];

assign rd = instruction [15:11];

assign funct = instruction [5:0];

assign imm = instruction [15:0];

assign num\_write = regdst ? rd : rt;

ctrl CTRL(.reg\_write(reg\_write),.aluop(aluop),.op(op),.funct(funct),.regdst(regdst),.extop(extop),.alusrc(alusrc));

ext EXT(.immediate(imm), .ExtSel(extop), .extended\_immediate(eximm));

gpr GPR(.a(a),.b(b),.clock(clock),.reg\_write(reg\_write),.num\_write(num\_write),.rs(rs),.rt(rt),.data\_write(c));

assign b1 = alusrc ? eximm : b;

alu ALU(.c(c),.a(a),.b(b1),.aluop(aluop));

endmodule

比较顺利，没有遇到问题。

1. 思考疑问

新增了扩展单元，设置了extop这个变量判断这个指令需要符号扩展还是零扩展imm，将16位imm转换为32位eximm后即可进入alu计算。

* 1. MEM指令
     1. 指令分析、数据通路图

sw：存储数据，将GPR[rt]的值存入memory[Addr]。

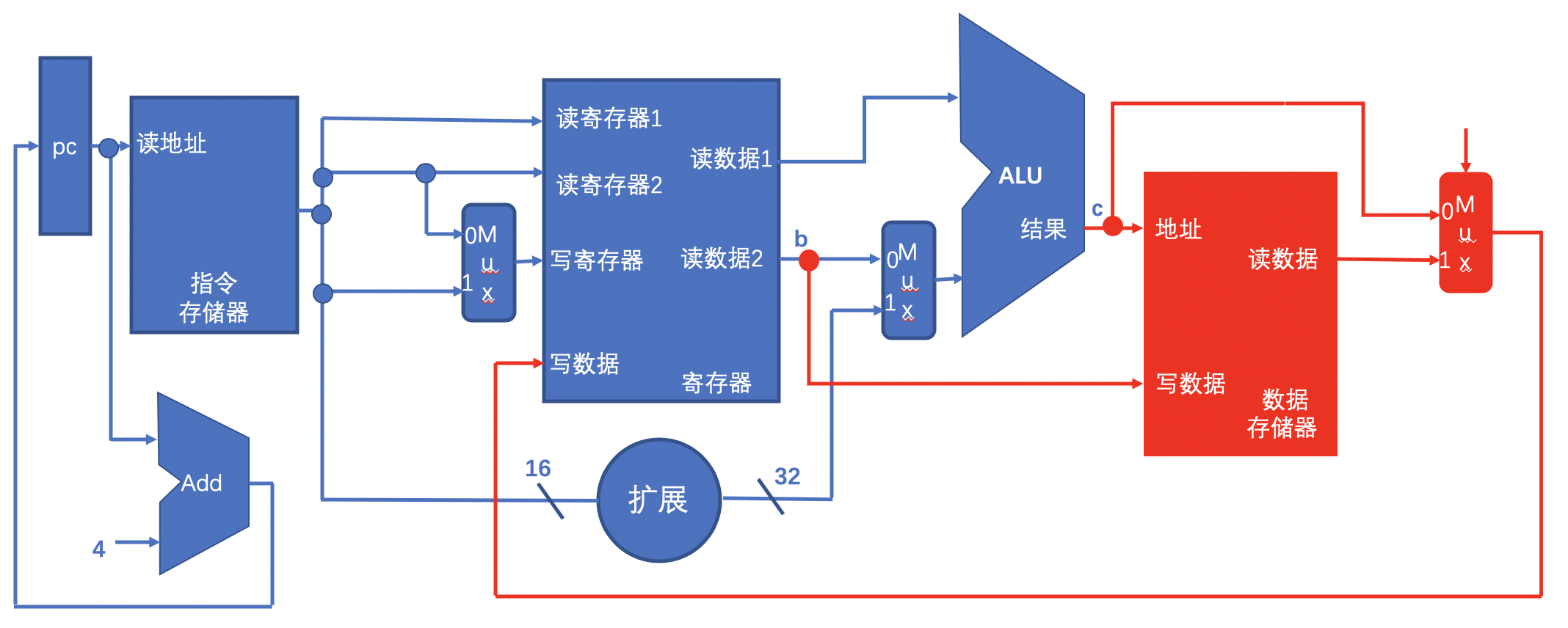
lw：加载数据，将memory[Addr]的值传入GPR[rt]。

31~26：op

25~21：base

20~16：rt

15~0：offset



* + 1. 实现过程

alu.v

`include "header.v"

module alu(c,a,b,aluop);

output reg signed [31:0] c;

input signed [31:0] a;

input signed [31:0] b;

input [2:0] aluop;

always @(a or b or aluop)

case (aluop)

`ADD: c = a + b;

`SUB: c = a - b;

`AND: c = a & b;

`OR: c = a | b;

`SLT: c = (a < b) ? 32'd1 : 32'd0;

