计算机组成原理实验——36条单周期CPU

一. 个人信息

学号：072110112

姓名：冉中益

班号：1621102

专业：计算机科学与技术

二. 模块定义及代码实现

1. 数据通路

datapath

基本描述

数据通路模块 接收各控制信号 组合所有功能模块以实现各指令对于数据的操作

模块接口

接口名	方向	描述
RegDst	I	控制信号 选择目标寄存器
RegWr	I	寄存器组写使能信号
ALUSrc	I	控制信号 选择ALU输入口来自busB还是扩展器
MemWr	I	数据存储器写使能信号
MemtoReg	I	控制信号 选择写入寄存器的数据来自ALU还是数据存储器
Branch	I	控制信号 1时表示为有条件跳转指令
clk	I	时钟信号
reset	I	复位信号
Jump	I	控制信号 1时表示为无条件跳转指令
ExtOp	I	扩展器操作控制信号
ALUctr	I	ALU操作控制信号
instruction	O	指令输出

功能

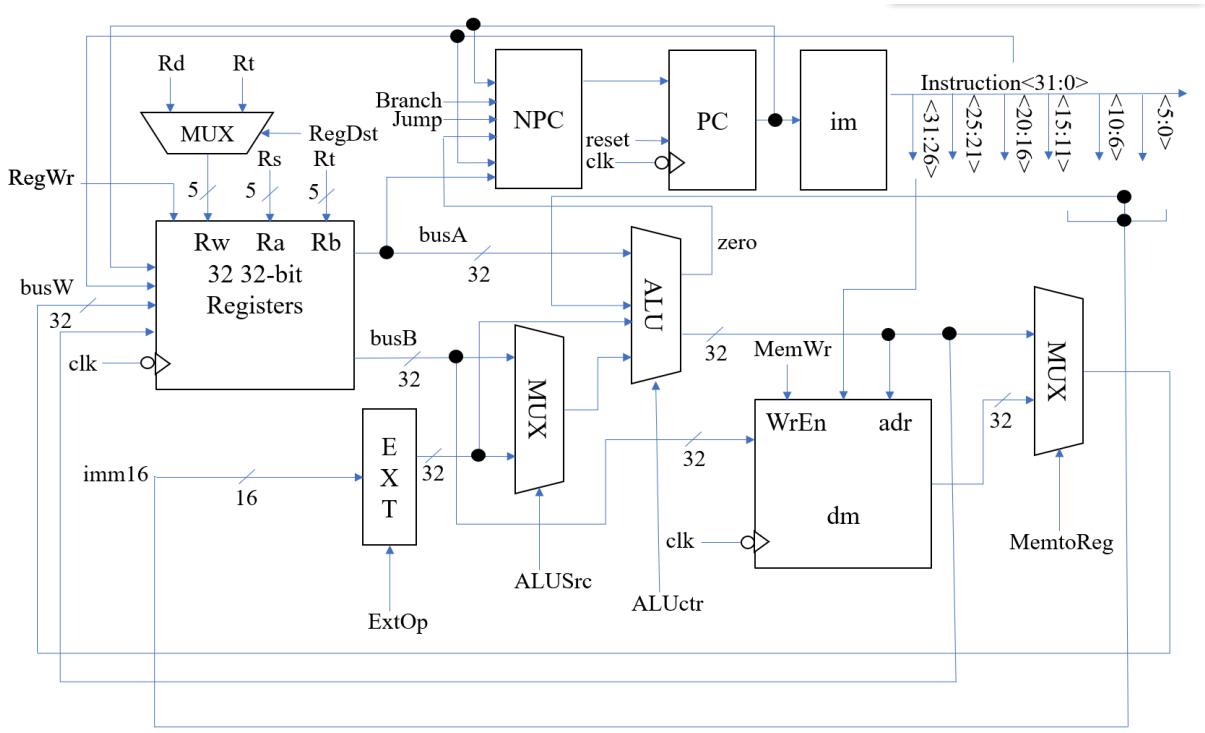
1. 根据输入的各控制信号完成对不同指令的数据操作
2. 输出下指令给控制器 使其产生下指令的控制信号备用

代码实现

```
1  `include"PC.v"
2  `include"NPC.v"
3  `include"im.v"
4  `include"RegFile.v"
5  `include"extender.v"
6  `include"mux.v"
7  `include"ALU.v"
```

```
8  `include"dm.v"
9
10 module datapath(RegDst, RegWr, ALUSrc, MemWr, MemtoReg, ExtOp, ALUctr,
   Branch, Jump, clk, reset, instruction);
11
12 input RegDst, RegWr, ALUSrc, MemWr, MemtoReg, Branch, clk, reset, Jump;
13 input[1:0] ExtOp;
14 input[4:0] ALUctr;
15 output[31:0] instruction;
16
17 wire[31:2]  PC;
18 wire[31:2]  NPC;
19 wire    zero;
20 wire[31:0]  alure;     //alu result
21 wire[31:0]  exout;
22 wire[31:0]  r1,r2;
23 wire[31:0]  data2;
24 wire[31:0]  wdata;
25 wire[31:0]  dout;
26
27 PC pc(NPC, clk, reset, PC);
28 NPC npc(PC, zero, Branch, instruction[31:26], instruction[15:0],
   instruction[25:0], Jump, NPC, r1, instruction[25:21],
   instruction[20:16], instruction[15:11], instruction[10:6],
   instruction[5:0]);
29 im_4k ReOrM(PC[11:2], instruction);
30 RegFile RM(instruction[31:26], PC, instruction[25:21],
   instruction[20:16], instruction[15:11], instruction[10:6],
   instruction[5:0], wdata, RegWr, RegDst, r1, r2, clk, reset);
31 extender extM(instruction[15:0], ExtOp, exout);
32 mux2 mux2_32M(r2, exout, ALUSrc, data2);
33 ALU AM(ALUctr, r1, data2, instruction[10:6], alure, zero);
34 dm_4k DMM(instruction[31:26], alure, r2, dout, clk, MemWr);
35 mux2 mux2_M(alure, dout, MemtoReg, wdata);
36
37 endmodule
```

