# 实验报告一 单周期CPU

报告人：胡博岩 2019302785

1. **实验要求**

1.1：设计单周期cpu的基本模块，包括：pc模块（程序计数器），im模块（指令存储器），gpr模块（通用寄存器），alu模块（算术逻辑单元），dm模块（数据存储器）。

1.2：连接基本模块，实现能够执行addu指令的单周期CPU。

1.3：在1.2的基础上完善功能，实现能够执行以下R型指令的单周期CPU：addu，subu，add，and，or，sl。

1.4：在1.3的基础上完善功能，增加实现以下I型指令：

addi，addiu，andi，ori，lui。

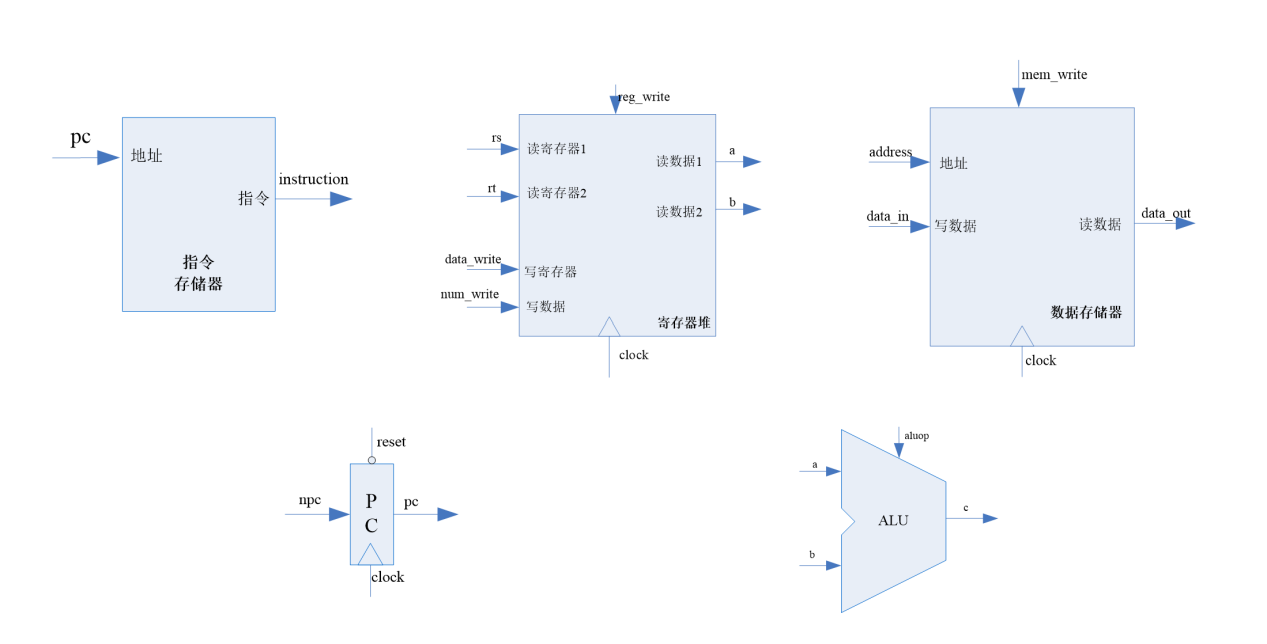
1.5：在1.4的基础上完善功能，增加实现以下和数据存储器相关的MEM型指令：lw，sw。

1.6：在1.5的基础上完善功能,增加实现以下跳转指令：

beq，j，jal，jr 。

最终，设计出一个能执行R型、I型、MEM型和跳转指令的单周期CPU。

1. **实验过程**
   1. **基本模块**



* + 1. **PC模块**

程序计数器（PC）是用于存放下一条指令所在单元的地址的地方。

当执行一条指令时，首先需要根据PC中存放的指令地址，将指令由内存取到[指令寄存器](https://baike.baidu.com/item/%E6%8C%87%E4%BB%A4%E5%AF%84%E5%AD%98%E5%99%A8/3219483" \t "https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E8%AE%A1%E6%95%B0%E5%99%A8/_blank)中，此过程称为取指令。与此同时，PC中的地址或自动加4或由跳转指令给出下一条指令的地址。此后经过分析指令，执行指令，完成第一条指令的执行，而后根据PC再取出第二条指令的地址，如此循环，执行每一条指令。



上图中：

npc为下条指令地址；

pc为当前指令地址；

clock为时钟信号，上升沿有效；

reset为复位信号，低电平有效，其作用为异步复位，pc复位为：32’h0000\_3000。

**模块代码如下：**

pc.v

//pc模块

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

output reg [31:0] pc; //当前指令地址

input [31:0] npc; //下条指令地址

input clock,reset;

always@(posedge clock or negedge reset)

begin

if(!reset)

pc <= 32'h0000\_3000;

else

pc <= npc;

end

endmodule

* + 1. **IM模块**

根据指令在存贮器中的地址（由PC模块给出），把指令从存贮器中取出来之后，需要有一个专门用于存放指令的地方，以便对指令进行分析和执行，即IM模块。

IM模块的输入地址pc是32位，但指令存储器ins\_memory只有4kB（即1KW），所以取pc的低12位作为ins\_memory的地址。 另一方面，虽然MIPS指令都是固定长度的32位（一个字），但是MIPS是按字节进行编址的，所以字地址为pc>>2



上图中：

pc为当前指令地址；

instruction为取出的指令；

**模块代码为：**

Im.v

//指令存储器

module im(instruction,pc);

output reg [31:0] instruction;

input [31:0] pc;