`LUI: c = b << 16;

default: c = a + b;

endcase

endmodule

ctrl.v

`include "header.v"

module ctrl(reg\_write,aluop,op,funct,regdst,extop,alusrc,memwrite,memread);

output reg reg\_write;

output reg [2:0] aluop;

input [5:0] op;

input [5:0] funct;

output reg extop;

output reg alusrc;

output reg regdst;

output reg memread;

output reg memwrite;

/\*assign reg\_write = (op == `op\_R)? 1 : 0;

assign aluop = (op == `op\_R) ? (funct == `funct\_addu) ? `ADD:

(funct == `funct\_subu) ? `SUB:

(funct == `funct\_add) ? `ADD:

(funct == `funct\_and) ? `AND:

(funct == `funct\_or) ? `OR:

(funct == `funct\_slt) ? `SLT:

`ADD:

(op == `op\_addi) ? `ADD:

(op == `op\_addiu) ? `ADD:

(op == `op\_andi) ? `AND:

(op == `op\_ori) ? `OR:

(op == `op\_lui) ? `LUI:

`ADD; \*/

always @(\*)

begin

reg\_write = 0; regdst = 0; extop = 0; alusrc = 1; aluop = `ADD; memwrite = 0; memread = 0; //增加这两个代表从dm中写入还是读出数据

case(op)

`op\_R:

begin

reg\_write = 1; regdst = 1; extop = 0; alusrc = 0;

case (funct)

`funct\_addu: aluop = `ADD;

`funct\_subu: aluop = `SUB;

`funct\_add: aluop = `ADD;

`funct\_and: aluop = `AND;

`funct\_or: aluop = `OR;

`funct\_slt: aluop = `SLT;

endcase

end

`op\_addi: begin reg\_write = 1; regdst = 0; extop = 1; alusrc = 1; aluop = `ADD; end

`op\_addiu: begin reg\_write = 1; regdst = 0; extop = 1; alusrc = 1; aluop = `ADD; end

`op\_andi: begin reg\_write = 1; regdst = 0; extop = 0; alusrc = 1; aluop = `AND; end

`op\_ori: begin reg\_write = 1; regdst = 0; extop = 0; alusrc = 1; aluop = `OR; end

`op\_lui: begin reg\_write = 1; regdst = 0; extop = 0; alusrc = 1; aluop = `LUI; end

`op\_lw: begin reg\_write = 1; regdst = 0; extop = 1; alusrc = 1; aluop = `ADD; memwrite = 0; memread = 1; end

`op\_sw: begin reg\_write = 0; regdst = 1; extop = 1; alusrc = 1; aluop = `ADD; memwrite = 1; memread = 0; end

endcase

end

endmodule

dm.v //新增的用于存储的模块

module dm(data\_out,clock,mem\_write,address,data\_in);

output [31:0] data\_out;

input clock;

input mem\_write;

input [31:0] address;

input [31:0] data\_in;

reg [31:0] data\_memory[1023:0]; //4K数据存储器

assign data\_out = data\_memory[address>>2];

always @(posedge clock)

begin

if (mem\_write)

data\_memory[address[11:2]] <= data\_in;

end

endmodule

ext.v

module ext (

input [15:0] immediate,

input ExtSel,

output [31:0] extended\_immediate

);

assign extended\_immediate = (ExtSel)?{{16{immediate[15]}}, immediate[15:0]}

:{{16{1'b0}}, immediate[15:0]};

endmodule

gpr.v

module gpr(a,b,clock,reg\_write,num\_write,rs,rt,data\_write);

output [31:0] a;

output [31:0] b;

input clock;

input reg\_write;

input [4:0] rs; //读寄存器1

input [4:0] rt; //读寄存器2

input [4:0] num\_write; //写寄存器

input [31:0] data\_write; //写数据

reg [31:0] gp\_registers[31:0]; //32个寄存器

assign a = rs ? gp\_registers[rs] : 0;

assign b = rt ? gp\_registers[rt] : 0;

always @(posedge clock)

begin

if (reg\_write)

gp\_registers[num\_write] <= data\_write;

end

endmodule

header.v //增加了sw和lw两个op

//aluop

`define ADD 3'b000

`define SUB 3'b001

`define AND 3'b010

`define OR 3'b011

`define SLT 3'b100

`define LUI 3'b101

//funct (when op=0)