（附）36条单周期数据通路图



分析：

1. **PC模块**： 输入端口为NPC, reset, clk,输出端口为PC；
2. **NPC模块**： 输入端口为PC, Jump, Branch, zero, instruction(为方便调用，将其分部分输入),busA，clk,输出端口为NPC；
3. **im模块**： 输入端口为PC[11:2],输出端口为instruction;
4. **RegFile模块**： 输入端口为PC, alure, instruction(为方便调用，将其分部分输入), busW, RegWr, RegDst, reset, clk,输出端口为busA, busB;
5. **exterder模块**：输入端口为：instruction[15:0],ExtOp,输出端口为Exout；
6. **ALU模块**：输入端口为：ALUctr, busA, busB与Exout选择后的结果data2，Exout, instruction[10:6]，输出端口为alure,zero；
7. **dm模块**：输入端口为：instruction[31:26],alure, busB, clk, MemWr, 输出端口为：dout。

综合以上模块，根据对应控制信号进行操作。因R,I,J类型的指令都存在这样的情况：同种类型的指令之间存在路径不同的现象，所以按类型指令给出总体路径不具备很好的代表性，若要综述同一类型的指令之间的路径，要所有同一类型的指令都满足，则R,I,J三种类型的综述路径几乎一致，因而此处不给出类型指令数据通路演示。

ALU

基本描述

根据不同的译码结果，对传入的数据信号进行不同操作的基本运算，并输出运算结果

模块接口

接口名	方向	描述
ALUctr	I	操作信号
A	I	操作数A
B	I	操作数B
shamt	I	移位指令移位数
result	O	运算结果
zero	O	0标志位

功能

1. 根据ALUctr计算并返回不同操作下的运算结果
2. 判断运算结果是否为0，返回zero信号以供后续判断使用

（附）ALUctr对应的各操作类型

ALUctr	操作名称	操作类型
00000	addu	无符号加
00001	subu	无符号减
00010	slt	小于则置位（有符号比较）
00011	and	按位与
00100	nor	按位或非
00101	or	按位或
00110	xor	按位异或
00111	sll	逻辑左移
01000	srl	逻辑右移
01001	sltu	小于则置位（无符号比较）
01100	sllv	逻辑左移（移位数来自寄存器）
01101	sra	算数右移
01110	srav	算数右移（移位数来自寄存器）
01111	srlv	逻辑右移（移位数来自寄存器）
10000	addiu	无符号立即数加法
10001	slti	小于则置位（立即数有符号比较）
10010	sltiu	小于则置位（立即数无符号比较）
10011	andi	立即数按位与
10100	ori	立即数按位或
10101	xori	立即数按位异或

代码实现

```
module ALU(ALUctr, A, B, shamt, result, zero);

input[4:0] ALUctr;
input[31:0] A, B;
input[4:0] shamt;

output[31:0] result;
output zero;

reg[31:0] result;

assign zero = (result == 32'b0); //0标志位

always @(*)
```

```
15  begin
16    case(ALUctr)
17      5'b00000: result <= A + B;//addu
18      5'b00001: result <= A - B;//subu
19      5'b00010://slt
20        //由于内置比较运算只能进行无符号比较运算 有符号比较需要另加判断
21        begin
22          if(A[31] ^ B[31]) //如果符号位不同 显然存在大小关系
23            result <= (A[31] == 1) ? 1 : 0; //若A符号为负 则显然A < B
24          else //如果符号位相同 可以用内置运算比较
25            //是负数比较的结果与补码视为无符号比较的结果不冲突
26            result <= (A < B) ? 1 : 0;
27        end
28      5'b00011: result <= A & B;//and
29      5'b00100: result <= ~(A | B);//nor
30      5'b00101: result <= A | B;//or
31      5'b00110: result <= A ^ B;//xor
32      5'b00111: result <= B << shamt;//sll 逻辑左移
33      5'b01000: result <= B >> shamt;//srl 逻辑右移
34      5'b01001: result <= (A < B) ? 1 : 0;//sltu 可直接使用内置运算
35      //5'b01010:jalr
36      //5'b01011:jr
37      //以上两条指令在alu阶段没有运算操作 即使涉及运算操作也放在寄存器组中进行了
38      5'b01100: result <= B << A;//sllv 逻辑左移 左移量由寄存器提供
39      //以下两个操作中 signed函数用来对操作数进行符号扩位 >>>是算术右移运算符 这两个
步骤都是为了保证算数右移的正确性
40      5'b01101: result <= $signed(B) >>> shamt;//sra 算数右移
41      5'b01110: result <= $signed(B) >>> A;//srav //算数右移 右移量由寄存器提
供
42      5'b01111: result <= B >> A;//srlv //逻辑右移 右移量由寄存器提供
43      5'b10000: result <= A + B;//addiu sign extension 无符号立即数加法 符号
扩展
44      //slti操作中传入的B已经是经过符号扩展之后的了
45      5'b10001://slti 立即数无符号比较 比较逻辑类似于前面的slt
46        begin
47          if(A[31] ^ B[31])
48            result <= (A[31] == 1) ? 1 : 0;
49          else
50            result <= (A < B) ? 1 : 0;
51        end
52      5'b10010: result <= (A < B) ? 1 : 0;//sltiu 立即数无符号比较
53      5'b10011: result <= A & B;//andi
```

```
        5'b10100: result <= A | B;//ori
        5'b10101: result <= A ^ B;//xori
      endcase
    end

    endmodule
```

im_4k

基本描述

从 `code.txt` 中读取并存储指令，每一周期根据地址返回响应指令

模块接口

接口名	方向	描述
addr	I	指令地址
dout	O	指令输出

功能

1. 初始化时从 `code.txt` 中读取指令 并将其按序存储在 `im` 指令存储器中
2. 每一周期根据传入的addr读取指令，通过 `dout` 接口连接到其他模块输入中

代码实现

```
module im_4k(addr, dout);

input[11:2] addr; //这里只采用[11:2]位可以避免处理PC初始地址0000_3000和im中存储第一条指令的地址0000_0000不匹配的问题
output[31:0] dout;

reg[31:0] im[1023:0];

initial
begin
    $readmemh("code.txt", im);
end

assign  dout = im[addr]; //设计成了组合逻辑电路 而非时序

endmodule
```

PC

基本描述

用于在 `reset` 信号有效时对PC进行初始化，并在每一时钟周期开始时对 `PC` 赋值

模块接口

接口名	方向	描述
NPC	I	下条指令的地址
clk	I	时钟周期
reset	I	复位信号
PC	O	执行指令地址

功能

1. 复位信号有效时将PC异步复位为 `0000_3000`
2. 复位信号无效时，在 `clk` 下降沿将 `NPC` 的值赋给 `PC`

代码实现

```
module PC(NPC, clk, reset, PC);

input[31:2] NPC;
input clk,reset;
output[31:2] PC;

reg[31:2] PC;

initial
begin
  PC = 0;
end

always @(negedge clk or posedge reset)
begin
  if(reset)
    PC <= 30'h0000_3000; //reset高电平时发生异步复位
  else
    PC <= NPC; //若复位信号无效 则在时钟下降沿将NPC赋值给PC
end
```

```
21
22  endmodule
```