//指令存储器只有4k大小，所以pc低12位地址有效（屏蔽掉高位）

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

reg [31:0] address;

always @(\*)

begin

address <= pc[11:2];

instruction <= ins\_memory[address];

end

endmodule

* + 1. **GPR模块**

寄存器堆（GPR），是[CPU](https://baike.baidu.com/item/CPU" \t "https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8%E6%96%87%E4%BB%B6/_blank)中多个[寄存器](https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8" \t "https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8%E6%96%87%E4%BB%B6/_blank)组成的阵列，通常由快速的静态随机读写存储器（SRAM）实现。这种RAM具有专门的读端口与写端口，可以多路并发访问不同的寄存器。

CPU的[指令集架构](https://baike.baidu.com/item/%E6%8C%87%E4%BB%A4%E9%9B%86%E6%9E%B6%E6%9E%84" \t "https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8%E6%96%87%E4%BB%B6/_blank)总是定义了一批寄存器，用于在内存与CPU运算部件之间暂存数据。在更为简化的CPU，这些架构寄存器（architectural registers）一一对应与CPU内的物理存在的寄存器。在更为复杂的CPU，使用[寄存器重命名](https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8%E9%87%8D%E5%91%BD%E5%90%8D" \t "https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8%E6%96%87%E4%BB%B6/_blank)技术，使得执行期间哪个架构寄存器对应于哪个寄存器堆的物理存储条目（physical entry stores）是动态改变的。寄存器堆是指令集架构的一部分，程序可以访问，这与透明的CPU高速缓存（cache）不同。



上图中：

rs为读寄存器1编号，其值为instruction[25:21];

rt为读寄存器2编号，其值为instruction[20:16];

1. b分别为从寄存器1、2中读出的数据；

reg\_write为写使能信号，上升沿有效；

data\_write为写寄存器编号；

num\_write为写入寄存器的数据。

**模块代码为：**

gpr.v

//通用寄存器

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

output [31:0] a; //寄存器1的值

output [31:0] b; //寄存器2的值

input clock;

input reg\_write; //写使能信号

input [4:0] rs; //读寄存器1编号

input [4:0] rt; //读寄存器2编号

input [4:0] num\_write; //写寄存器编号

input [31:0] data\_write; //写数据

//32个寄存器

reg [31:0] gp\_registers[31:0];

assign a = gp\_registers[rs];

assign b = gp\_registers[rt];

always @(posedge clock)

begin

if(reg\_write&&num\_write!=0)

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

end

endmodule

* + 1. **ALU模块**

ALU（arithmetic and logic unit），即算术逻辑单元是能实现多组算术运算和逻辑运算的组合逻辑电路。

此阶段的ALU模块仅实现加法功能。



**模块代码如下：**

alu.v

//alu模块

module alu(c, a, b);

output reg [31:0] c;

input [31:0] a, b;

assign c = a+b;

endmodule

* + 1. **DM模块**

DM模块（数据存储器）的功能是建立一个4kB大小的存储器以及读写存储器。

对于读操作，data\_out一直输出address地址处的值；对于写操作，在clock上升沿且mem\_write信号有效时data\_in被写入address地址处。



上图中：

addres为数据存储的地址；

data\_in为写入数据存储器的数据；

mem\_write为写使能信号，上升沿有效；

data\_out为从数据存储器中读出的数据。

**模块代码如下：**

dm.v

//数据存储器

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

output [31:0] data\_out;

input clock,mem\_write;

input [31:0] address,data\_in;

//4KB数据存储器

reg [31:0] data\_memory[1023:0];

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

always @(posedge clock)

begin

if(mem\_write)

begin

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

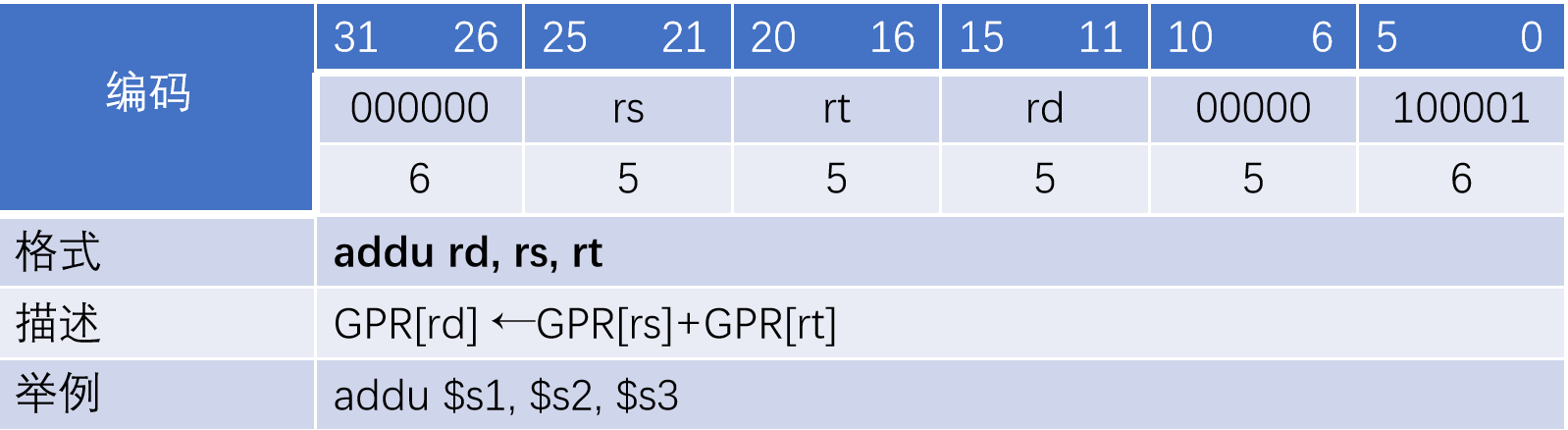
end

end

endmodule

* 1. **能够执行addu指令的单周期CPU**

**指令分析：**



按照pc值从指令存储器中取出指令，按照指令定义，从寄存器堆中读取GPR[rs]和GPR[rt]，用ALU模块实现GPR[rs]+GPR[rt]，将结果存入寄存器GPR[rd] 中。指令执行的同时，通过pc+4计算下条指令的地址npc。

