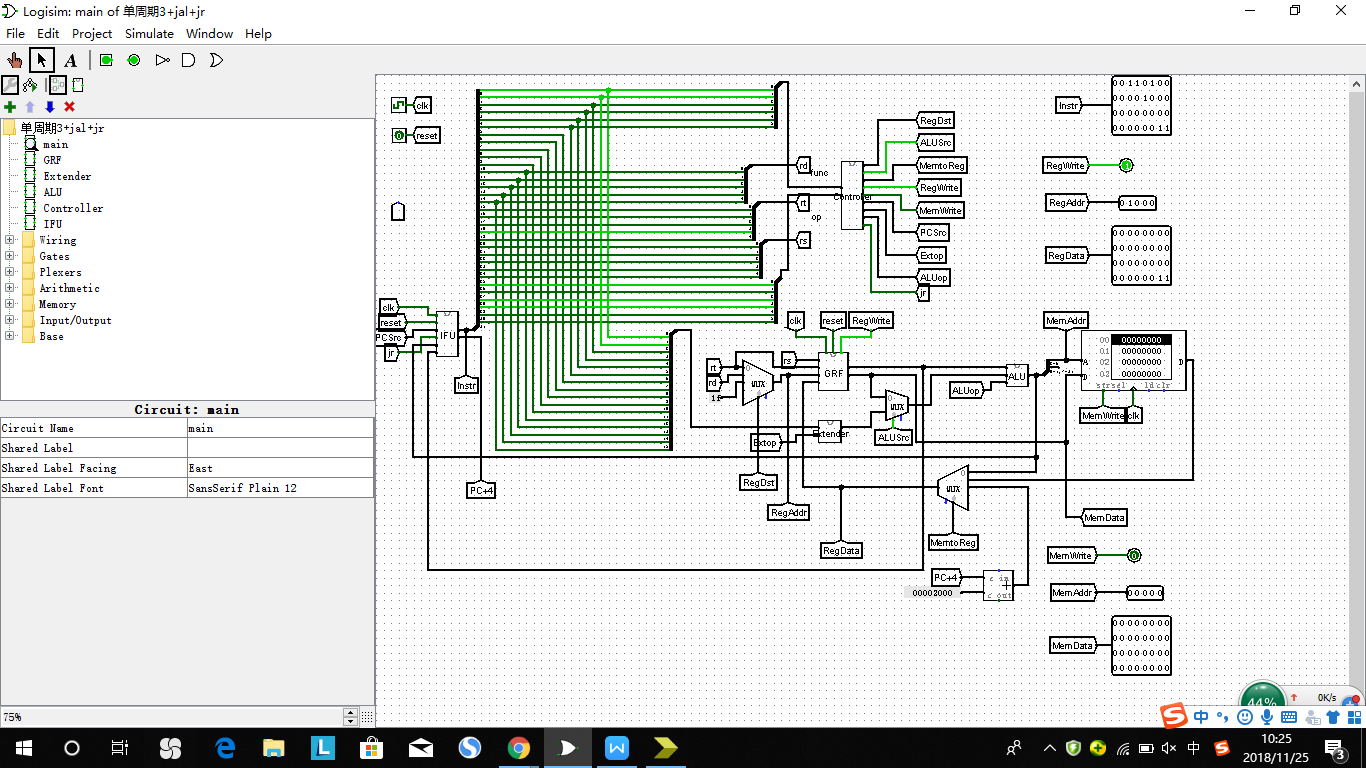
利用Verilog开发MIPS单周期处理器

一、整体结构：

控制器（Controller）、IFU（取指令单元）、GRF（通用寄存器组，也称为寄存器文件、寄存器堆）、ALU（算术逻辑单元）、DM（数据存储器）、EXT（位扩展器）。

处理器为32位处理器

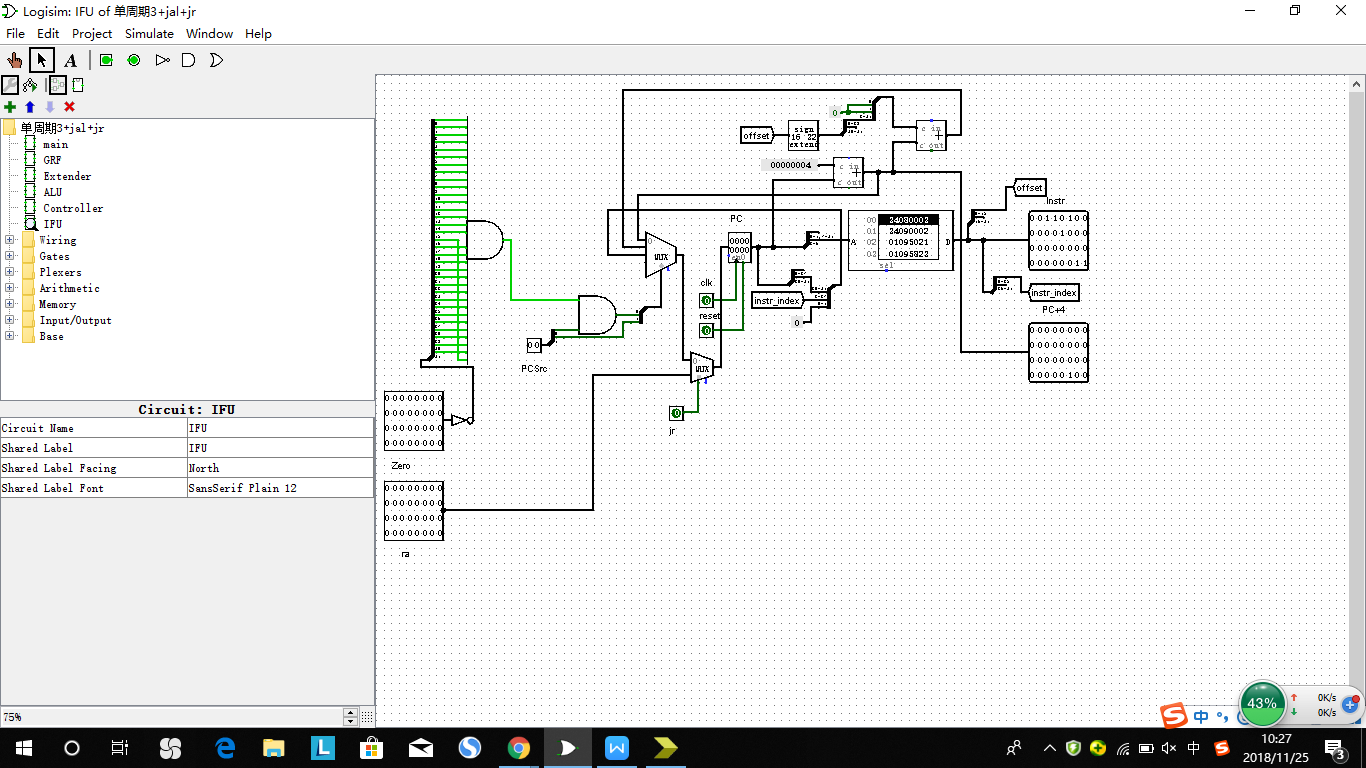
处理器应支持的指令集为：{addu,subu,ori,lw,sw,beq,lui,jal,jr,nop}。