NPC

基本描述

根据当前 `PC` 和各控制信号计算并返回下条执行指令的地址

模块接口

接口名	方向	描述
PC	I	执行指令地址
zero	I	0信号
branch	I	有条件跳转信号
jump	I	无条件跳转信号
op	I	指令操作码
func	I	R型指令操作码
imm16	I	I型指令立即数
target	I	J型指令直接地址
busA	I	寄存器组A输出口
rs	I	R型指令寄存器组地址rs
rt	I	R型指令寄存器组地址rt
rd	I	R型指令寄存器组地址rd
shamt	I	移位指令移位数
NPC	O	下条指令地址

功能

1. 初始化时将NPC置位0
2. 各输入信号发生变动时，计算下地址，并将下地址赋给NPC作为输出，传入PC寄存器

代码实现

```
1  module NPC(PC, zero, branch, op, imm16, target, jump, NPC, busA, rs, rt,
   rd, shamt, func);
2
3  input[31:2] PC;
4  input zero, branch, jump;
5  input[5:0] op, func;
```

```
6   input[15:0] imm16;
7   input[25:0] target;
8   input[31:0] busA;
9   input[4:0]  rs, rt, rd, shamt;
10  output[31:2] NPC;
11
12  reg[31:2] NPC;
13  wire[31:2] B_NPC = {{14{imm16[15]}}, imm16[15:0]} + PC; //条件跳转 将立即
    数进行符号扩展后与PC相加
14  wire[31:2] J_NPC = {PC[31:28], target}; //无条件跳转 target和PC组合成跳转指
    令
15  wire[31:2] N_NPC = PC + 1; //不发生跳转时正常的下一条指令
16
17  initial
18  begin
19    NPC = 0;
20  end
21
22  always @(*)
23  begin
24  //根据op的不同来执行不同的操作
25  case(op)
26    6'b000100: NPC = (zero == 1 && branch == 1) ? B_NPC : N_NPC;//beq
27    6'b000101: NPC = (zero == 0 && branch == 1) ? B_NPC : N_NPC;//bne
28    6'b000001:
29    //注意这里的bgez和bltz都是有符号比较 可以通过符号位来判断与0的大小关系
30      begin
31        if(rt == 1 && branch == 1 && (busA == 0 || busA[31] == 0))//bgez
32          NPC = B_NPC;
33        else if(rt == 0 && branch == 1 && busA[31] == 1 && busA ≠
    0)//bltz 此处还要保证busA不为0是要排除-0的情况
34          NPC = B_NPC;
35        else
36          NPC = N_NPC;
37      end
38    6'b000111: NPC = (branch == 1 && busA[31] == 0 && busA ≠ 0) ? B_NPC :
    N_NPC;//bgtz
39    6'b000110: NPC = (branch == 1 && (busA[31] == 1 || busA == 0)) ? B_NPC
    : N_NPC;//blez
40    6'b000010: NPC = (jump == 1) ? J_NPC : N_NPC;//j
41    6'b000011: NPC = (jump == 1) ? J_NPC : N_NPC;//jal 该指令额外的操作并不在
    NPC中进行
```

```
    6'b000000:
      begin
        if(rt == 0 && imm16 == 16'b1111100000001001)
          NPC = busA[31:2];//jalr 同理 额外操作不在NPC中进行
        else if(rt == 0 && imm16 == 16'b0000000000001000)
          NPC = busA[31:2];//jr
        else
          NPC = N_NPC;
      end
    default: NPC = N_NPC;
  endcase
  end

  endmodule
```

mux2

基本描述

二路选择器 根据选择信号选通不同数据

模块接口

接口名	方向	描述
a	I	数据输入a
b	I	数据输入b
s	I	选择信号
y	O	数据输出

功能

根据选通信号 `s` 选通数据 `s == 0` 时选通 `a` `s == 1` 时选通 `b`

代码实现

```
module mux2(a, b, s, y);

input s;              //选择信号
input[31:0] a,b;      //两个数据输入
output reg[31:0] y;   //输出

always @(*)
begin
    y <= (s == 0) ? a : b;
end

endmodule
```

extender

基本描述

位扩展器 根据传入的控制信号进行不同类型的扩展操作

模块接口

接口名	方向	描述
imm16	I	输入立即数
Extop	I	扩展指令操作码
Exout	O	扩展输出

功能

根据指令码 `Extop` 对立即数进行0扩展、符号扩展或低位0扩展（用于lui指令）

`Extop` 与操作的关系可见代码

代码实现

```
module extender(imm16, Extop, Exout);

input[15:0] imm16;
input[1:0]  Extop;
output[31:0] Exout;

reg[31:0] Exout;
```

```
8
9   always @(*)
10  begin
11    if(Extop == 2'b00) //0扩展
12      Exout <= {16'b0, imm16};
13    else if(Extop == 2'b01) //符号扩展
14      Exout <= {{16{imm16[15]}}, imm16};
15    else if(Extop == 2'b10) //用于lui指令的高位赋值 低位0扩展
16      Exout <= {imm16, 16'b0};
17  end
18
19  endmodule
```

dm_4k

基本描述

数据存储器 对数据进行写入存储和读出

模块接口

接口名	方向	描述
op	I	指令操作码 用于sb指令的判断
addr	I	地址码
din	I	数据输入
dout	O	数据输出
clk	I	时钟信号
MemWr	I	写使能信号

功能

1. `MemWr` 信号为1时将 `din` 中的数据写入 `dm` 中
2. 将 `addr` 地址码指定的数据赋给 `dout` 连接输出到其他模块中

代码实现

```
1   module dm_4k(op, addr, din, dout, clk, MemWr);
2
3   input[5:0]  op;
4   input[31:0] addr;
5   input[31:0] din;
```

```
input   clk;
input   MemWr;
output[31:0]  dout;

reg[7:0] dm[4095:0]; //采取以字节位单位存储数据才是最合理的

integer i;

always @(negedge clk)
begin
if(MemWr)
begin
    if(op == 6'b101000)   //sb
        dm[addr] <= din[7:0];
    else {dm[addr + 3], dm[addr + 2], dm[addr + 1], dm[addr]} <= din;
end
end

assign dout = {dm[addr + 3], dm[addr + 2], dm[addr + 1], dm[addr]};

endmodule
```