**数据通路：**



代码实现：

在2.1的基础上，增加一个ADD模块以实现向后取址，在顶层模块中将各个模块逐一实现，并对其进行连线。

由于reg\_write作为控制信号对于R型指令一直为1，所以我们暂时将其数值直接设置为1’b1。

**模块代码如下：**

s\_cpu\_cycle.v

//顶层代码

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

input clock,reset;

wire [31:0] npc,pc,instruction,a,b,c;

wire [4:0] num\_write,rs,rt;

wire reg\_write;

assign rs=instruction[25:21];

assign rt=instruction[20:16];

assign num\_write=instruction[15:11];

assign reg\_write=1'b1;

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

add ADD(npc,pc,4);

im IM(instruction,pc);

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

alu ALU(c,a,b);

endmodule

add.v

//加法器

module add(c,a,b);

input [31:0] a,b;

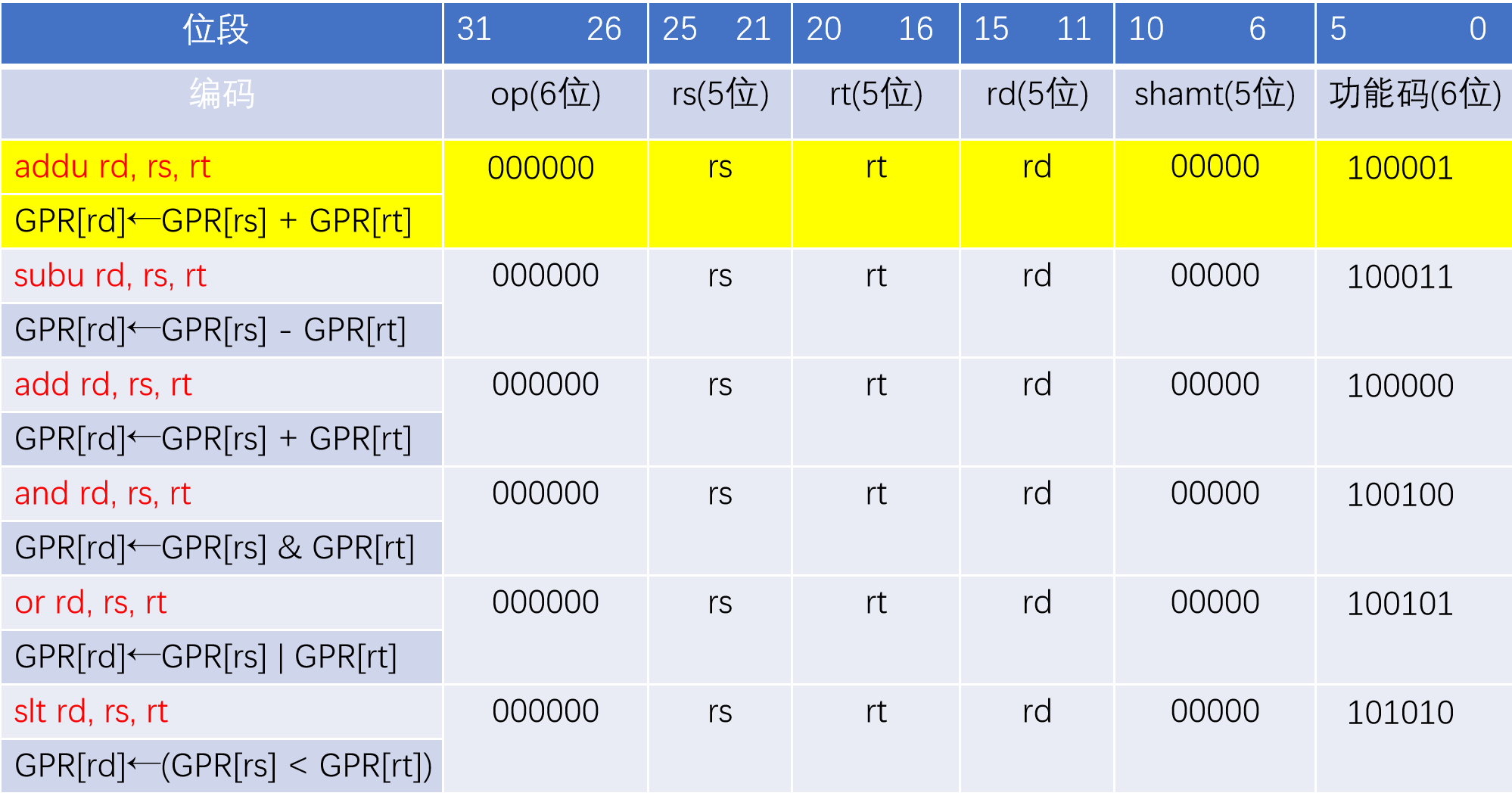
output [31:0] c;

assign c = a + b;

endmodule

* 1. **能够执行R型指令的单周期CPU**

**指令分析：**



以上R型指令格式的格式与addu指令类似：

op均为6’b000000；

rs为instruction[25:21]，rt为instruction[20:16]，rd为instruction[15:11]；

shamt均为5’b00000;