1. 数据通路设计：

（一）模块规格撰写

1.IFU（取指令单元）：内部包括PC、IM、NPC



1）PC（程序计数器）

器件：32bit寄存器

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| NPC[31:0] | I | 下一条指令的地址 |
| clk | I | 时钟信号 |
| reset | I | 复位信号  1：有效  0：无效 |
| PC[31:0] | O | 当前指令地址 |

功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 复位 | 当复位信号有效时，PC被设置为起始地址0x00003000 |

2）IM（指令存储器）

IM容量为4KB（32bit×1024字）

因为ROM中储存了1024个地址，且IM实际地址宽度仅为10位，，从而将地址的低10位（2~11位）连接到ROM选择地址端口。

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| addr[31:0] | I | 当前指令的地址 |
| op[5:0] | O | 6位op信号 |
| func[5:0] | O | 6位func信号 |
| rs | O | rs寄存器编号 |
| rt | O | rt寄存器编号 |
| rd | O | rd寄存器编号 |
| imm[15:0] | O | 16位offset |
| imm26[25:0] | O | 26位instr\_index |

3）NPC（计算下一条指令）

器件：加法器、多路选择器

在处理器支持的指令集中，下一条指令的选择分为两种情况：

1. addu,subu,ori,lw,sw,lui,nop：PC=PC+4
2. beq：PC=(PC+4）+sign\_extend(imm16)<<2
3. jal：PC={PC[31:28]，instr\_index，2’b00}
4. jr：PC=GPR[rs]

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| PC[31:0] | I | 当前指令地址 |
| imm[15:0] | I | 16位offset |
| imm26[25:0] | I | 26位instr\_index |
| jr\_ra[31:0] | I | jr指令时，rs寄存器中的值 |
| PCSrc[1:0] | I | PC选择信号  01：当前指令为beq  10：当前指令为jal  00：其他指令 |
| jr | I | jr指令信号 |
| zero | I | ALU计算结果为0标志  0：计算结果为0  非0：计算结果非0 |
| NPC[31:0] | O | 下一条指令的地址 |
| PCplus4 | O | PC+4的值 |

功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 复位 | 当复位信号有效时，PC被设置为起始地址0x00000000 |
| 3 | 计算下一条指令地止 | PCSrc=00且jr=0时：PC=PC+4  PCSrc=01且jr=0时：PC=(PC+4）+sign\_extend(imm16)<<2  PCSrc=10且jr=0时：PC={PC[31:28]，instr\_index，2’b00}  jr=1时：PC=GPR[rs] |