RegFile

基本描述

32 * 32bit寄存器组 既可向指定寄存器写入数据 也可从指定寄存器中读取数据

模块接口

接口名	方向	描述
op	I	指令操作码 此处用以区分一般指令和涉及跳转与字节装载的特殊指令
PC	I	执行指令地址
rs	I	寄存器组地址rs
rt	I	寄存器组地址rt
rd	I	寄存器组地址rd
shamt	I	移位数 此处用以辅助判断jalr
func	I	R型指令操作码 此处用以辅助判断jalr
data	I	输入数据
RegWr	I	写使能信号
RegDst	I	地址选择信号 用以选择写入的目标寄存器是rd还是rt
clk	I	时钟信号
reset	I	复位信号
ra	O	输出口a 保存$rs的值
rb	O	输出口b 保存$rt的值

功能

1. 在 `RegWr` 有效时，利用操作码与控制信号进行判断，将不同数据写入到指定的寄存器中
2. 将 `$rs` 和 `$rt` 寄存器的值赋给 `$ra` 和 `$rb` 并将输出连接到其他模块以供后续执行

代码实现

```
1  module RegFile(op, PC, rs, rt, rd, shamt, func, data, RegWr, RegDst, ra,
   rb, clk, reset);
2
3  input[5:0]   op;
4  input[31:2] PC;
5  input[4:0]  rs,rt,rd,shamt;
6  input[5:0]  func;
7  input[31:0] data;
8  input   RegWr;
9  input   RegDst;
10 input     clk;
11 input   reset;
12 output[31:0]  ra,rb;
13
14 reg[31:0] rgs[31:0];
15 integer   i;
16
```

```
always @(negedge clk or posedge reset)
begin
  if(reset) //复位 所有寄存器初始化为0
  begin
    for(i = 0; i < 32; i = i + 1)
      rgs[i] <= 32'b0;
  end

  if(RegWr)
  begin
    case(op)
      6'b100000: //lb 装载字节 作符号扩展 此处alure是计算出的内存取值地址 根据除4的余数关系确定装载的字节是4个字节中的哪一个字
        begin
          rgs[rt] <= {{24{data[7]}}, data[7:0]};
        end
      6'b100100: //lbu 装载字节 作0扩展 此处alure是计算出的内存取值地址 根据与4的倍数关系确定装载的字节是4个字节中的哪一个字
        begin
          rgs[rt] <= {{24'b0}, data[7:0]};
        end
      6'b000011: rgs[31] <= {(PC+30'd1), 2'b00}; //jal 在跳转同时还要将PC + 4并赋值给31号寄存器
      6'b001111: rgs[rt] <= {rd, shamt, func, 16'b0}; //lui 将立即数载入高位 末尾进行0扩展 这里的{rd, shamt, func}实际上是拼凑出了I型指令中的imm
      default:
        begin
          //jalr指令无法通过op判断 所以在这里另加判断
          if(rt == 0 && rd == 5'b11111 && shamt == 0 && func == 6'b001001)
            rgs[31] <= {(PC + 30'd1),2'b00}; //jalr 在跳转同时还要将PC + 4并赋值给31号寄存器
          //这里的操作可以省掉一个mux
          else if(RegDst == 1)  rgs[rd] <= data;
          else if(RegDst == 0)  rgs[rt] <= data;
        end
    endcase
  end
end

//1. 这里的操作是为了保证0号寄存器的值始终为0
//2. 当输入地址的值为0时 输出的值永远是0 而不是寄存器中的值
```

```
53    //3. 此处设计的是组合逻辑电路
54    assign ra = (rs == 0) ? rgs[rs] : 0;
55    assign rb = (rt == 0) ? rgs[rt] : 0;
56
57    endmodule
```

2. 控制器

conrtol

基本描述

控制器模块 接收指令的 `op` 和 `func` 域信号 对指令译码 产生该指令对应的控制信号

模块接口

接口名	方向	描述
op	I	指令操作码
func	I	R型指令辅助操作码
RegDst	O	目标寄存器选择信号
RegWr	O	寄存器组写使能信号
ALUSrc	O	ALU B输入口选择信号
MemWr	O	存续器写使能信号
MemtoReg	O	寄存器组写数据选择信号
ExtOp	O	扩展器操作码
ALUctr	O	ALU操作码
Branch	O	有条件跳转指令信号
Jump	O	无条件跳转指令信号

功能

1. 当 `op` 不为0 即指令不为R型指令时 根据 `op` 对指令进行译码 将各信号输送给对应的控制信号
2. 当 `op` 的值为0 即指令为R型指令时 根据 `func` 对指令进行译码 将各信号书送给对应的控制信号

代码实现

```
1   module ctrl(op, func, RegDst, RegWr, ALUSrc, MemWr, MemtoReg, ExtOp,
    ALUctr, Branch, Jump);
2
3   input[5:0]  op;
```

```
input[5:0]  func;
output    RegDst;
output    RegWr;
output    ALUSrc;
output    MemWr;
output    MemtoReg;
output[1:0] ExtOp;
output[4:0] ALUctr;
output    Branch;
output    Jump;

reg[13:0] controls;

assign {RegDst, RegWr, ALUSrc, MemWr, MemtoReg, ExtOp, ALUctr, Branch,
Jump} = controls;

always @(*)
case(op)
  6'b000000:    //R
    begin
      case(func)
        6'b100001: controls <= 14'b11000000000000;//addu
        6'b100011: controls <= 14'b11000000000100;//subu
        6'b101010: controls <= 14'b11000000001000;//slt
        6'b100100: controls <= 14'b11000000001100;//and
        6'b100111: controls <= 14'b11000000010000;//nor
        6'b100101: controls <= 14'b11000000010100;//or
        6'b100110: controls <= 14'b11000000011000;//xor
        6'b000000: controls <= 14'b11000000011100;//sll
        6'b000010: controls <= 14'b11000000100000;//srl
        6'b101011: controls <= 14'b11000000100100;//sltu
        6'b001001: controls <= 14'b11000000000000;//jalr
        6'b001000: controls <= 14'b10000000000000;//jr
        6'b000100: controls <= 14'b11000000110000;//sllv
        6'b000011: controls <= 14'b11000000110100;//sra
        6'b000111: controls <= 14'b11000000111000;//srav
        6'b000110: controls <= 14'b11000000111100;//srlv
      endcase
    end
  6'b001001: controls <= 14'b01100011000000;//addiu
  6'b000100: controls <= 14'b00000000000110;//beq
  6'b000101: controls <= 14'b00000000000110;//bne
```

```
    6'b100011: controls <= 14'b01101011000000;//lw
    6'b101011: controls <= 14'b00110011000000;//sw
    6'b001111: controls <= 14'b01100100000000;//lui
    6'b001010: controls <= 14'b01100011000100;//slti
    6'b001011: controls <= 14'b01100011001000;//sltiu
    6'b000001: controls <= 14'b00000000000010;//bgez,bltz

    6'b000111: controls <= 14'b00000000000010;//bgtz
    6'b000110: controls <= 14'b00000000000010;//blez
    6'b100000: controls <= 14'b01101011000000;//lb
    6'b100100: controls <= 14'b01101011000000;//lbu
    6'b101000: controls <= 14'b00110011000000;//sb
    6'b001100: controls <= 14'b01100001001100;//andi
    6'b001101: controls <= 14'b01100001010000;//ori
    6'b001110: controls <= 14'b01100001010100;//xori


    6'b000010: controls <= 14'b00000000000001;//j
    6'b000011: controls <= 14'b01000000000001;//jal

    default: controls <= 14'b00000000000000;
  endcase
  endmodule
```