由funct（功能码）决定ALU进行的运算类型；

ALU的两个源操作数是寄存器GPR[rs]和GPR[rt]；

ALU运算的结果写入寄存器GPR[rd]。

**数据通路：**



**代码实现：**

在ALU模块中增加减、与、或和比较(<)的功能，并增加控制信号aluop，通过该信号来控制ALU模块进行哪种运算。

增加ctrl模块，根据op(instruction[31:26])和funct(instruction[5:0])产生控制信号（暂时只有aluop信号和reg\_write信号），对reg\_write信号我们仍将其直接设置为1’b1,而对于aluop信号，我们考虑到各R型信号的instruction后3位并不相同，直接通过ctrl模块将其设置为funct[3:0]。

增加头文件，在头文件中对控制信号的值进行宏定义。

**模块代码如下：**

head.v

//头文件

//aluop

`define ADDU\_ALUOP 4'b0001

`define SUB\_ALUOP 4'b0011

`define ADD\_ALUOP 4'b0000

`define AND\_ALUOP 4'b0100

`define OR\_ALUOP 4'b0101

`define SLT\_ALUOP 4'b1010

`define LU\_ALUOP 4'b1111

alu.v

//alu模块

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

output reg [31:0] c;

input [31:0] a, b;

input [3:0] aluop;

always @(\*)

case(aluop)

`ADDU\_ALUOP:c<=a+b;

`SUB\_ALUOP:c<=a-b;

`ADD\_ALUOP:c<=$signed(a)+$signed(b);

`AND\_ALUOP:c<=a&b;

`OR\_ALUOP:c<=a|b;

`SLT\_ALUOP:c<=$signed(a)<$signed(b);

`LU\_ALUOP:c<=b;

endcase

endmodule

crtl.v

//控制模块

module ctrl(aluop,op,funct);

output[3:0] aluop;

input[5:0] op;

input[5:0] funct;

assign aluop = funct[3:0];

endmodule

s\_cycle\_cpu.v

assign rs=instruction[25:21];

assign rt=instruction[20:16];

assign num\_write=instruction[15:11];

assign reg\_write=1'b1;

assign op=instruction[31:26];

assign funct=instruction[5:0];

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

add ADD(npc,pc);

im IM(instruction,pc);

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

ctrl CTRL(aluop,op,funct);

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

* 1. **添加I型指令**

**指令分析：**



对于以上I型指令：

op字段均不相同；

除lui指令rs为5’b00000外，rs为instruction[25:21]；

rt为instruction[20:16]；

指令低十六位为立即数字段，十六位立即数需要扩展为32位，而且需要根据s\_extend 指令类型确定是有符号扩展还是无符号扩展；

ALU的两个源操作数是寄存器GPR[rs]和扩展后的立即数imm\_32；

ALU结果写入寄存器GPR[rt]。

**数据通路：**



**代码实现：**

增加extend模块，将16位立即数imm扩展为32位imm\_32，而且需要根据s\_extend指令进行判断进行符号扩展或者无符号扩展。

增加mux模块，功能为2选1选择器。

对于GPR模块，写寄存器增加一个输入源（rt），需要增加数据选择器MUX\_1根据控制信号s\_b对第二个源操作数b的两个输入源（rd、rt）进行选择。

对于ALU模块，第二个源操作数增加一个输入源（立即数imm\_32），需要增加数据选择器MUX\_2根据控制信号s\_b对第二个源操作数b的两个输入源（data\_2、imm\_32）进行选择。

对于ctrl模块，增加输出控制信号s\_num\_write、s\_b、s\_extend。

**增加模块代码如下：**

head.v

增加s\_num\_write、s\_b、s\_extend控制信号数值的宏定义

//MUX\_1

`define RT\_S 2'b00

`define RD\_S 2'b01

`define R\_31\_S 2'b10

//MUX\_2

`define DATA\_2\_S 1'b0

`define IMM\_32\_S 1'b1

//EXTEND

`define ZERO\_EXTEND\_S 2'b00

`define SIGNED\_EXTEND\_S 2'b01

`define LU\_EXTEND\_S 2'b10

extend.v

//扩展模块

`include"head.v"

module extend(imm\_32,imm,s\_extend);

output reg [31:0] imm\_32;

input [15:0] imm;

input [1:0] s\_extend;

always @(\*)

begin

case(s\_extend)

`ZERO\_EXTEND\_S:imm\_32<={{16{1'b0}}, imm};//无符号扩展

`SIGNED\_EXTEND\_S:imm\_32<={{16{imm[15]}}, imm};//有符号扩展

`LU\_EXTEND\_S:imm\_32<={imm,{16{1'b0}}};//lui

endcase

end

endmodule

mux.v

//2选1选择器

module mux #(parameter width = 32)(res,input\_0,input\_1,s);

output reg [width - 1:0] res;

input [width - 1:0] input\_0,input\_1;

input s;

always@(\*)

begin

case(s)

1'b0:res<=input\_0;

1'b1:res<=input\_1;

endcase

end

endmodule

ctrl.v

利用case语句用op对R型指令和I指令进行分类，然后，对R型指令，统一处理；对I型指令，分开讨论，最终产生各控制信号。

always @(\*)

begin

case(op)

6'b000000:

begin

aluop<=funct[3:0];s\_num\_write<=`RD\_S;s\_b<=`DATA\_2\_S;s\_extend<=`SIGNED\_EXTEND\_S; end

