Verilog搭建单周期CPU设计报告

1. 数据通路设计

1、IFU（取指令单元）

IFU是取指令单元，包含PC（程序计数器）、IM（指令存储器）和相关逻辑。IFU根据当前指令计算出计算出下一条指令的地址并将指令取出。

表格 1 IFU端口说明

|  |  |  |
| --- | --- | --- |
| 端口名 | 方向 | 说明 |
| nPC\_sel[1:0] | I | nPC计算方式的判断信号。（该信号根据当前指令的opcode产生）  0：(非跳转指令) PC←PC+4  1：(beq) PC←PC+4+sign\_ext(imm16||00)  2：(j/jal) PC← (PC[31:28]||imm26||00)  3：(jr) PC←$ra |
| br\_cmp | I | 该转移指令是否需要转移的判断信号（该信号根据ALU判断busA与busB是否相等产生）。  0：不需要转移（busA!=busB）  1：需要转移（busA==busB） |
| imm16[15:0] | I | beq指令的指令移位量，左移两位并符号扩展后存入PC |
| imm26[25:0] | I | j/jal指令的跳转地址，左移两位并无符号扩展后存入PC |
| jrPC[31:0] | I | jr指令的跳转地址，$ra寄存器的值，直接存入PC |
| reset | I | 同步复位信号 |
| clk | I | 时钟信号 |
| instr[31:0] | O | 当前被取出的指令 |
| PC[31:0] | O | 输出当前PC值 |

表格 2 IFU功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 计算beqPC | br\_e==0时，beqPC←PC+4。  br\_e=1时，  beqPC←PC+4+(sign\_ext(imm16)||00)。 |
| 2 | 计算jPC | jPC←{PC+4[31:28],imm26,{2{1'b0}}} |
| 3 | 计算nPC并更新PC | nPC\_sel==2’b00时，PC←PC+4  nPC\_sel==2’b01时，PC←brPC  nPC\_sel==2’b10时，PC←jPC  nPC\_sel==2’b11时，PC←jrPC-32’b00003000 |
| 4 | 取指令 | 从IM中取出地址为PC的指令（注：由于PC以字节为单位、位宽为32，但IM中的ROM以字为单位、位宽为5，故只取PC的2-6位作为指令的地址） |
| 5 | 复位 | clr=1时，PC被设置为0x00003000。 |
| 6 | 输出PC+4 | PC4←PC+4 |

2、GRF（寄存器堆）

GRF由32个寄存器构成（其中0号寄存器恒为0）。首先计算RegA、RegB、RegD信号的值，再进行读或写操作：读操作时，将编号为RegA、RegB的两个寄存器的值读出到busA、busB。写操作时，在RegWr信号为1时，将dataWr写入RegD编号对应的寄存器中。

表格 3 GRF端口说明

|  |  |  |
| --- | --- | --- |
| 端口名 | 方向 | 说明 |
| rt[4:0] | I | 待读出数据1的寄存器编号($rs的编号)。 |
| rd[4:0] | I | 待读出数据2的寄存器编号($rt的编号)。 |
| rs[4:0] | I | 待写入寄存器的编号($rd或$rt的编号) |
| RegDst[1:0] | I | 由Controller产生的控制信号，存储寄存器选择信号。  2’b00: rt  2’b01: rd  2’b10: $31(0x1f)  2’b11: 未定义 |
| dataWr[31:0] | I | 待写入寄存器的数据 |
| RegWr | I | 寄存器写使能信号  0：不写入 1：写入 |
| clk | I | 时钟信号 |
| reset | I | 复位信号 |
| PC[31:0] | I | PC值，$display时会使用 |
| dataA[31:0] | O | 读出数据1($rs的值) |
| dataB[31:0] | O | 读出数据2($rt的值) |

表格 4 GRF功能定义

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 功能名称 | 功能描述 | |
| 1 | 确定各寄存器 | RegA=rs RegB=rt  RegDst==2’b00时，RegD=rt  RegDst==2’b01时，RegD=rd  RegDst==2’b10时，RegD=0x1f |
| 2 | 读寄存器 | 将编号为RegA的寄存器中的数据输出到dataA端口；  将编号为RegB的寄存器中的数据输出到dataB端口 |
| 3 | 写寄存器 | RegWr=1时，在时钟上升沿将dataWr写入到编号为RegW的寄存器中；写入寄存器时进行输出操作 |