（附）控制信号真值表

	RegDst	RegWr	ALUSrc	MemWr	MemtoReg	ExtOp	ALUctr	Branch	Jump	R/I/J	31-26(op)	25-21	20-16	15_11	10_6	5-0(func)
addu	1	1	0	0	0	xx	0 0000	0	0	R	0 00000	rs	rt	rd	0 0000	100001
subu	1	1	0	0	0	xx	0 0001	0	0	R	0 00000	rs	rt	rd	0 0000	100011
slt	1	1	0	0	0	xx	0 0010	0	0	R	0 00000	rs	rt	rd	0 0000	101010
and	1	1	0	0	0	xx	0 0011	0	0	R	0 00000	rs	rt	rd	0 0000	100100
nor	1	1	0	0	0	xx	0 0100	0	0	R	0 00000	rs	rt	rd	0 0000	100111
or	1	1	0	0	0	xx	0 0101	0	0	R	0 00000	rs	rt	rd	0 0000	100101
xor	1	1	0	0	0	xx	0 0110	0	0	R	0 00000	rs	rt	rd	0 0000	100110
sll	1	1	0	0	0	xx	0 0111	0	0	R	0 00000	0 0000	rt	rd	shf	0 00000
srl	1	1	0	0	0	xx	0 1000	0	0	R	0 00000	0 0000	rt	rd	shf	0 00010
addiu	0	1	1	0	0	0 1	1 0000	0	0	I	0 01001	rs	rt	imm	imm	imm
beq	x	0	0	0	x	xx	0 0001	1	0	I	0 00100	rs	rt	offset	offset	offset
bne	x	0	0	0	x	xx	0 0001	1	0	I	0 00101	rs	rt	offset	offset	offset
lw	0	1	1	0	1	0 1	1 0000	0	0	I	1 00011	base	rt	offset	offset	offset
sw	x	0	1	1	x	0 1	1 0000	0	0	I	1 01011	base	rt	offset	offset	offset
lui	0	1	1	0	0	1 0	x	0	0	I	0 01111	0 0000	rt	imm	imm	imm
j	x	0	x	0	x	x x	x	0	1	J	0 00010	target	target	target	target	target
sltu	1	1	0	0	0	xx	0 1001	0	0	R	0 00000	rs	rt	rd	0 0000	101011
jalr	1	1	0	0	0	xx	x	0	0	R	0 00000	rs	0 0000	1 1111	0 0000	0 01001
jr	1	0	0	0	0	xx	x	0	0	R	0 00000	rs	0 0000	0 0000	0 0000	0 01000
sllv	1	1	0	0	0	xx	0 1100	0	0	R	0 00000	rs	rt	rd	0 0000	0 00100
sra	1	1	0	0	0	xx	0 1101	0	0	R	0 00000	0 0000	rt	rd	shf	0 00011
srav	1	1	0	0	0	xx	0 1110	0	0	R	0 00000	rs	rt	rd	0 0000	0 00111
srlv	1	1	0	0	0	xx	0 1111	0	0	R	0 00000	rs	rt	rd	0 0000	0 00110
slti	0	1	1	0	0	0 1	1 0001	0	0	I	0 01010	rs	rt	imm	imm	imm
sltiu	0	1	1	0	0	0 1	1 0010	0	0	I	0 01011	rs	rt	imm	imm	imm
bgez	x	0	0	0	x	xx	x	1	0	I	0 00001	rs	0 0001	offset	offset	offset
bgtz	x	0	0	0	x	xx	x	1	0	I	0 00111	rs	0 0000	offset	offset	offset
blez	x	0	0	0	x	xx	x	1	0	I	0 00110	rs	0 0000	offset	offset	offset
bltz	x	0	0	0	x	xx	x	1	0	I	0 00001	rs	0 0000	offset	offset	offset
lb	0	1	1	0	1	0 1	1 0000	0	0	I	1 00000	base	rt	offset	offset	offset
lbu	0	1	1	0	1	0 1	1 0000	0	0	I	1 00100	base	rt	offset	offset	offset
sb	0	0	1	1	0	0 1	1 0000	0	0	I	1 01000	base	rt	offset	offset	offset
andi	0	1	1	0	0	0 0	1 0011	0	0	I	0 01100	rs	rt	imm	imm	imm
ori	0	1	1	0	0	0 0	1 0100	0	0	I	0 01101	rs	rt	imm	imm	imm
xori	0	1	1	0	0	0 0	1 0101	0	0	I	0 01110	rs	rt	imm	imm	imm
jal	x	1	x	0	0	xx	x	0	1	J	0 00011	target	target	target	target	target

3. 整体模块

mips

基本描述

整合 `datapath` 与 `control` 模块 形成可根据预设指令自动执行的单周期cpu

模块接口

接口名	方向	描述
clk	I	时钟信号
rst	I	异步复位信号

功能

由 `datapath` 模块根据 `PC` 读取预设指令 传入 `control` 模块生成控制信号之后执行当前指令 并由 `PC` 读取下一条指令重复读取 译码 执行过程 从而实现对预设指令的自动执行