6'b001000:

begin

aluop<=`ADD\_ALUOP;s\_num\_write<=`RT\_S;s\_b<=`IMM\_32\_S;s\_extend<=`SIGNED\_EXTEND\_S;

end

6'b001001:

begin

aluop<=`ADDU\_ALUOP;s\_num\_write<=`RT\_S;s\_b<=`IMM\_32\_S;s\_extend<=`SIGNED\_EXTEND\_S;

end

6'b001100:

begin

aluop<=`AND\_ALUOP; s\_num\_write<=`RT\_S; s\_b<=`IMM\_32\_S; s\_extend<=`ZERO\_EXTEND\_S;

end

6'b001101:

begin

aluop<=`OR\_ALUOP;s\_num\_write<=`RT\_S; s\_b<=`IMM\_32\_S; s\_extend<=`ZERO\_EXTEND\_S;

end

6'b001111:

begin

aluop<=`LU\_ALUOP; s\_num\_write<=`RT\_S; s\_b<=`IMM\_32\_S; s\_extend<=`LU\_EXTEND\_S;

end

endcase

end

s\_cycle\_cpu

assign rs=instruction[25:21];

assign rt=instruction[20:16];

assign rd=instruction[15:11];

assign reg\_write=1'b1;

assign imm=instruction[15:0];

assign op=instruction[31:26];

assign funct=instruction[5:0];

pc PC(pc,clock,reset,npc);//pc模块

add ADD(npc,pc);//加法器模块

im IM(instruction,pc);//指令存储器

ctrl CTRL(aluop,s\_num\_write,s\_b,s\_extend,op,funct);//控制模块

extend EXTEND(imm\_32,imm,s\_extend);

mux MUX\_1(num\_write,rt,rd,s\_num\_write);

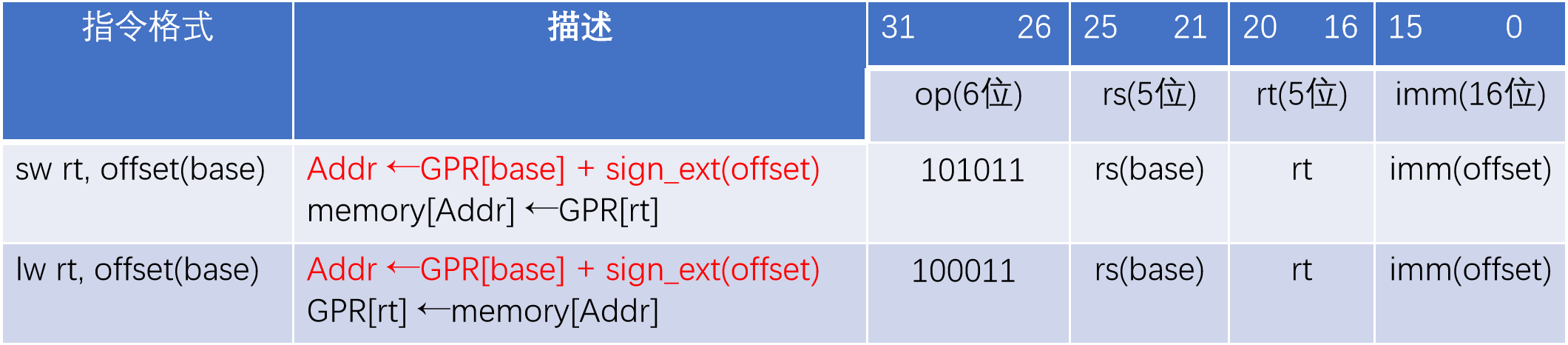
gpr GPR(a,data\_2,clock,reg\_write,num\_write,rs,rt,c);//通用寄存器

mux MUX\_2(b,data\_2,imm\_32,s\_b);

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

* 1. **添加MEM型指令**

**指令分析：**



对于以上MEM型指令：

op决定指令功能；

rs为instruction[25:21]，并且为基址base；

rt为instruction[20:16]；

Imm(offset)为instruction[15:0]；

数据存储器的地址address=基址(GPR[rs])+偏移（imm符号扩展）；

lw指令数据存储器中读出的结果写入寄存器GPR[rt]；

sw指令把寄存器GPR[rt]的值写入数据存储器DM[address]处。

**数据通路：**



**代码实现：**

需要在顶层模块中连入DM模块（数据存储器）；

数据存储器地址address计算过程就是ALU模块执行立即数加法；

对于ctrl模块，需要增加输出控制信号s\_data\_write、mem\_write，并在case语句中增加考虑MEM型指令对应的各控制信号；

对于寄存器堆，写数据增加一个输入源（数据存储器读结果data\_out），需要增加数据选择器MUX\_3根据控制信号s\_data\_write对两个输入源（c、data\_out）进行选择。

**模块代码如下：**

head.v

增加s\_data\_write控制信号数值的宏定义

//MUX\_3

`define NPC\_1\_S 2'b00

`define C\_S 2'b01

`define DATA\_OUT\_S 2'b10

ctrl.v

在case语句中增加考虑MEM型指令对应的各控制信号

always @(\*)

begin

case(op)