2.GRF（通用寄存器组）：内部包括32个寄存器

具有写使能的寄存器实现，寄存器总数为 32 个

**0 号寄存器**的值始终保持为 0。其他寄存器初始值均为 0

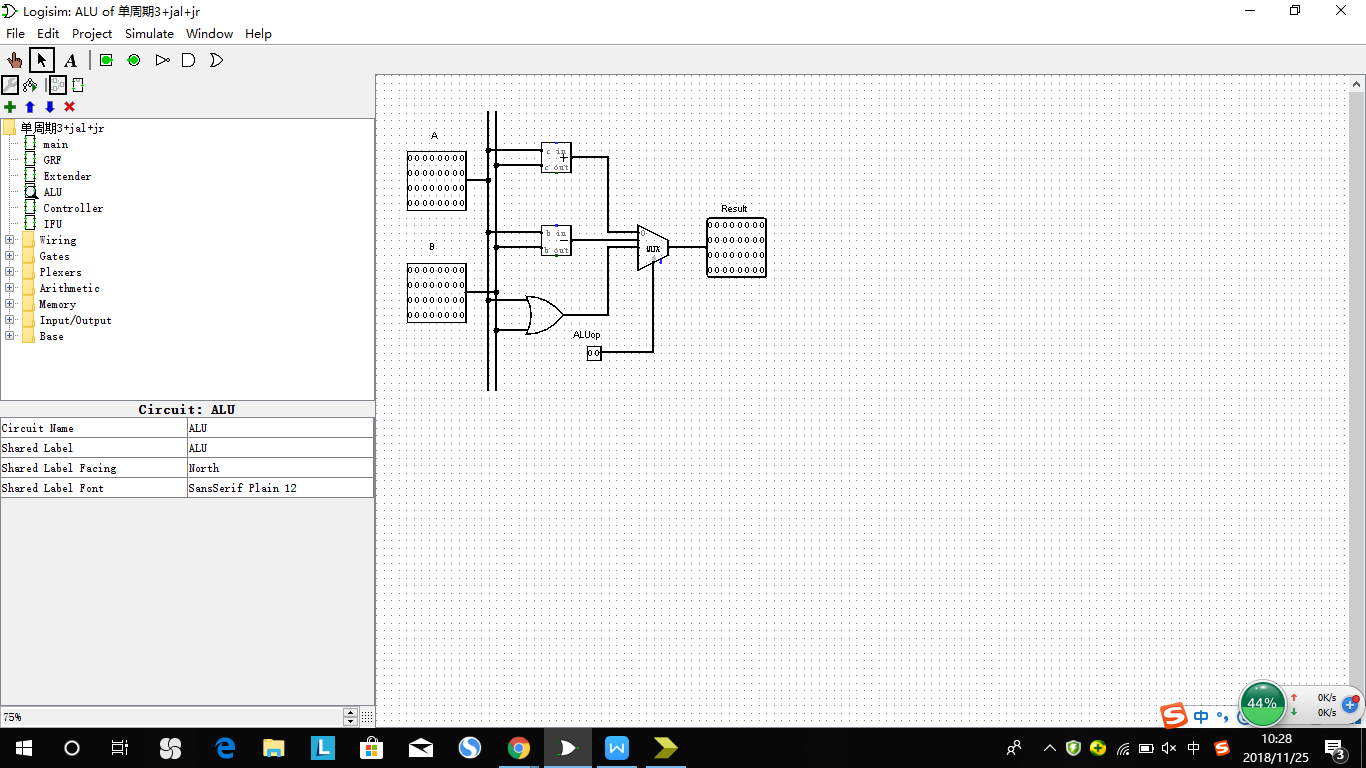
模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| WPC[31:0] | I | 相应指令存储地址 |
| clk | I | 时钟信号 |
| reset | I | 复位信号  1：有效  0：无效 |
| RegWrite | I | 读写控制信号  1：写操作  0：读操作 |
| Read\_register1[4:0] | I | 读寄存器1的地址 |
| Read\_register2[4:0] | I | 读寄存器2的地址 |
| Write\_register[4:0] | I | 写寄存器的地址 |
| Write\_data[31:0] | I | 向写寄存器中写入的值 |
| Read\_data1[31:0] | O | 32位输出1 |
| Read\_data2[31:0] | O | 32位输出2 |

功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 复位 | 当复位信号有效时，所有寄存器的值被设置为0x00000000 |
| 2 | 写寄存器 | 根据输入的写寄存器地址，把输入的数据写入写寄存器中 |
| 3 | 读寄存器 | 根据输入的读寄存器地址，将数据读出 |

1. ALU（算术逻辑单元）



提供32位加、减、或运算

可以不支持溢出

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| A[31:0] | I | ALU32位输入数据A |
| B[31:0] | I | ALU32位输入数据B |
| ALUop[1:0] | I | ALU功能选择信号  00:加法  01:减法  10:或运算 |
| Result[31:0] | O | 32位数据输出 |

功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 或 | A|B |
| 2 | 减 | A-B |
| 3 | 加 | A+B |

4.DM（数据存储器）

 DM容量为4KB（32bit×1024字）

起始地址：0x00003000

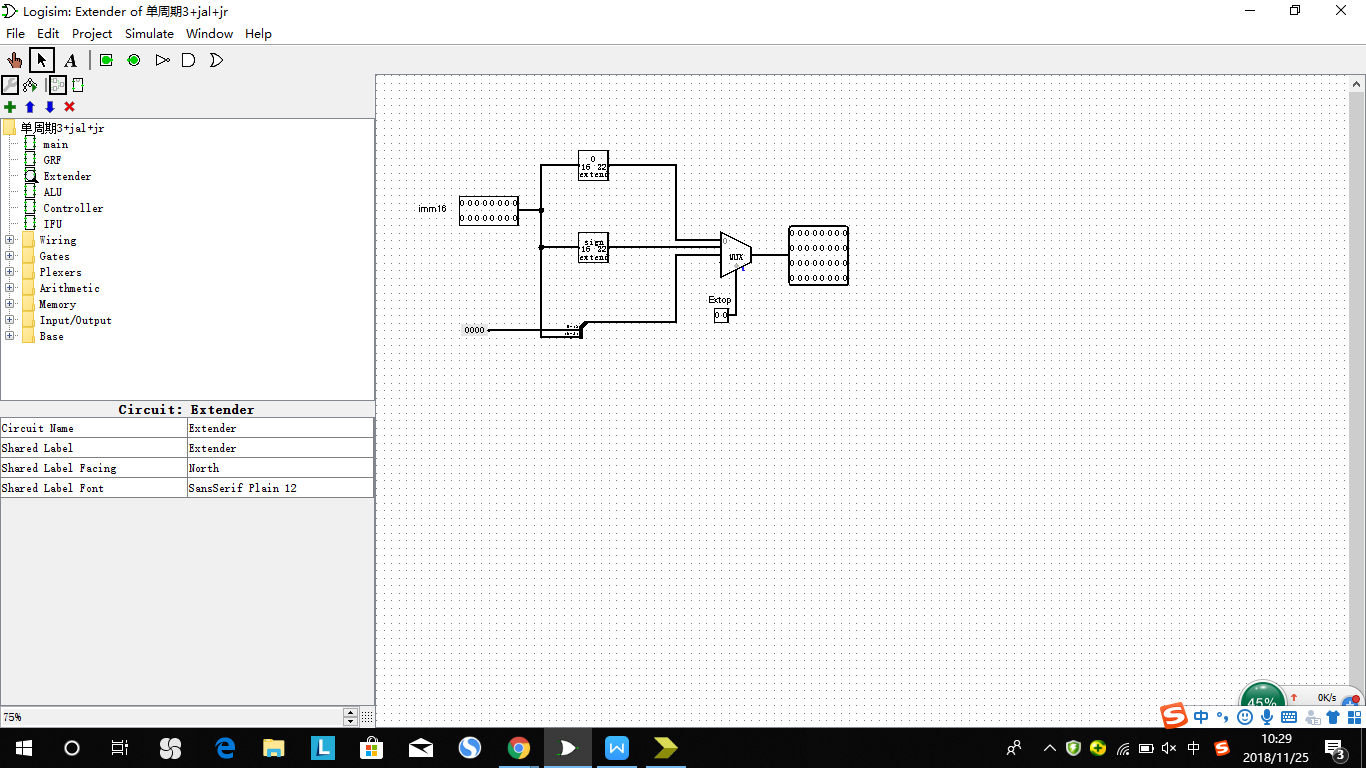
模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| pc[31:0] | I | 对应的pc |
| addr[31:0] | I | 对应指令的地址 |
| clk | I | 时钟信号 |
| reset | I | 复位信号  1：有效  0：无效 |
| MemData[31:0] | I | 写入数据的输入 |
| MemWrite | I | 读写控制信号  1：写操作 |
| out | O | 读取数据的输出 |