代码实现

```
`include"ctrl.v"
`include"datapath.v"

module mips(clk, rst);

input clk;
input rst;

wire[5:0]   op;
wire[4:0] rs,rt,rd,shamt;
wire[5:0] func;
wire    RegDst;
wire    RegWr;
wire    ALUSrc;
wire    MemWr;
wire    MemtoReg;
wire[1:0] ExtOp;
wire[4:0] ALUctr;
wire    Branch;
wire    Jump;
wire[31:0]  instruction;
```

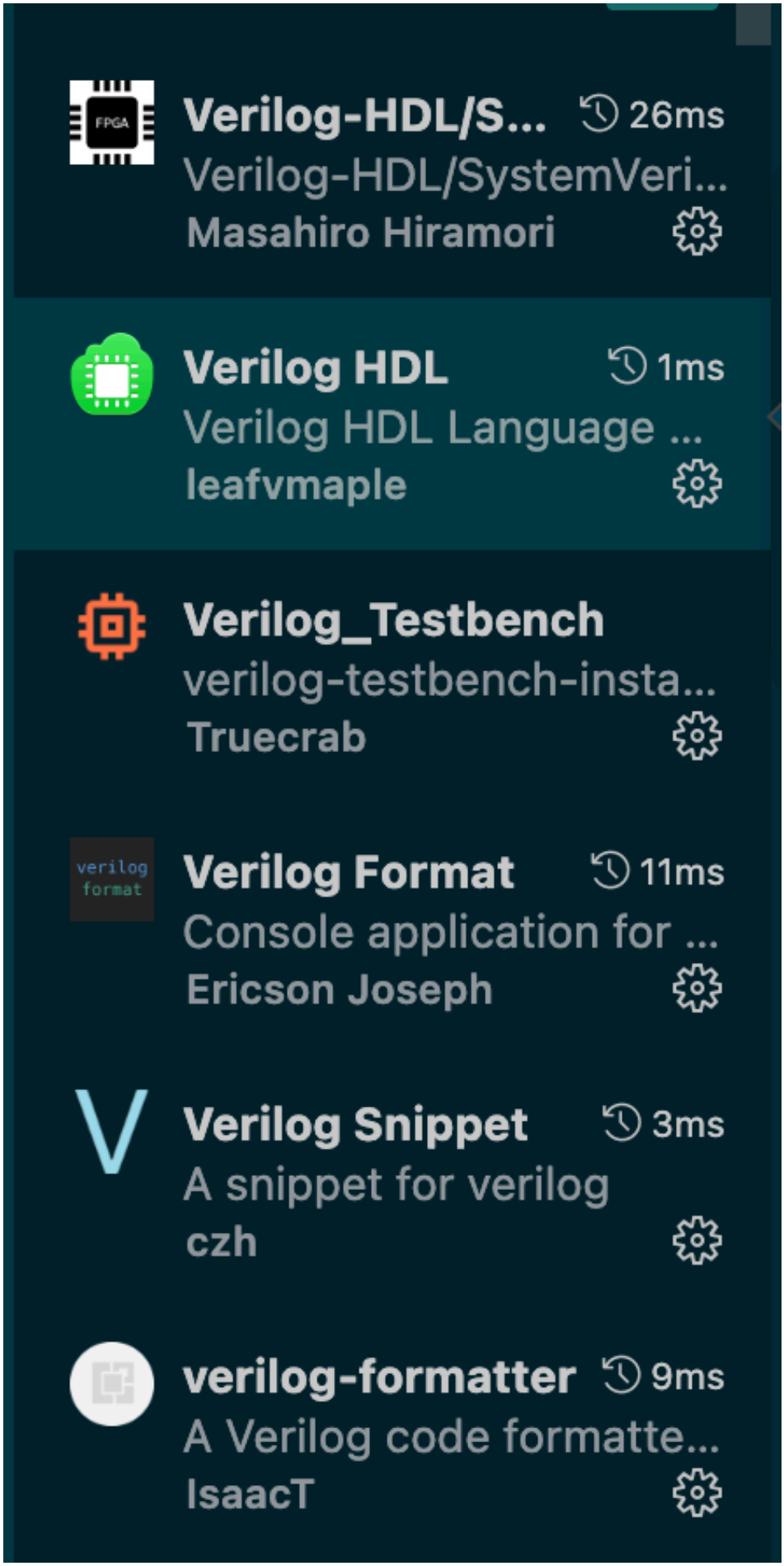
```
22
23    assign    op = instruction[31:26];
24    assign    func = instruction[5:0];
25    assign    rs = instruction[25:21];
26    assign    rt = instruction[20:16];
27    assign    rd = instruction[15:11];
28    assign    shamt = instruction[10:6];
29
30    ctrl CTRL(op, func, RegDst, RegWr, ALUSrc, MemWr, MemtoReg, ExtOp,
      ALUctr, Branch, Jump);
31    datapath DPU(RegDst, RegWr, ALUSrc, MemWr, MemtoReg, ExtOp, ALUctr,
      Branch, Jump, clk, rst, instruction);
32
33    endmodule
```

三. 测试代码及方法

1. 环境配置

由于我的电脑是MacOS系统 无法安装modelsim进行仿真 故根据网上共享的配置方案 用vscode + 插件 + gtkwave搭配的方法对代码进行仿真与运行 具体来说 执行如下：

- 在vscode中安装如下插件


Verilog-HDL/S... 26ms
Verilog-HDL/SystemVeri...
Masahiro Hiramori
Verilog HDL 1ms
Verilog HDL Language ...
leafvmaple
Verilog_Testbench
verilog-testbench-insta...
Truecrab
verilog format
Verilog Format 11ms
Console application for ...
Ericson Joseph
Verilog Snippet 3ms
A snippet for verilog
czh
verilog-formatter 9ms
A Verilog code formatte...
IsaacT

> 安装的插件列表

- 通过Homebrew 在终端中安装如下软件

  1. 安装Iverilog

  ```
  % brew install icarus-verilog
  ```

  2. 安装verilator

  ```
  % brew install verilator
  ```

  3. 安装gtkwave（波形图展示工具）

  ```
  % brew cask install xquartz
  % brew cask insatll gtkwave
  ```