6'b101011:

begin

aluop<=ADD\_ALUOP; reg\_write<=1'b0; mem\_write<=1'b1;s\_num\_write<=RT\_S;

s\_b<=IMM\_32\_S; s\_extend<=SIGNED\_EXTEND\_S; s\_data\_write<=DATA\_OUT\_S;

end

6'b100011:

begin

aluop<=ADD\_ALUOP;reg\_write<=1'b1;mem\_write<=1'b0;s\_num\_write<=RT\_S;

s\_b<=IMM\_32\_S; s\_extend<=SIGNED\_EXTEND\_S; s\_data\_write<=`DATA\_OUT\_S;

end

endcase

end

s\_cycle\_cpu.v

assign rs=instruction[25:21];

assign rt=instruction[20:16];

assign rd=instruction[15:11];

assign imm=instruction[15:0];

assign address = c;

assign op=instruction[31:26];

assign funct=instruction[5:0];

pc PC(pc,clock,reset,npc);//pc模块

add ADD(npc,pc);//加法器模块

im IM(instruction,pc);//指令存储器

ctrl CTRL(aluop,reg\_write,s\_num\_write,s\_b,s\_extend,s\_data\_write,mem\_write,op,funct);//控制模块

extend EXTEND(imm\_32,imm,s\_extend);

mux #(5) MUX\_1(num\_write,rt,rd,s\_num\_write);

gpr GPR(a,data\_2,clock,reg\_write,num\_write,rs,rt,data\_write);//通用寄存器

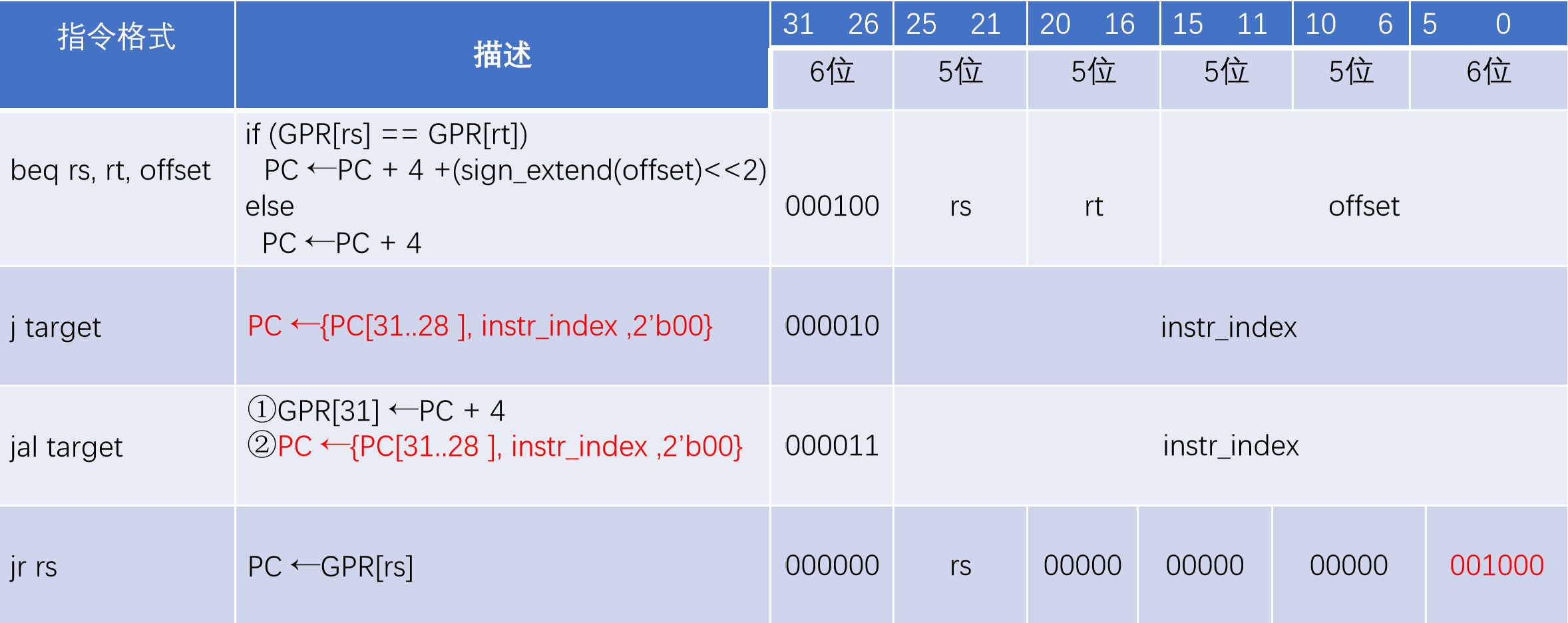
mux MUX\_2(b,data\_2,imm\_32,s\_b);

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

dm DM(data\_out,clock,mem\_write,address,data\_2);//数据存储器

mux MUX\_3(data\_write,c,data\_out,s\_data\_write);

* 1. **添加J型指令**



**2.6.1绝对跳转指令j和jal**

**指令分析：**

对于J和jal指令：

op决定指令类型；

j、jal指令的跳转地址，即npc都是绝对地址{PC[31..28 ],instr\_index ,2’b00}；

jal指令还需要把返回地址PC+4保存在GPR[31] ；

**数据通路：**



**代码实现：**

对于pc寄存器，增加一个输入源（绝对地址），需要增加数据选择器MUX\_4根据s\_npc控制信号对两个输入源（绝对地址、npc）进行选择；

对于寄存器堆，写寄存器号增加一个输入源31，数据选择器MUX\_1由二选一选择器改为三选一选择器；写数据增加一个输入源pc+4，数据选择器MUX\_3由二选一选择器改为三选一选择器。