`define funct\_addu 6'b100001

`define funct\_subu 6'b100011

`define funct\_add 6'b100000

`define funct\_and 6'b100100

`define funct\_or 6'b100101

`define funct\_slt 6'b101010

`define funct\_jr 6'b001000

//op

`define op\_R 6'b000000

`define op\_addi 6'b001000

`define op\_addiu 6'b001001

`define op\_andi 6'b001100

`define op\_ori 6'b001101

`define op\_lui 6'b001111

`define op\_sw 6'b101011

`define op\_lw 6'b100011

im.v

module im(instruction,pc);

output [31:0] instruction;

input [31:0] pc;

reg [31:0] ins\_memory[1023:0]; //4k指令存储器

assign instruction = ins\_memory[pc[11:0]>>2];

endmodule

pc.v

module pc(pc,clock,reset,npc);

output [31:0] pc;

input clock;

input reset;

input [31:0] npc;

reg [31:0] data;

always @(posedge clock or negedge reset)

begin

if (reset == 0)

data <= 32'h00003000;

else

data <= npc;

end

assign pc = data;

endmodule

s\_cycle\_cpu.v

module s\_cycle\_cpu(clock,reset);

//输入

input clock;

input reset;

wire [31:0] npc;

wire [31:0] pc;

wire [31:0] instruction;

wire [4:0] rs; //读寄存器1

wire [4:0] rt; //读寄存器2

wire [4:0] rd;

wire reg\_write;

wire [31:0] a;

wire [31:0] b;

wire [31:0] b1;

wire [31:0] c;

wire [2:0] aluop;

wire [5:0] op;

wire [5:0] funct;

wire regdst;

wire extop;

wire alusrc;

wire [15:0] imm;

wire [31:0] eximm;

wire [4:0] num\_write;

wire memwrite;

wire memread;

//wire [31:0] data\_in;

wire [31:0] data\_out;

wire [31:0] bus\_out;

pc PC(.pc(pc),.clock(clock),.reset(reset),.npc(npc));

assign npc = pc + 4;

im IM(.instruction(instruction),.pc(pc));

assign op = instruction [31:26];

assign rs = instruction [25:21];

assign rt = instruction [20:16];

assign rd = instruction [15:11];

assign funct = instruction [5:0];

assign imm = instruction [15:0];

assign num\_write = regdst ? rd : rt;

ctrl CTRL(.reg\_write(reg\_write),.aluop(aluop),.op(op),.funct(funct),.regdst(regdst),.extop(extop),.alusrc(alusrc),.memwrite(memwrite),.memread(memread));

ext EXT(.immediate(imm), .ExtSel(extop), .extended\_immediate(eximm));

gpr GPR(.a(a),.b(b),.clock(clock),.reg\_write(reg\_write),.num\_write(num\_write),.rs(rs),.rt(rt),.data\_write(bus\_out));

assign b1 = alusrc ? eximm : b;

alu ALU(.c(c),.a(a),.b(b1),.aluop(aluop));

dm DM(.data\_out(data\_out),.clock(clock),.mem\_write(memwrite),.address(c),.data\_in(b));

assign bus\_out = memread ? data\_out : c;