功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 复位 | 当复位信号有效时，所有数据被设置为0x00000000 |
| 2 | 写操作 | 根据输入的寄存器地址，把输入的数据写入 |
| 3 | 读操作 | 根据输入的寄存器地址，将其中的数据读出 |

1. EXT（位扩展器）：



使用logisim内置的Bit Extender

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| imm[15:0] | I | 16位imm数据输入 |
| Extop[1:0] | I | 位扩展选择信号  00:高位补0  01：高位符号扩展  10：低位补0 |
| after\_ext[31:0] | O | 位扩展后的32位输出 |

功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 高位补0 | 高16位补0 |
| 2 | 低位补0 | 低16位补0 |
| 3 | 符号扩展 | 若符号位为0，则高位补0  若符号位为1，则高位补1 |

6.控制器（Controller）

模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| func[5:0] | I | 6位function |
| op[5:0] | I | 6位op |
| RegDst[1:0] | O | 写寄存器地址控制 |
| ALUsrc | O | ALU的B操作数的选择控制 |
| MemtoReg[1:0] | O | 写寄存器的数据来源选择控制 |
| RegWrite | O | GRF读写控制信号 |
| MemWrite | O | DM写控制信号，写入GRF的数据选择 |
| PCsrc[1:0] | O | PC选择信号 |
| Extop[1:0] | O | 控制扩展方式 |
| ALUSrc[1:0] | O | 控制ALU进行相应运算 |
| jr | O | jr信号 |

1. mux（多路选择器）

1)mux2\_32

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| Control | I | 控制选择信号 |
| din0[31:0] | I | Control=0时的选择 |
| din1[31:0] | I | Control=1时的选择 |
| out[31:0] | O | 选择的输出 |

2)mux3\_5

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| Control[1:0] | I | 控制选择信号 |
| din0[4:0] | I | Control=00时的选择 |
| din1[4:0] | I | Control=01时的选择 |
| din2[4:0] | I | Control=10时的选择 |
| out[4:0] | O | 选择的输出 |

3)mux3\_5

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| Control[1:0] | I | 控制选择信号 |
| din0[31:0] | I | Control=00时的选择 |
| din1[31:0] | I | Control=01时的选择 |
| din2[31:0] | I | Control=10时的选择 |
| out[31:0] | O | 选择的输出 |

1. 思考题
2. 根据你的理解，在下面给出的DM的输入示例中，地址信号addr位数为什么是[11:2]而不是[9:0]？这个addr信号又是从哪里来的？



答：在DM中，内存是按字划分的。一个字是四字节，因此在进行DM中进行内存的存取时，应当将输入地址除以4（或者右移两位），作为实际地址，即取2~11位。

addr信号来自ALU的运算结果。

1. 在相应的部件中，reset的优先级比其他控制信号（不包括clk信号）都要高，且相应的设计都是同步复位。清零信号reset是针对哪些部件进行清零复位操作？这些部件为什么需要清零？

答：reset针对PC、GRF、DM，其中PC需要回到初始指令地址0x00003000，GRF需要清空全部寄存器的值，DM也清空。

（三）IM设计

module IM(

input[31:0] addr,

output [5:0] op,

output [5:0] func,

output [4:0] rs,

output [4:0] rt,

output [4:0] rd,

output [15:0] imm,

output [25:0] imm26

);

wire[9:0] address;

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

assign address[9:0]=addr[11:2];

initial begin

$readmemh("code.txt",im); //指令放入im

end

assign op[5:0]=im[address][31:26];

assign func[5:0]=im[address][5:0];

assign rs[4:0]=im[address][25:21];

assign rt[4:0]=im[address][20:16];

assign rd[4:0]=im[address][15:11];

assign imm[15:0]=im[address][15:0];