**2.6.2寄存器跳转指令jr**

**指令分析：**

对于jr指令：

这是一条R型指令；

funct决定指令类型；

跳转地址为GPR[rs]；

考虑利用其funct将其与其他R型指令区分开进行处理，以得到其对应的控制信号。

**数据通路：**



**代码实现：**

对于pc寄存器，增加一个输入源，数据选择器MUX\_4由二选一选择器改为三选一选择器。

**2.6.3分支跳转指令beq**

**指令分析：**

对于beq指令：

op决定指令类型；

需要运用到ALU的减法功能对GPR[rs]和GPR[rt]进行比较；

当GPR[rs] == GPR[rt]时，npc= pc + 4 +(sign\_extend(offset)<<2)，

否则，npc =pc + 4。

**数据通路：**



**代码实现：**

比较GPR[rs] 和GPR[rt]是否相等，可以采用ALU执行减法功能，结果为0表示相等，所以ALU模块增加一个输出信号zero；

对于pc寄存器，增加一个输入源（相对跳转地址），数据选择器MUX\_4由三选一选择器改为四选一选择器；

对于MUX\_4数据选择器，增加一个输入源（pc + 4 +(sign\_extend(offset)<<2))，需要增加数据选择器MUX\_5根据控制信号zero对两个输入源（pc + 4 +(sign\_extend(offset)<<2)、npc）进行选择。

**代码模块如下：**

mux\_3.v

//3选1选择器

module mux\_3 #(parameter width = 32)(res,input\_0,input\_1,input\_2,s);

output reg [width - 1:0] res;

input [width - 1:0] input\_0,input\_1,input\_2;

input [1:0] s;

always@(\*)

begin

case(s)

2'b00:res<=input\_0;

2'b01:res<=input\_1;

2'b10:res<=input\_2;

endcase

end

endmodule

mux\_4.v

//4选1选择器

module mux\_4 #(parameter width = 32)(res,input\_0,input\_1,input\_2,input\_3,s);

output reg [width - 1:0] res;

input [width - 1:0] input\_0,input\_1,input\_2,input\_3;

input [1:0] s;

always@(\*)

begin

case(s)

2'b00:res<=input\_0;

2'b01:res<=input\_1;

2'b10:res<=input\_2;

2'b11:res<=input\_3;

endcase

end

endmodule

ctrl.v

增加控制信号s\_pc，增加考虑j、jal、jr、beq指令对应的控制信号，其中在考虑jr时，使用if-else语句将其与其他R型指令分开考虑

always @(\*)

begin

case(op)

6'b000000:begin

if(funct==6'b001000)

begin

aluop<=`ADD\_ALUOP;reg\_write<=1'b0;s\_num\_write<=`RD\_S;

s\_b<=`DATA\_2\_S;s\_extend<=`SIGNED\_EXTEND\_S;s\_data\_write<=`C\_S;

mem\_write<=1'b0;s\_pc<=`REG\_S;

end

else

begin

aluop<=funct[3:0];reg\_write<=1'b1;s\_num\_write<=`RD\_S;

s\_b<=`DATA\_2\_S;s\_extend<=`SIGNED\_EXTEND\_S;s\_data\_write<=`C\_S;

mem\_write<=1'b0;s\_pc<=`NPC\_2\_S;

end

End

6'b000010:begin

aluop<=`ADD\_ALUOP;reg\_write<=1'b0;s\_num\_write<=`R\_31\_S;

s\_b<=`DATA\_2\_S;s\_extend<=`SIGNED\_EXTEND\_S;s\_data\_write<=`C\_S;

mem\_write<=1'b0;s\_pc<=`ABS\_S;

end

6'b000011:begin

aluop<=`ADD\_ALUOP;reg\_write<=1'b1;s\_num\_write<=`R\_31\_S;

s\_b<=`DATA\_2\_S;s\_extend<=`SIGNED\_EXTEND\_S;s\_data\_write<=`NPC\_1\_S;

mem\_write<=1'b0;s\_pc<=`ABS\_S;

end

6'b000100:begin

aluop<=`SUB\_ALUOP;reg\_write<=1'b0;s\_num\_write<=`R\_31\_S;

s\_b<=`DATA\_2\_S;s\_extend<=`SIGNED\_EXTEND\_S;s\_data\_write<=`C\_S;

mem\_write<=1'b0;s\_pc<=`BEQ\_S;

end

s\_cycle\_cpu.v

assign rs=instruction[25:21];

assign rt=instruction[20:16];

assign rd=instruction[15:11];

assign imm=instruction[15:0];

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

assign op=instruction[31:26];

assign funct=instruction[5:0];

assign address = c;

assign reg\_addr = a;

assign abs\_addr = {pc[31:28],instr\_index,2'b00};

add ADD\_2(beq\_addr\_1,npc,imm\_32<<2);

mux MUX\_5(beq\_addr,npc,beq\_addr\_1,zero);

mux\_4 MUX\_4(n\_pc,beq\_addr,reg\_addr,abs\_addr,npc,s\_pc);

pc PC(pc,clock,reset,n\_pc);//pc模块

add ADD\_1(npc,pc,4);//加法器模块

im IM(instruction,pc);//指令存储器

ctrl CTRL(aluop,reg\_write,s\_num\_write,s\_b,s\_extend,s\_data\_write,mem\_write,s\_pc,op,funct);//控制模块

extend EXTEND(imm\_32,imm,s\_extend);

mux\_3 #(5) MUX\_1(num\_write,rt,rd,5'b11111,s\_num\_write);

gpr GPR(a,data\_2,clock,reg\_write,num\_write,rs,rt,data\_write);//通用寄存器

mux MUX\_2(b,data\_2,imm\_32,s\_b);

alu ALU(zero,c,a,b,aluop);//alu

dm DM(data\_out,clock,mem\_write,address,data\_2);//数据存储器

mux\_3 MUX\_3(data\_write,npc,c,data\_out,s\_data\_write);

mips\_tb.v

module mips\_tb();

reg clk, rst;

s\_cycle\_cpu U\_MIPS(

.clock(clk), .reset(rst)