endmodule

将上一次实验写的dm模块加入，此模块用来存储和读取数据，其中数据保存在data\_memory寄存器堆中。

操作和变量很多，容易搞混搞串，特别又对了一遍所有操作码应该有的memwrite和memread等的值解决了指令无法运行的bug。

* 1. J型指令
     1. 指令分析、数据通路图

beq：相等时跳转，此时pc不仅要+4也要加指令后面的立即数，即可跳转到想要的位置。

31~26：op

25~21：base

20~16：rt

15~0：offset

j：跳转

31~26：op

25~0：instr\_index

jal：跳转并链接

31~26：op

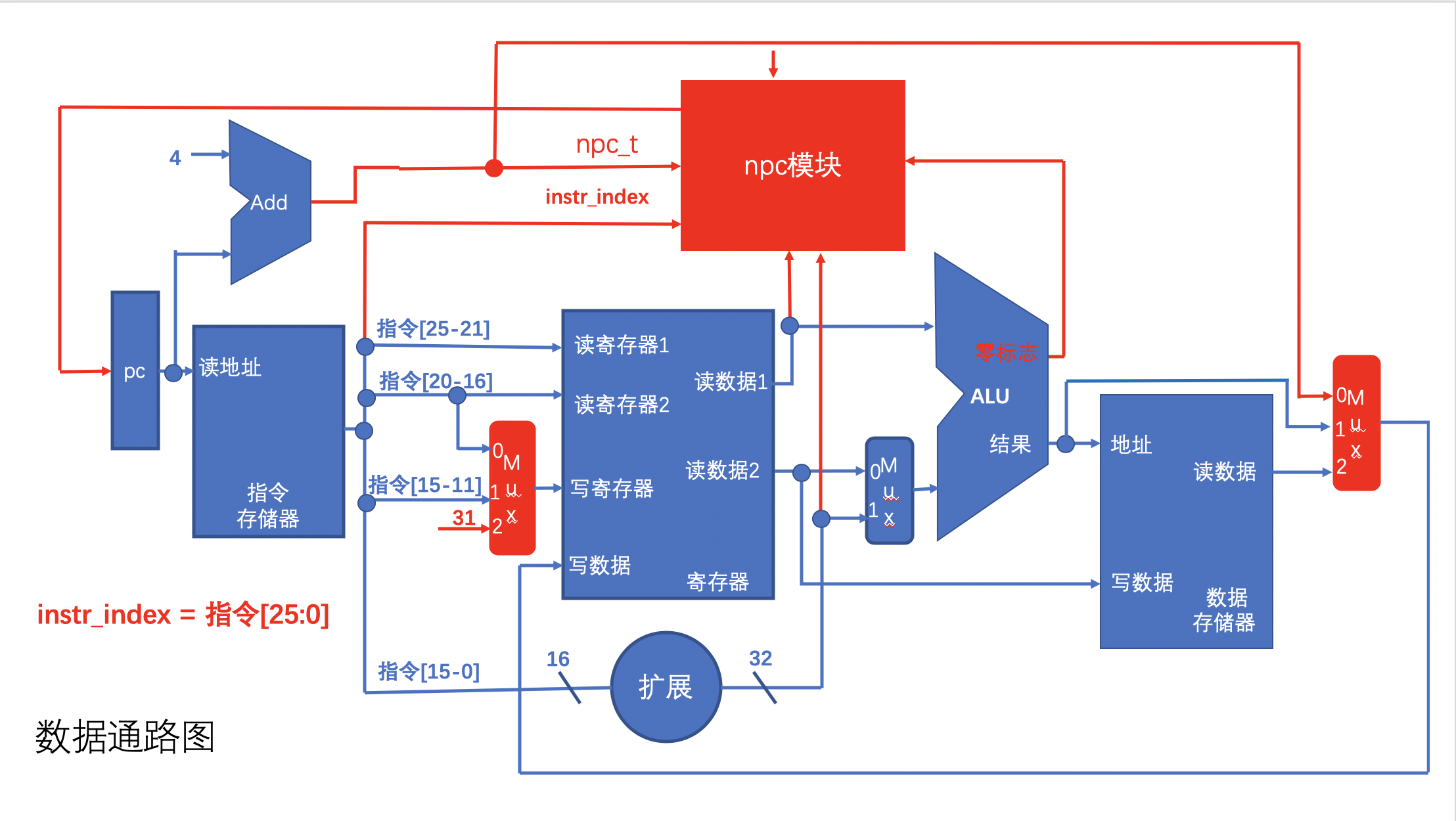
25~0：instr\_index

jr：跳转至寄存器

31~26：op

25~21：rs

20~0：空



* + 1. 实现过程

alu.v

`include "header.v"

module alu(c,a,b,aluop,zero);

output reg signed [31:0] c;

input signed [31:0] a;

input signed [31:0] b;

input [2:0] aluop;

output reg zero;

//always @(a or b or aluop)

always @(\*)

begin

zero = 0;

case (aluop)

`ADD: c <= a + b;

`SUB: c <= a - b;

`AND: c <= a & b;

`OR: c <= a | b;

`SLT: c <= $signed(a) < $signed(b) ? 32'd1 : 32'd0;

`LUI: c <= b << 16;

`EQB: c <= b; //直接等于b，不做任何操作

default: c <= b;

endcase

zero <= (c == 0) ? 1 : 0;

end

endmodule

ctrl.v

`include "header.v"

module ctrl(reg\_write,aluop,op,funct,regdst,extop,alusrc,memwrite,memread,s,r31);

output reg reg\_write;

output reg [2:0] aluop;

input [5:0] op;

input [5:0] funct;

output reg extop;

output reg alusrc;

output reg regdst;

output reg memread;

output reg memwrite;

output reg [1:0] s;

output reg r31;

always @(\*)

begin

reg\_write = 1'bx; regdst = 1'bx; extop = 1'bx; alusrc = 1'bx; aluop = 5'bxxxxx; memwrite = 1'bx; memread = 1'bx; s = `NONE; r31 = 1'b0;

//初始化为x，具体值后面再赋，便于调试和观察错误。

case(op)

`op\_R:

begin

reg\_write = 1; regdst = 1; extop = 0; alusrc = 0; memwrite = 0; memread = 0;

case (funct)

`funct\_addu: aluop = `ADD;

`funct\_subu: aluop = `SUB;

`funct\_add: aluop = `ADD;

`funct\_and: aluop = `AND;

`funct\_or: aluop = `OR;

`funct\_slt: aluop = `SLT;

`funct\_jr: begin reg\_write = 0; s = `JR; end

endcase

end

`op\_addi: begin reg\_write = 1; regdst = 0; extop = 1; alusrc = 1; aluop = `ADD; memwrite = 0; memread = 0; end

`op\_addiu: begin reg\_write = 1; regdst = 0; extop = 1; alusrc = 1; aluop = `ADD; memwrite = 0; memread = 0; end

`op\_andi: begin reg\_write = 1; regdst = 0; extop = 0; alusrc = 1; aluop = `AND; memwrite = 0; memread = 0; end

`op\_ori: begin reg\_write = 1; regdst = 0; extop = 0; alusrc = 1; aluop = `OR; memwrite = 0; memread = 0; end