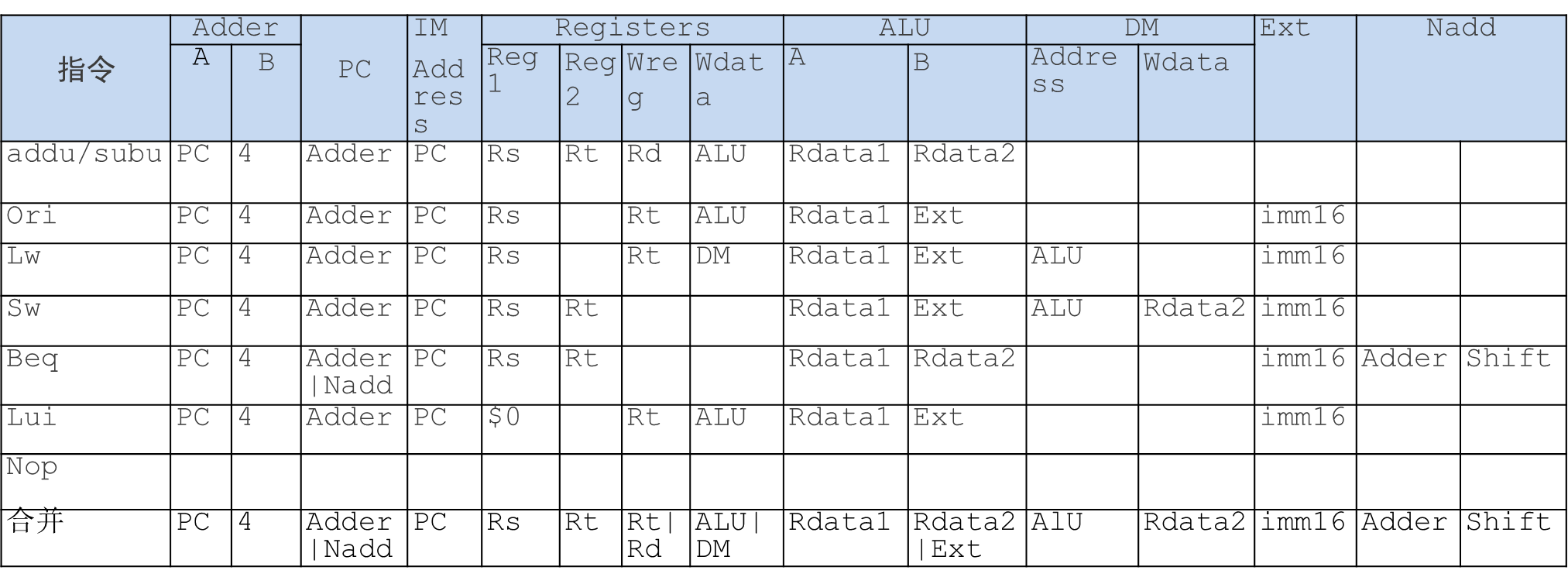
assign imm26[25:0]=im[address][25:0];

endmodule

（四）控制器设计

（一）设计方式

1.数据通路设计



2.主控单元真值表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | addu | subu | ori | lw | sw | beq | lui |
| op | 000000 | 000000 | 001101 | 100011 | 101011 | 000100 | 001111 |
| func | 100001 | 100011 |  |  |  |  |  |
| RegDst | 01 | 01 | 00 | 00 | 00 | 00 | 00 |
| ALUSrc | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| MemtoReg | 00 | 00 | 00 | 01 | 00 | 00 | 00 |
| RegWrite | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| MemWrite | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| PCSrc | 00 | 00 | 00 | 00 | 00 | 01 | 00 |
| Extop | xx | xx | 00 | 01 | 01 | 01 | 10 |
| ALUop | 00 | 01 | 10 | 1 | 00 | 01 | 10 |
| jr | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|  | jal | jr |  |  |  |  |  |
| op | 000011 | 000000 |  |  |  |  |  |
| func |  | 001000 |  |  |  |  |  |
| RegDst | 10 | 00 |  |  |  |  |  |
| ALUSrc | x | 0 |  |  |  |  |  |
| MemtoReg | 10 | 00 |  |  |  |  |  |
| RegWrite | 1 | 0 |  |  |  |  |  |
| MemWrite | 0 | 0 |  |  |  |  |  |
| PCSrc | 10 | 00 |  |  |  |  |  |
| Extop | 00 | 00 |  |  |  |  |  |
| ALUop | 00 | 00 |  |  |  |  |  |
| jr | 0 | 1 |  |  |  |  |  |

1. 思考题
2. 列举出用Verilog语言设计控制器的几种编码方式（至少三种），并给出代码示例。

答：

1. 利用if-else（或case）完成操作码和控制信号的值之间的对应；

always@(\*)begin

case(op)

6'b000000:

begin

if(func==6'b001000)begin //jr

RegDst=2'b00;

ALUSrc=0;

MemtoReg=2'b00;

RegWrite=0;

MemWrite=0;

PCSrc=2'b00;

Extop=2'b00;

ALUop=2'b00;

jr=1;

beq=0;

end

else if(func==6'b100001)begin //addu

RegDst=2'b01;

ALUSrc=0;

MemtoReg=2'b00;

RegWrite=1;

MemWrite=0;

PCSrc=2'b00;

Extop=2'b00;

ALUop=2'b00;

jr=0;

beq=0;

end

else if(func==6'b100011)begin //subu

RegDst=2'b01;

ALUSrc=0;

MemtoReg=2'b00;

RegWrite=1;

MemWrite=0;

PCSrc=2'b00;

Extop=2'b00;

ALUop=2'b01;

jr=0;

beq=0;

end

else begin

RegDst=2'b00;

ALUSrc=0;

MemtoReg=2'b00;

RegWrite=0;

MemWrite=0;

PCSrc=2'b00;

Extop=2'b00;