);

initial begin

$readmemh( "test.txt" , U\_MIPS.IM.ins\_memory ) ;

clk = 1 ;rst = 1 ;

#5 ;

rst = 0 ;

#20 ;

rst = 1 ;

end

always

#(50) clk = ~clk;

endmodule

test.txt

341d000c

34021234

34033456

00432021

00643023

ac020000

ac030004

afa40004

8c050000

10450001

8fa30004

8c050004

1065fffd

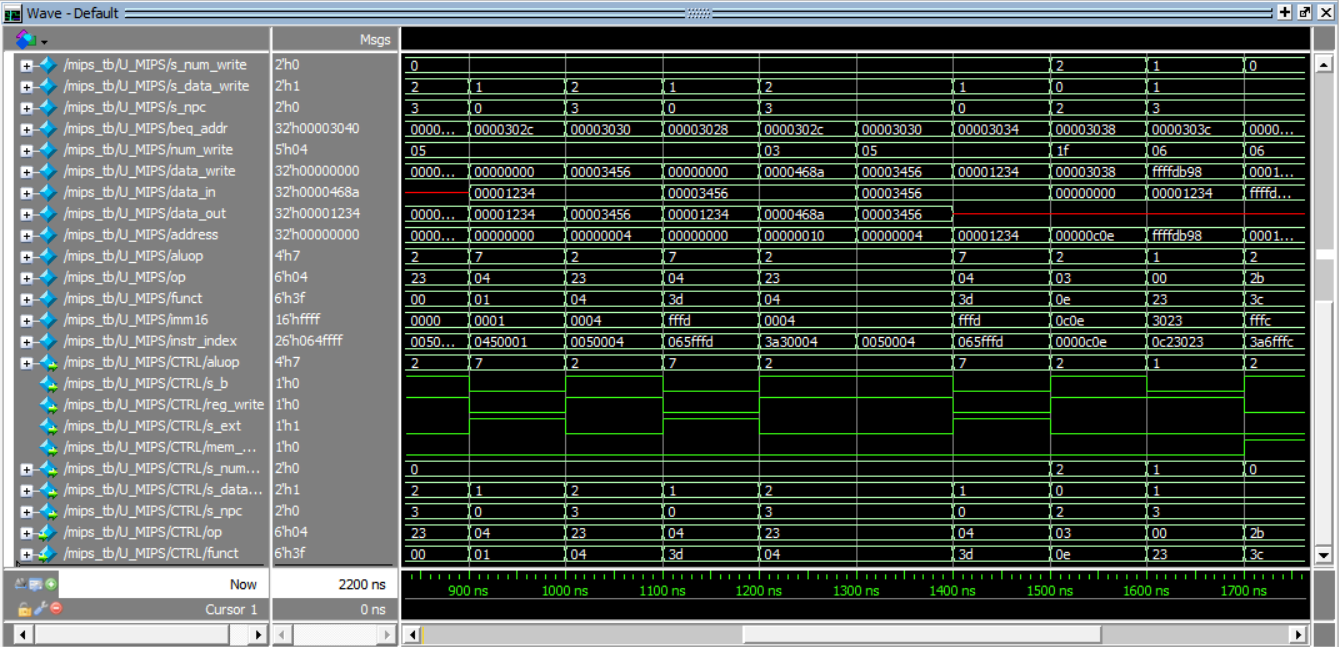
0c000c0e

00c23023

afa6fffc

1064ffff

**仿真波形：**



仿真结果符合预期

1. **实验总结**

**首先，我在实验遇到了很多问题：**

1. 虽然在上学期的数字逻辑设计课程中我们学习过verilog语言，但奈何学艺不精，故我重新深入学习了verilog的语法和quastadim软件的使用。

例如一些基础知识：

在always语块中进行赋值的信号要设置成reg类型，并使用<=进行赋值；

在assign语块中进行赋值的信号要设置成wire类型，并使用=进行赋值；

端口信号先后顺序要和模块定义时一样，或者在实例化时指明对应的信号。

还例如：

数据选择器模块里module mux #(parameter width = 32)(res,input\_0,input\_1,s);中paramater的用法作用；

头文件中宏定义的写法和其他文件中对其的引用。

1. 当处理信号过多时，将信号误认混淆。

例如在编写添加J型指令的单周期CPU时，我将信号npc和n\_pc混淆误用，导致出现错误。

**其次，再谈谈我对单周期CPU的理解：**

首先，要设计一个单周期CPU，第一步就是要设计好它的数据通路，然后再将各个模块分别进行实现，最后在顶层端口将数据通路连接起来可以降低难度。设计好数据通路以后就要给控制信号赋值，在设计好的数据通路中自己手动运行各条指令，分析每条指令在不同的基本模块和选择器处分别应该为控制信号选择什么值，并在控制模块进行赋值。单周期CPU的执行过程就是，在PC对应的指令寄存器处取指，对指令进行译码，根据译码的结果判断指令，并给相应的控制信号赋值，在设计好的数据通路中完成操作，如果是R型指令，I型指令，MEM型指令进行算术运算，即ALU操作，MEM型指令还需用到数据存储器单元，如果是J型指令，根据具体指令对PC进行操作。

**最后，说一下我对这次实验的感受：**

我感觉这次实验首先使我加深了对单周期cpu各方面上的理解，使我能够更好地学计算机组成与系统结构的理论课程，其次，经过这次实验，我的verilog语言的编程能力和对quastasim软件的使用能力也都得到了提高，可以说是收获满满。