`op\_lui: begin reg\_write = 1; regdst = 0; extop = 0; alusrc = 1; aluop = `LUI; memwrite = 0; memread = 0; end

`op\_lw: begin reg\_write = 1; regdst = 0; extop = 1; alusrc = 1; aluop = `ADD; memwrite = 0; memread = 1; end

//`op\_sw: begin reg\_write = 0; regdst = 1; extop = 1; alusrc = 1; aluop = `ADD; memwrite = 1; memread = 0; end

`op\_sw: begin reg\_write = 0; extop = 1; alusrc = 1; aluop = `ADD; memwrite = 1; end

`op\_beq: begin reg\_write = 0; regdst = 1; extop = 1; alusrc = 0; aluop = `SUB; memwrite = 0; s = `BEQ; end

`op\_j: begin reg\_write = 0; memwrite = 0; s = `J\_JAL; end

`op\_jal: begin reg\_write = 1; regdst = 0; alusrc = 0; aluop = `EQB; memwrite = 0; memread = 0; s = `J\_JAL; r31 = 1; end //增加了新的变量r31，在后面多路选择器中作为选择信号

endcase

end

endmodule

dm.v //不变

module dm(data\_out,clock,mem\_write,address,data\_in);

output [31:0] data\_out;

input clock;

input mem\_write;

input [31:0] address;

input [31:0] data\_in;

reg [31:0] data\_memory[1023:0]; //4K数据存储器

assign data\_out = data\_memory[address[11:2]];

always @(posedge clock)

begin

if (mem\_write)

data\_memory[address[11:2]] <= data\_in;

end

endmodule

ext.v

module ext (

input [15:0] immediate,

input ExtSel,

output [31:0] extended\_immediate

);

assign extended\_immediate = (ExtSel)?{{16{immediate[15]}}, immediate[15:0]}

:{{16{1'b0}}, immediate[15:0]};

endmodule

gpr.v

module gpr(a,b,clock,reg\_write,num\_write,rs,rt,data\_write);

output [31:0] a;

output [31:0] b;

input clock;

input reg\_write;

input [4:0] rs; //读寄存器1

input [4:0] rt; //读寄存器2

input [4:0] num\_write; //写寄存器

input [31:0] data\_write; //写数据

reg [31:0] gp\_registers[31:0]; //32个寄存器

assign a = rs ? gp\_registers[rs] : 0;

assign b = rt ? gp\_registers[rt] : 0;

always @(posedge clock)

begin

if (reg\_write)

gp\_registers[num\_write] <= data\_write;

end

endmodule

header.v //增加了s部分，让不同的跳转指令在npc模块中进行不同操作

//aluop

`define ADD 3'b000

`define SUB 3'b001

`define AND 3'b010

`define OR 3'b011

`define SLT 3'b100

`define LUI 3'b101

`define EQB 3'b110

//funct (when op=0)

`define funct\_addu 6'b100001

`define funct\_subu 6'b100011

`define funct\_add 6'b100000

`define funct\_and 6'b100100

`define funct\_or 6'b100101

`define funct\_slt 6'b101010

`define funct\_jr 6'b001000

//op

`define op\_R 6'b000000

`define op\_addi 6'b001000

`define op\_addiu 6'b001001

`define op\_andi 6'b001100

`define op\_ori 6'b001101

`define op\_lui 6'b001111

`define op\_sw 6'b101011

`define op\_lw 6'b100011

`define op\_beq 6'b000100

`define op\_j 6'b000010

`define op\_jal 6'b000011

//s

`define NONE 2'b00

`define BEQ 2'b01

`define J\_JAL 2'b10

`define JR 2'b11

im.v

module im(instruction,pc);

output [31:0] instruction;

input [31:0] pc;

reg [31:0] ins\_memory[1023:0]; //4k指令存储器

assign instruction = ins\_memory[pc[11:2]];

endmodule

npc.v //新增模块，产生下一个pc

/\*确定npc模块的输入输出端口：

* beq指令：①需要判断GPR[rs] == GPR[rt]，alu模块增加零标志输出；（zero）②跳转偏移地址通过指令低16位符号扩展后左移2位计算；（imm32）③beq指令是相对于pc+4进行跳转。（npc\_t）
* j、jal指令的跳转地址都是pc[31..28] || instr\_index || 02 。（instr\_index、pc）
* jr指令的跳转地址是GPR[rs] 。（a）
* s\_npc信号由控制模块产生，确定是哪类跳转指令。

输出npc为下条指令的地址。\*/

`include "header.v"

module npc(npc,npc\_t,instr\_index,offset,a,zero,s);

output reg [31:0] npc;

input [31:0] npc\_t; //npc\_t = pc+4

input [25:0] instr\_index;

input [31:0] offset; //指令低16位符号扩展

input [31:0] a; //alu模块a输出

input zero; //alu模块zero输出

input [1:0] s; //ctrl模块产生，确定当前指令类型

always @(\*)

begin

npc = npc\_t;

case(s)