3、ALU（算术逻辑单元）

ALU由何种算数逻辑组成。根据ALUctr对ALUin1和ALUin2进行加、减、或、相等比较等操作并输出。

表格 5 ALU端口说明

|  |  |  |  |
| --- | --- | --- | --- |
| 端口名 | 方向 | | 说明 |
| RegA[31:0] | I | GRF的RegA输出，第一个待操作数据。 | |
| RegB[31:0] | I | GRF的RegB输出，可能为第二个待操作数据。 | |
| ExtOut[31:0] | I | EXT的ExtOut输出，可能为第二个待操作数。 | |
| ALUSrc1 | I | 第一个待操作数的选择信号  0：ALUIn1=RegA 1:ALUIn2=RegB | |
| ALUSrc2 | I | 第二个待操作数的选择信号  0: ALUIn2=RegB 1: ALUIn2=ExtOut | |
| ALUOp[1:0] | I | 进行何种运算的选择信号。  00：ALUIn1+ALUIn2  01：ALUIn1-ALUIn2  10：ALUIn1 | ALUIn2  11：ALUIn1<<ALUIn2 | |
| ALUOut[31:0] | O | 输出运算后的结果。 | |
| br\_e | O | 输出ALUin1与ALUin2的相等判断结果。  0：ALUin1!=ALUin2 1：ALUin1==AlUin2 | |

表格 6 ALU功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 确定ALU的两个操作数 | ALUSrc1=0时，ALUIn1=RegA  ALUSrc1=1时，ALUIn1=RegB  ALUSrc2=0时，ALUIn2=RegB  ALUSrc2=1时，ALUIn2=ExtOut |
| 2 | 加（无溢出） | ALUOut=ALUin1 + ALUin2 |
| 2 | 减（无溢出） | ALUOut=ALUin1 - ALUin2 |
| 3 | 或 | ALUOut=ALUin1 | ALUin2 |
| 4 | 移位 | ALUOut=ALUin1 << ALUin2 |
| 5 | 相等判断 | br\_e=(ALUin1==ALUin2) |

4、DM（数据存储器）

表格 7 DM端口说明

|  |  |  |
| --- | --- | --- |
| 端口名 | 方向 | 说明 |
| MemAddr[31:0] | I | DM中的读出/写入地址，即ALU的输出端ALUOut |
| MemData[31:0] | I | 待写入DM的数据,即GRF的输出端dataB |
| MemWr | I | 将DMdata写入DM的写使能信号。  0：不写入 1：写入 |
| clk | I | 时钟信号 |
| reset | I | 复位信号 |
| PC[31:0] | I | PC值，$display时会使用 |
| MemtoReg[1:0] | I | RegData的选择信号  00: RegData=ALUOut=MemAddr  01: RegData=MemOut  10: RegData=PC+0x4  11: （未定义） |
| RegData[31:0] | O | 将数据输出到GRF的dataWr端 |

表格 8 DM功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 读出 | MemOut=DM中MemAddr地址中的数据。 |
| 2 | 写入 | MemWR=1时，将DMdata写入DM的DMaddr地址中。 |
| 3 | 选择写入GRF的数据 | MemtoReg=2’b00时，RegData=ALUOut=MemAddr  MemtoReg=2’b01时，RegData=MemOut  MemtoReg=2’b11时，RegData=PC+0x4 |

5、EXT（扩展器）

EXT根据ExtOp对16位立即数imm16进行各类扩展。

表格 9 EXT端口说明

|  |  |  |
| --- | --- | --- |
| 端口名 | 方向 | 说明 |
| imm[15:0] | I | 待扩展的数据。 |
| ExtOp[1:0] | I | 进行何种扩展的选择信号。  00：无符号扩展  01：有符号扩展  10：加载到高16位，低16位补0  11：（未定义） |
| ExtOut[31:0] | O | 扩展后的数据。 |

表格 10 EXT功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 无符号扩展 | ExtOp=00时，对imm16进行无符号扩展并输出到ExtOut。 |
| 2 | 有符号扩展 | ExtOp=01时，对imm16进行有符号扩展并输出到ExtOut。 |
| 3 | 后补16位0 | ExtOp=10时，将imm16加载到高16位，在低16位补0，并输出到ExtOut。 |

6、DC（译码器）

DC将指令instr拆解。

表格 11 DC端口说明

|  |  |  |
| --- | --- | --- |
| 端口名 | 方向 | 说明 |
| instr[31:0] | I | 当前指令。 |
| fc[5:0] | O | fc[5:0]=instr[5:0] |
| shamt[4:0] | O | shamt[4:0]=instr[10:6] |
| rd[4:0] | O | rd[4:0]=instr[15:11] |
| rt[4:0] | O | rt[4:0]=instr[20:16] |
| rs[4:0] | O | rs[4:0]=instr[25:21] |
| op[5:0] | O | op[5:0]=instr[31:26] |
| imm16[15:0] | O | imm[15:0]=instr[15:0] |
| imm26[25:0] | O | imm26[25:0]=instr[25:0] |