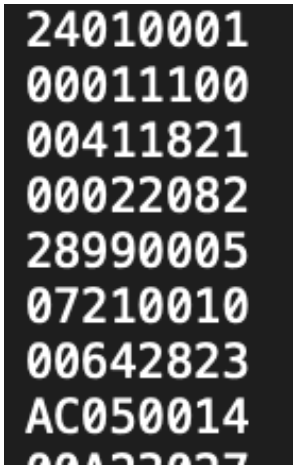
> 通过Homebrew安装的软件列表

上述配置完成后 可以在vscode中对 `Verilog` 代码进行编译运行 并使用gtkwave或vscode对波形进行仿真

2. 测试代码

来自老师发送的测试代码文件 `《36条指令-对应的汇编代码+指令字等-红色字新加说明》`

将测试代码的以十六进制存储在 `code.txt` 中效果如下：




00A23027
00C33825
00E64026
AC08001C
11030002
00C7482A
24010008
8C2A0014
15450004
00415824
AC2B001C
AC240010
0C000019
3C0C000C
004CD007
003AD804
0360F809
A07A0005
0063682B
1DA00003
00867004
000E7883
002F8006
1A000008
002F8007
240B008C
06000006
8D5C0003
179D0007
A0AF0008
80B20008
90B30008
2DF8FFFF
0185E825
01600008
31F4FFFF
35E5FFFF

```
39F6FFFF
08000000
```

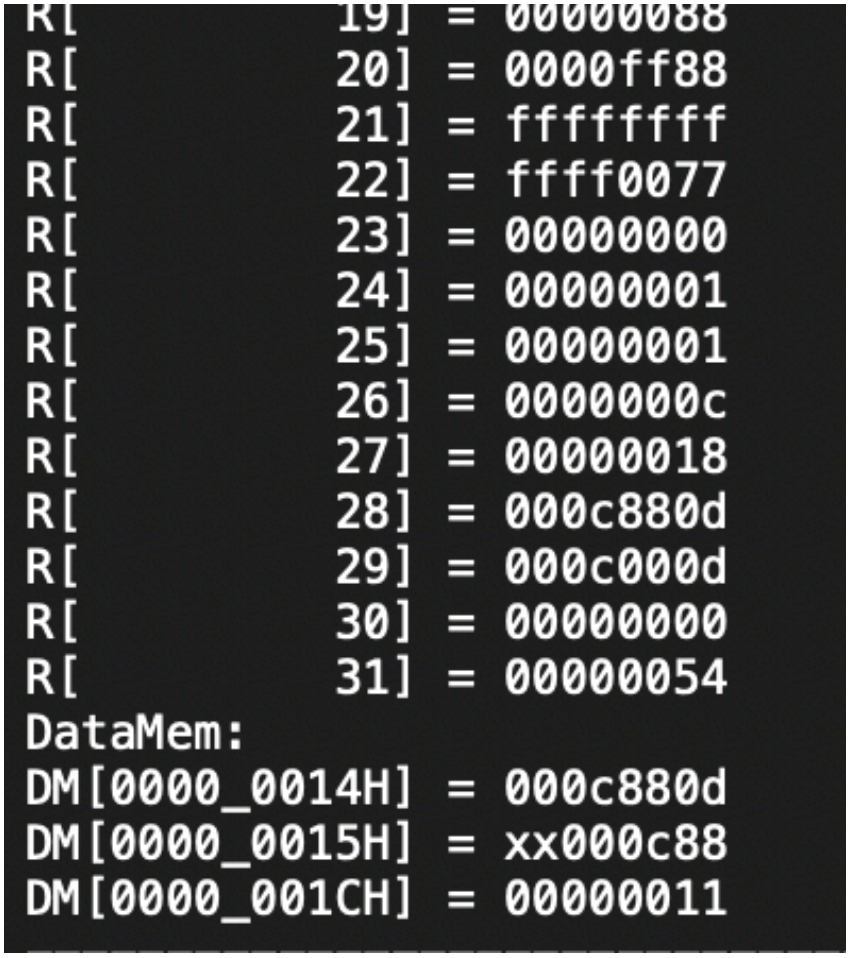
> 测试代码（十六进制）

3. 测试结果

由于 `gtkwave` 虽能仿真波形 但无法监测到模块内部定义的变量信号（如寄存器组中每个寄存器的值 数据存储器中每块内存的值）所以采用每一周期将寄存器值与数据存储器中的数据输出到 `out.txt` 文件的形式对数据进行侦测 从而判断指令执行的正确性

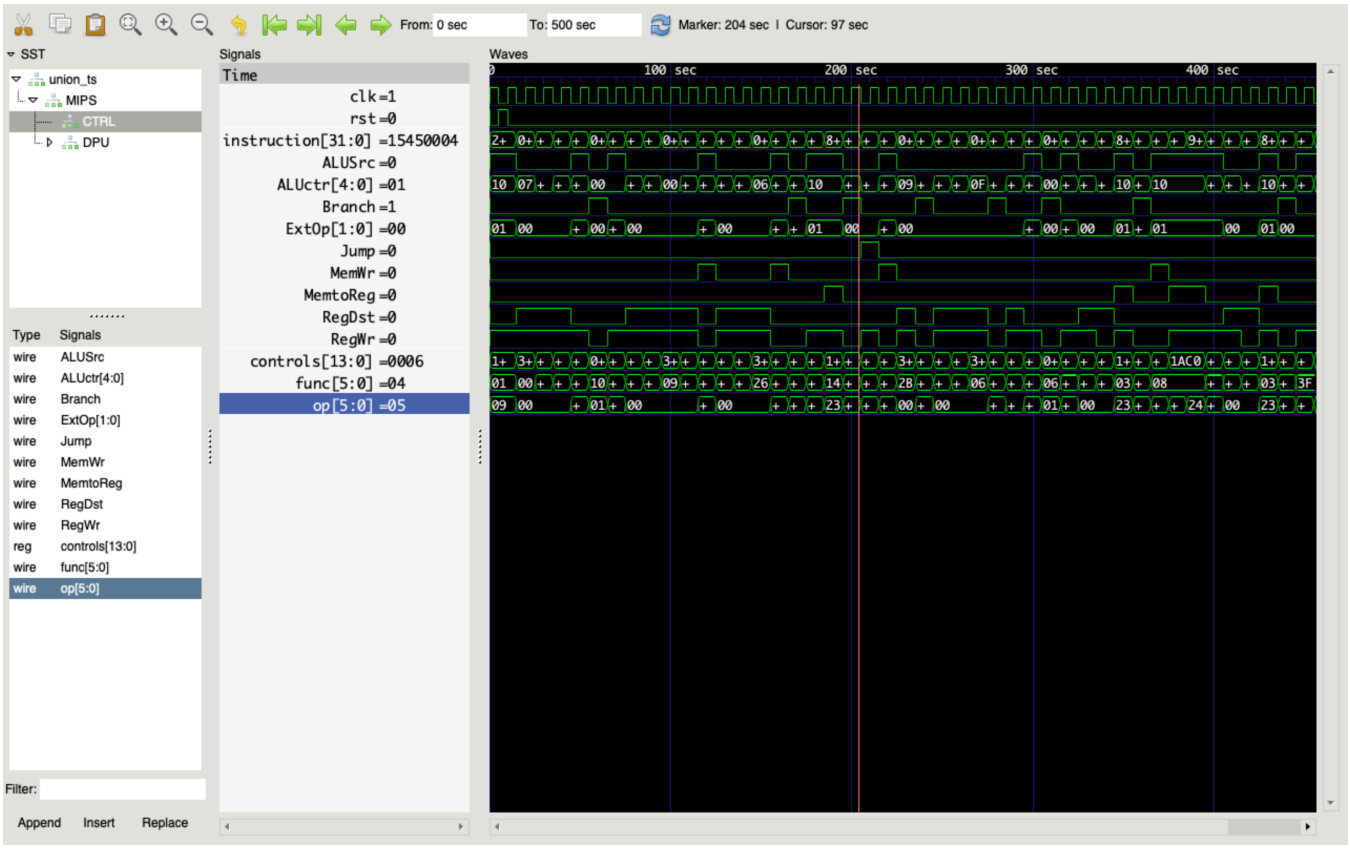
- 数据结果（存储在 `out.txt` 中）

```
clock [          50 pos]
instruction = 24010001
PC = 00000000
Registers:
R[           0] = 00000000
R[           1] = 00000008
R[           2] = 00000010
R[           3] = 00000011
R[           4] = 00000004
R[           5] = 0000000d
R[           6] = fffffffe2
R[           7] = ffffffff3
R[           8] = 00000011
R[           9] = 00000000
R[          10] = 00000011
R[          11] = 0000008c
R[          12] = 000c0000
R[          13] = 00000000
R[          14] = fffffe20
R[          15] = ffffff88
R[          16] = ffffffff
R[          17] = 00000000
R[          18] = ffffff88
```