`NONE: npc = npc\_t;

`BEQ:

begin

if(zero)

npc = npc\_t + {offset[29:0], 2'b00};

else

npc = npc\_t;

end

`J\_JAL: npc = {npc\_t[31:28], instr\_index, 2'b00};

`JR: npc = a;

default: npc = 32'hxxxxxxxx;

endcase

end

endmodule

pc.v

module pc(pc,clock,reset,npc);

output [31:0] pc;

input clock;

input reset;

input [31:0] npc;

reg [31:0] data;

always @(posedge clock or negedge reset)

begin

if (reset == 0)

data <= 32'h00003000;

else

data <= npc;

end

assign pc = data;

endmodule

s\_cycle\_cpu.v

`include "header.v"

module s\_cycle\_cpu(clock,reset);

//输入

input clock;

input reset;

wire [31:0] npc;

wire [31:0] pc;

wire [31:0] instruction;

wire [4:0] rs; //读寄存器1

wire [4:0] rt; //读寄存器2

wire [4:0] rd;

wire reg\_write;

wire [31:0] a;

wire [31:0] b;

wire [31:0] b1;

wire [31:0] c;

wire [2:0] aluop;

wire [5:0] op;

wire [5:0] funct;

wire regdst;

wire extop;

wire alusrc;

wire [15:0] imm;

wire [31:0] eximm;

wire [4:0] num\_write;

wire memwrite;

wire memread;

//wire [31:0] data\_in;

wire [31:0] data\_out;

wire [31:0] bus\_out;

wire [31:0] npc\_t;

wire [25:0] instr\_index;

wire zero;

wire [1:0] s;

wire r31;

pc PC(.pc(pc),.clock(clock),.reset(reset),.npc(npc));

assign npc\_t = pc + 4;

im IM(.instruction(instruction),.pc(pc));

assign op = instruction [31:26];

assign rs = instruction [25:21];

assign rt = instruction [20:16];

assign rd = instruction [15:11];

assign funct = instruction [5:0];

assign imm = instruction [15:0];

assign instr\_index = instruction [25:0];

ctrl CTRL(.reg\_write(reg\_write),.aluop(aluop),.op(op),.funct(funct),.regdst(regdst),.extop(extop),.alusrc(alusrc),.memwrite(memwrite),.memread(memread),.s(s),.r31(r31));

assign num\_write = r31 ? 5'd31 : (regdst ? rd : rt);

ext EXT(.immediate(imm),.ExtSel(extop),.extended\_immediate(eximm));

gpr GPR(.a(a),.b(b),.clock(clock),.reg\_write(reg\_write),.num\_write(num\_write),.rs(rs),.rt(rt),.data\_write(bus\_out));

//assign b1 = alusrc ? eximm : b;

assign b1 = r31 ? pc + 4 : (alusrc ? eximm : b);

alu ALU(.c(c),.a(a),.b(b1),.aluop(aluop),.zero(zero));

npc NPC(.npc(npc),.npc\_t(npc\_t),.instr\_index(instr\_index),.offset(eximm),.a(a),.zero(zero),.s(s));

dm DM(.data\_out(data\_out),.clock(clock),.mem\_write(memwrite),.address(c),.data\_in(b));

//assign bus\_out = r31 ? npc\_t : (memread ? data\_out : c);

assign bus\_out = memread ? data\_out : c;

endmodule

测试部分s\_cycle\_cpu\_tb.v

//此测试程序针对于添加完跳转指令后最终的单周期CPU

//不需要修改，配套code\_fibonacci2.txt使用

`timescale 10ns/1ns

module s\_cycle\_cpu\_tb;

reg CLOCK;

reg RESET;

s\_cycle\_cpu CPU(CLOCK,RESET);

always

#5 CLOCK = ~CLOCK;

initial

$readmemh("code\_fibonacci2.txt", CPU.IM.ins\_memory); //用code\_fibonacci3.txt初始化指令存储器

//$readmemh("C:\\Users\\guoxiaoyan\\Desktop\\code\_fibonacci2.txt", CPU.IM.ins\_memory);相对路径读不到，写绝对路径

integer i;

initial

begin

CLOCK = 1; RESET = 1;

#2 RESET = 0;

for(i=0;i<32;i=i+1)

CPU.GPR.gp\_registers[i] = 0;

#4 RESET = 1;

end

always @(CPU.PC.pc)

begin

if(CPU.PC.pc == 32'h0000\_3054)//程序最后一条指令地址

begin

$display("display execute time is %6d",$time);

$display("display ----------------------------------------------------");

#50 for(i=0;i<32;i=i+1)//仿真结束时打印寄存器的值

$display("display gp\_registers[%2d] = %8h",i,CPU.GPR.gp\_registers[i]);

$display("display ----------------------------------------------------");

for(i=0;i<64;i=i+1)//仿真结束时打印数据存储器前32个字的值

$display("display data\_memory[%2d] = %8h",i,CPU.DM.data\_memory[i]);