ALUop=2'b00;

jr=0;

beq=0;

end

end

1. 利用assign语句完成操作码和控制信号的值之间的对应；

assign RegDst[0] = ( (op == 6'b000000 && func == 6'b100001) || (op == 6'b000000 && func == 6'b100011) )? 1 : 0 ;

1. 利用宏定义

`define state1 4'b0001

`define state2 4'b0010

`define state3 4'b0100

`define state4 4'b1000

`define 标识符(宏名) 字符串(宏内容)

如：`define signal string

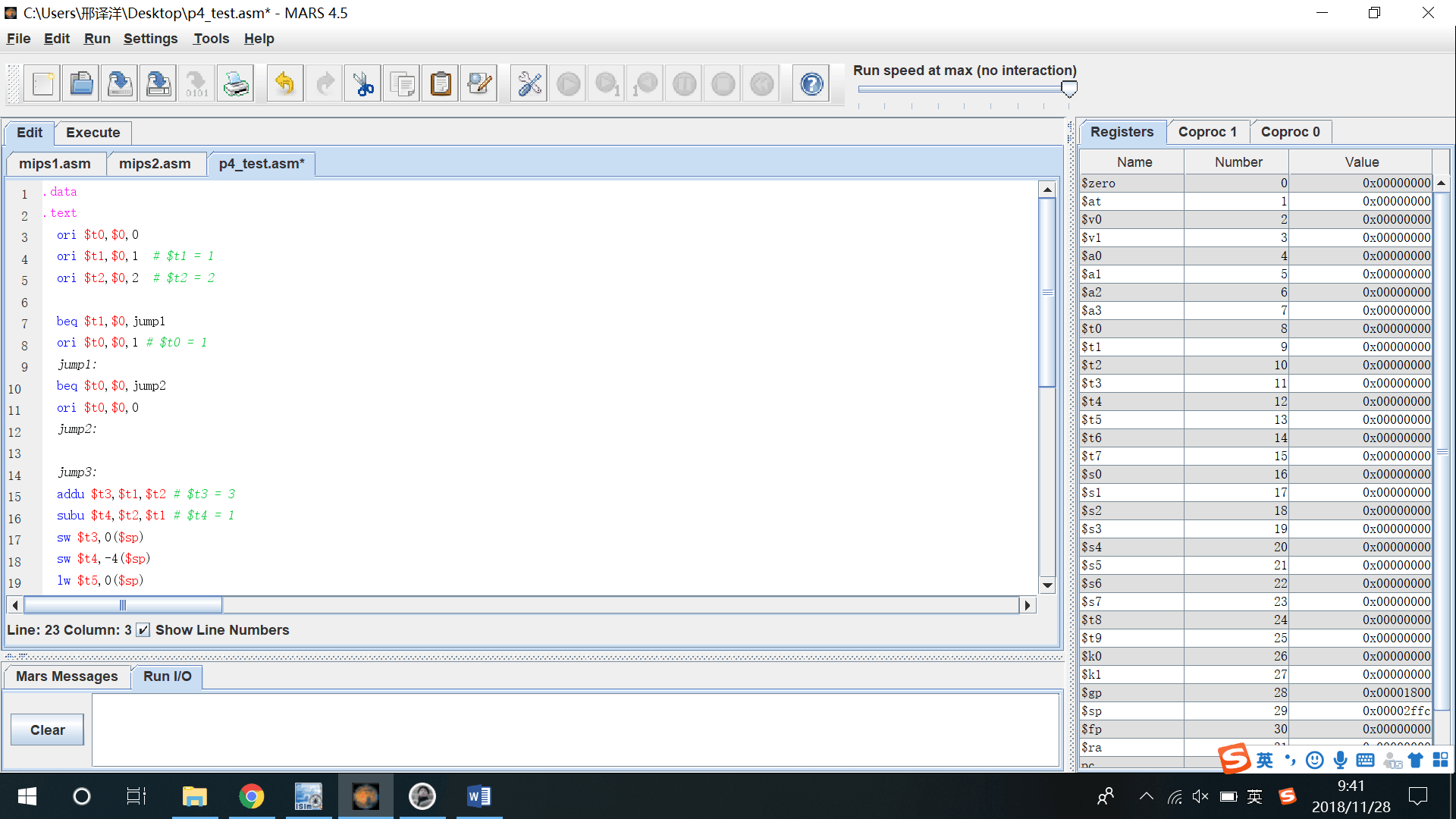
它的作用是指定用标识符signal来代替string这个字符串，在编译预处理时，把程序中在该命令以后所有的signal都替换成string。这种方法使用户能以一个简单的名字代替一个长的字符串，也可以用一个有含义的名字来代替没有含义的数字和符号，因此把这个标识符(名字)称为“宏名”，在编译预处理时将宏名替换成字符串的过程称为“宏展开”。`define是宏定义命令。