> 此处展示的是最后一周期所有代码执行完毕之后的结果

- 波形仿真结果（存储在`single_cpu_wave.vcd`中）



> 由于数据结果另做处理 此处只展示控制信号的仿真结果

请老师检查时注意 由于数据存储器 `dm` 中的各数据没有初始化为0 所以在查看与数据存储器相关的波形时可能会出现红线 但并不是错误!

四. 问答与解决

- ***问题***:

  C 语言是一种弱类型程序设计语言。C 语言中不对计算结果溢出进行处理， 这意味着 C 语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如 果仅仅支持 C 语言，MIPS 指令的所有计算指令均可以忽略溢出。请说明为什么在忽略溢出的前提下，addi与addiu是等价的，add与addu是等价的。

- ***回答***:

  `《MIPS32® Architecture For II: The MIPS32® Instruction Set》` 中 对上述涉及的指令有如下介绍

  - add

Operation:

```
temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← temp
endif
```

Exceptions:

Integer Overflow

Programming Notes:

ADDU performs the same arithmetic operation but does not trap on overflow.

- addi

Operation:

```
temp ← (GPR[rs]31||GPR[rs]31..0) + sign_extend(immediate)
if temp32 ≠ temp31 then
    SignalException(IntegerOverflow)
else
    GPR[rt] ← temp
endif
```

Exceptions:

Integer Overflow

Programming Notes:

ADDIU performs the same arithmetic operation but does not trap on overflow.

- addu

Operation:

```
temp ← GPR[rs] + GPR[rt]
GPR[rd] ← temp
```

Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

- addiu

Operation:

```
temp ← GPR[rs] + sign_extend(immediate)
GPR[rt]← temp
```

Exceptions:

None

Programming Notes:

The term "unsigned" in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.

不难看出，addu除了不考虑溢出之外，其执行步骤与add一致，所以在不考虑溢出的情况下add与addu是等价的，同理可得在不考虑溢出的情况下addi与addiu是等价的

五. 总结与展望

写单周期CPU给我的感受是，提到verilog中的哪个语法，我都知道，提到数据通路，我也大致了解，但是真正开始着手写单周期CPU时，还是无从下手，不知道从哪开始写起，不知道怎么一步步的把部件结合起来。拿着书，对着ppt，琢磨了一两天，才有些头绪，刚准备动手，又抓耳挠腮，百思不得其解。后来重新梳理了一遍课本上的设计思路，大致理清了写部件的顺序：PC->NPC->IM->RegFile->extender->mux->ALU->DM->datapath->ctrl，写单个部件时，搞清楚每个部件的输入输出，暂时不用多担心部件与部件的互联。自此，我很快把一个个部件实现。

在写数据通路时，我深刻理解了verilog语言的并行性，也在脑海中对数据通路有了初步的建模。在写ctrl时，我借鉴了学姐的想法，书上用逻辑表达式的方法，这种方法虽然能够理解，但实际操作的时候需要输入的代码过于冗长，极大地增加了出错的可能性，而且如果要进行逻辑表达式化简的话，又要花不少时间，所以我建立了一个controls变量，将所有的控制信号全都整合成一个01串，并事先按控制信号的顺序列好对应的真值表，只需要将真值表中对应的数据一一赋给对应指令的信号就可以了，这样有效地节省了时间，并且极大地提高了正确率，同时，在调试的过程中，信号的显示也十分清晰，易于对照检查。

但是，就算这样，我在debug时仍旧遇到了极大的困难。光看书和ppt，我想当然的以为MIPS系统里我定义的寄存器组的0号寄存器的值就是0，没想到这些MIPS里的特征都是我们给他设置的，不是一开始就这样。还有对于PC信号的初值，我在刚开始写时也很疑惑应该设置成什么，设置成全零又运行不了，后来在舍友的指导下才知道了mars里的规定，一步步才完成了调试，最终运行成功。

我觉得自己是相对较早开始写单周期CPU的，但是由于中途陆陆续续的不停课考试，使得这个过程断断续续，没有在一个连续的时间内把它完成。

在这次实验中，我加深了对计算机重要组成部分CPU的理解，同时更深入地理解了verilog语言的特点，同时为我接下来写流水线CPU作了铺垫。

感谢老师的指导与讲解！