$finish;

end

end

endmodule

用来初始化指令寄存器的code\_fibonacci2.txt：

21ad0028

240a0001

ad000000

ad0a0004

340e0002

01ae4823

340f0001

21080008

012f602a

118f0007

8d04fff8

8d05fffc

0c000c14

ad020000

21080004

2129ffff

08000c08

3c0eabcd

ad0e0000

08000c13

00851021

03e00008

其汇编代码为fibonacci2.asm

.text

addi $t5,$t5,40 # $t5 = 20

li $t2, 1 # $t2 = 1

sw $zero, 0($t0) # store F[0] with 0

sw $t2, 4($t0) # store F[1] with 1

ori $t6, $zero, 2 # $t6 = 2

subu $t1, $t5, $t6 # the number of loop is (size-2)

ori $t7, $zero, 1 # $t7 = 1

addi $t0, $t0, 8 # point to F[3]

Loop:

slt $t4, $t1, $t7 # $t4 = ($t1 < 1) ? 1 : 0

beq $t4, $t7, Loop\_End # repeat if not finished yet

lw $a0, -8($t0) # $a0 = F[n-2]

lw $a1, -4($t0) # $a1 = F[n-1]

jal fibonacci # $v0 = fibonacci( F[n-2], F[n-1] )

sw $v0, 0($t0) # store F[n]

addi $t0, $t0, 4 # $t0 point to element

addi $t1, $t1, -1 # loop counter decreased by 1

j Loop

Loop\_End:

lui $t6, 0xABCD # $t6 = 0xABCD0000

sw $t6, 0($t0) # \*$t0 = $t6

Loop\_Forever:

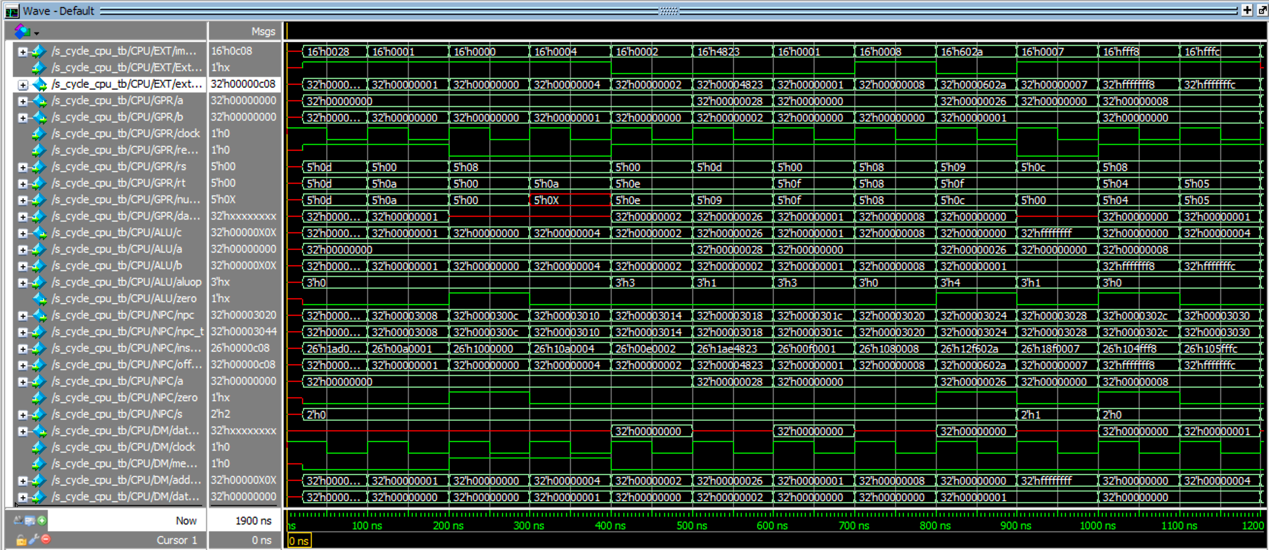
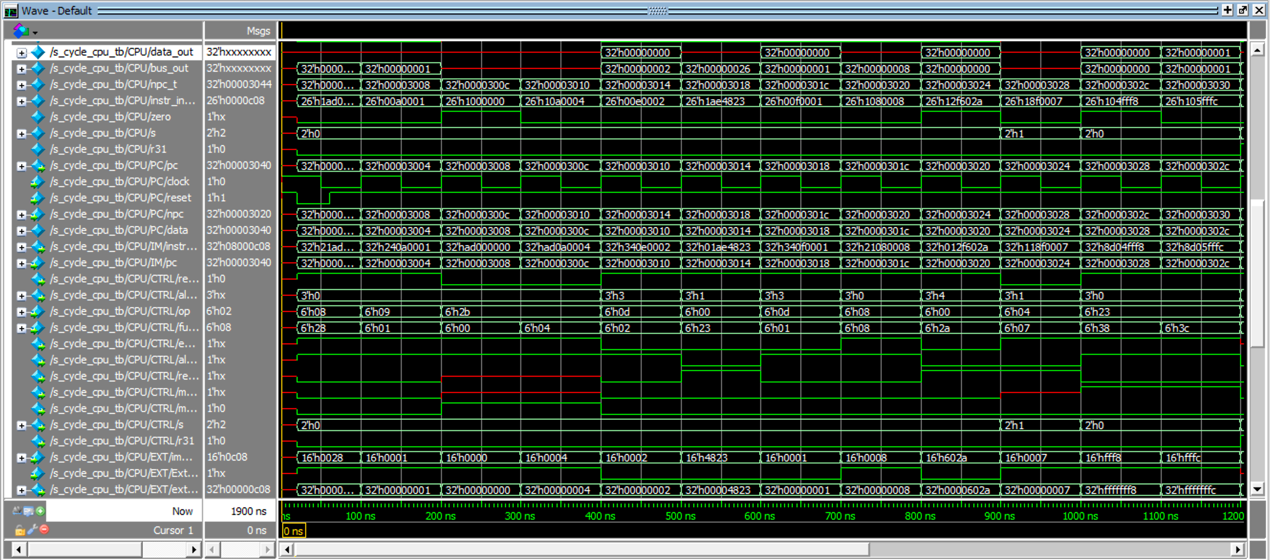
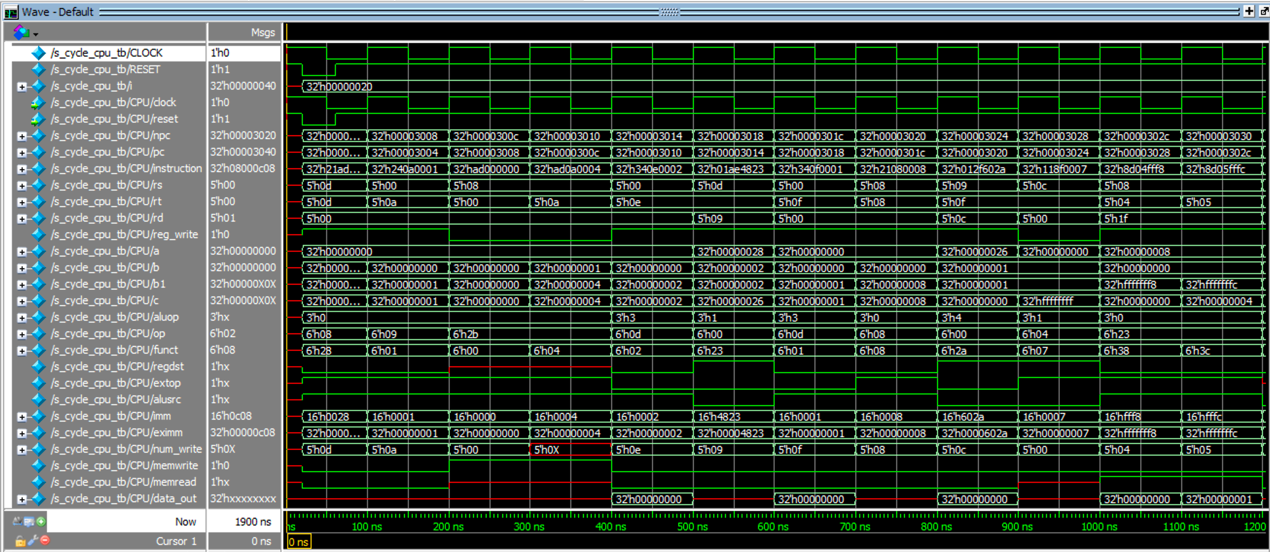
j Loop\_Forever # loop forever

fibonacci :

addu $v0, $a0, $a1 # $v0 = x + y

jr $ra # return

通过循环计算斐波那契数列前20位放入寄存器堆中。



1. 实验总结

花费了十几小时的时间，遇到过无数错误，不断debug和增加功能，终于写完了单周期CPU。很好地理解了单周期CPU的构造，自己动手写就也很有成就感。

1. 作业

4.1总结mips指令集中的三类指令：R型指令，I型指令和J型指令。

R型指令：addu无符号加，subu无符号减，add加，and与，or或，slt小于置1

I型指令：addi加立即数，addiu无符号加立即数，andi与立即数，ori或立即数，lui立即数加载至高位

J型指令：lw加载字、sw存储字

4.2详细分析slt功能（<）的实现。

GPR[rd] = (GPR[rs] < GPR[rt]) ? 1:0

1. 从im中取指，得到32位指令
2. 译码，判断为slt操作，找到rs和rt的字段
3. 从gpr中读出gpr[rs]和gpr[rt]的值
4. 进入alu做比较，将得到的值输出
5. 将结果写回寄存器
6. PC = PC + 4