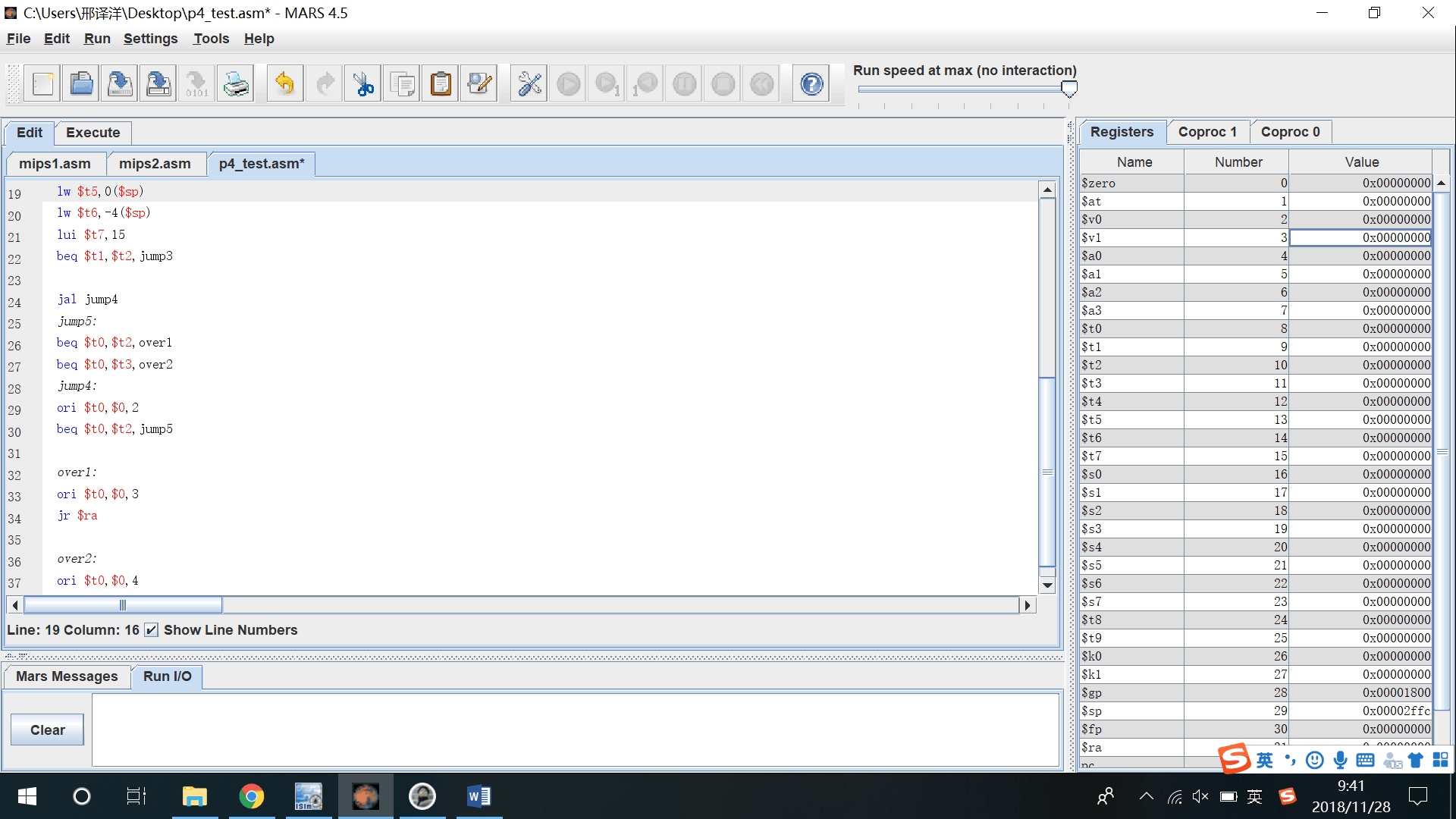
1. 根据你所列举的编码方式，说明他们的优缺点。

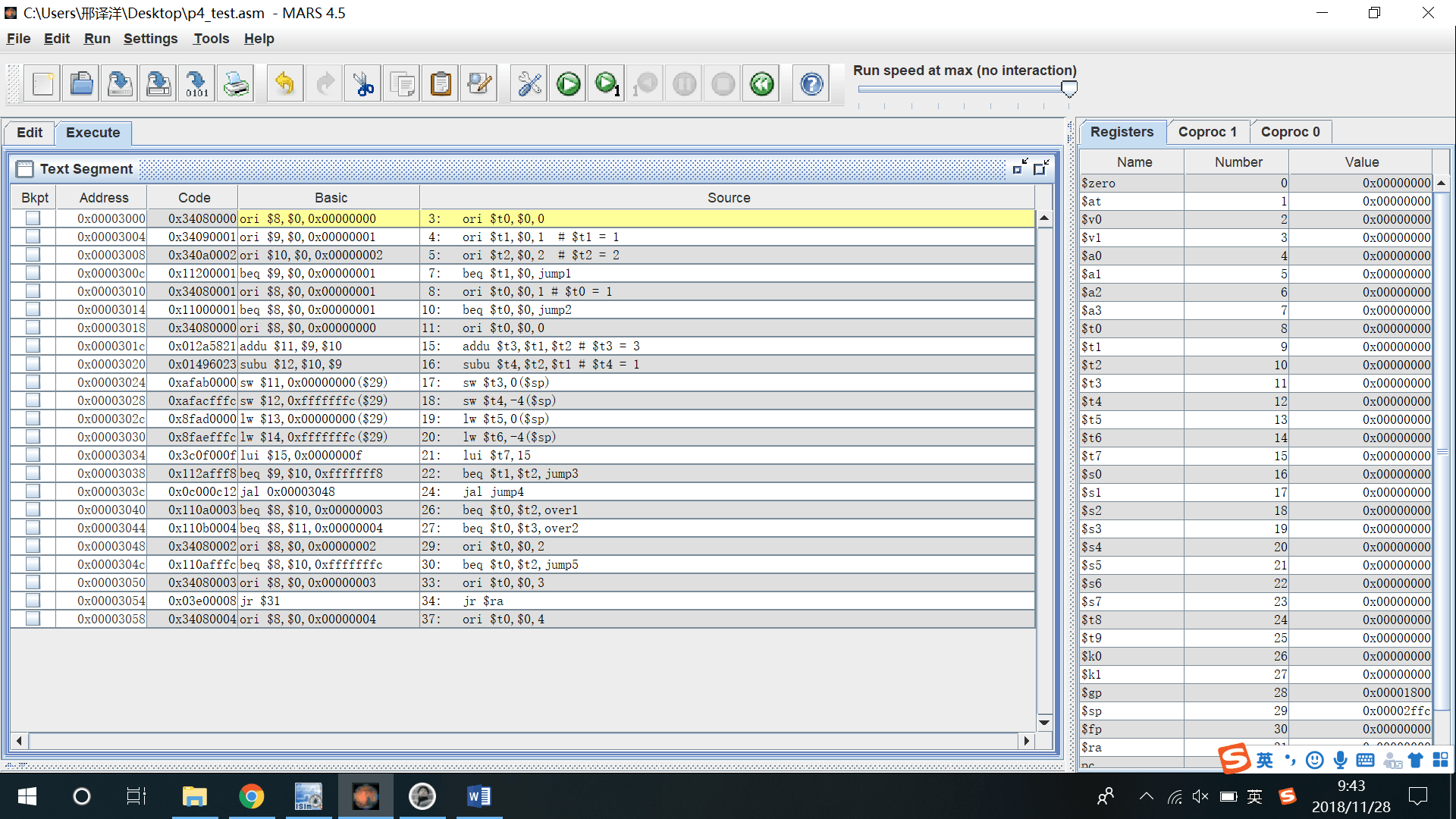
|  |  |  |
| --- | --- | --- |
| 编码方式 | 优点 | 缺点 |
| if-else/case | 代码写起来比较容易、直观，与C语言类似 | 它们在不注意的情况下容易产生锁存器，对毛刺敏感，使其处于不确定状态，且语句较为冗长 |
| assign | 语句较为简单，看起来比较方便 | 适用范围比较窄，只能对wire型变量赋值 |
| 宏定义 | 增加可读性、可修改性好、设计的可重用性好 | 定义时容易出错，且不容易查错 |

（五）在线测试相关信息

（一）测试代码







34080000

34090001

340a0002

11200001

34080001

11000001

34080000

012a5821

01496023

afab0000

afacfffc

8fad0000

8faefffc

3c0f000f

112afff8

0c000c12

110a0003

110b0004

34080002

110afffc

34080003

03e00008

34080004

（二）预期结果

@00003000: $ 8 <= 00000000

@00003004: $ 9 <= 00000001

@00003008: $10 <= 00000002

@00003010: $ 8 <= 00000001

@00003018: $ 8 <= 00000000

@0000301c: $11 <= 00000003

@00003020: $12 <= 00000001

@00003024: \*00000000 <= 00000003

@00003028: \*fffffffc <= 00000001

@0000302c: $13 <= 00000003

@00003030: $14 <= 00000001

@00003034: $15 <= 000f0000

@0000303c: $31 <= 00003040

@00003048: $ 8 <= 00000002

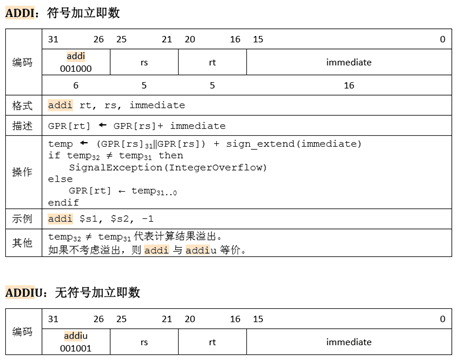
@00003050: $ 8 <= 00000003

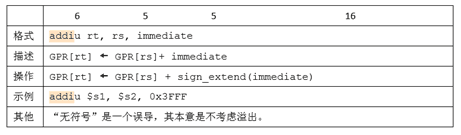
@00003058: $ 8 <= 00000004

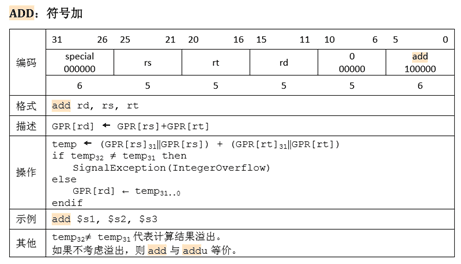
1. 思考题

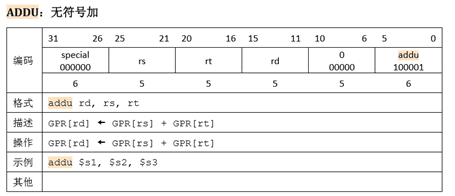
1.C语言是一种弱类型程序设计语言。C语言中不对计算结果溢出进行处理，这意味着C语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持C语言，MIPS指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi与addiu是等价的，add与addu是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中相关指令的Operation部分 。

答：addi和addiu的区别只是addi在发生溢出时会报错，add与addu也是一样，在发生溢出时add会报错。所以在不考虑溢出的情况下，addi和add忽略报错，因此等价。









2.根据自己的设计说明单周期处理器的优缺点。

答：

|  |  |
| --- | --- |
| 优点 | 缺点 |
| 控制部件相比多周期CPU更简单，且实际设计起来较为简便，不容易出错 | 用一个时钟周期执行一条指令，从而确定时钟周期的时间长度要考虑执行的时间最长的指令，一次确定CPU频率，不管指令复杂度如何，单周期CPU花费相同时间执行，这造成时间上浪费 |

简要说明jal、jr和堆栈的关系

答：在跳转到指定地址实现子程序调用的同时，需要将返回地址保存到ra寄存器，即通常所说的“函数调用的现场保护”，以便子程序返回时能够继续调用之前的流程。对于跳转/分支指令，MIPS CPU将自动保存ra；若子程序需要嵌套调用其他子程序，则必须先存储ra，通常是压入栈，子程序末尾弹出之前保存的ra，然后jr到ra。这两条指令分别实现了直接和间接子程序调用。