表格 12 DC功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 译码 | 将instr根据上表所述进行译码 |

7、电路图总览

计算机生成了可选文字:
BegPstrd
RegWr
rtnPCsel
CLK
Instr
Fetch
Unit
zero
RWRARBbusA
busW
RegFiIe
busB
32
0
1
ALUctr
1
MemtoReg
MemWr
0
CLK
imm16
ExtOp
0
0
32
Data|n
CLK
WrEnAddr
Data
Memory
1
ALUSrc

图 1 CPU主要部件示意图

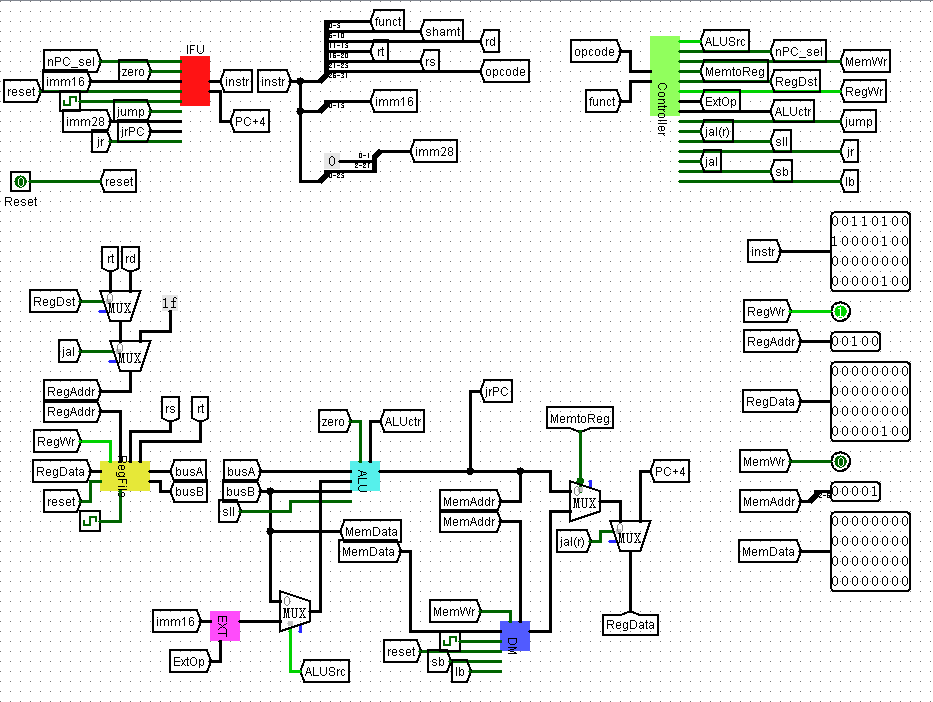


图 2 CPU主要部件示意图（Logisim）

1. Controller（控制器）设计

1、基本描述

Controller根据指令中的opcode段和funct段，先利用与门确定该指令类型，再利用或门确定各控制信号。

|  |  |  |
| --- | --- | --- |
| 端口名 | 方向 | 说明 |
| op [5:0] | I | =instr[25:21] |
| fc[5:0] | I | =instr[5:0] |
| ALUSrc1 | O | 说明见后 |
| ALUSrc2 | O |
| MemWr | O |
| RegWr | O |
| ExtOp[1:0] | O |
| ALUOp[1:0] | O |
| nPC\_sel[1:0] | O |
| RegDst[1:0] | O |
| MemtoReg[1:0] | O |

2、控制信号真值表

表格 13 控制信号真值表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| func | 100001 | 100011 |  |  |  |  |  |
| opcode | 000000 | 000000 | 001101 | 100011 | 101011 | 000100 | 001111 |
|  | addu | subu | ori | lw | sw | beq | lui |
| ExtOp[1:0] | xx | xx | 00 | 01 | 01 | xx | 10 |
| ALUSrc1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ALUSrc2 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| ALUOp[2:0] | 000 | 001 | 010 | 000 | 000 | xxx | 000 |
| nPC\_sel[2:0] | 000 | 000 | 000 | 000 | 000 | 001 | 000 |
| MemWr | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| MemtoReg  [1:0] | 00 | 00 | 00 | 01 | xx | xx | 00 |
| RegDst[1:0] | 01 | 01 | 00 | 00 | xx | xx | 00 |
| RegWr | 1 | 1 | 1 | 1 | 0 | 0 | 1 |

表格 14 控制信号真值表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| func | 000000 |  | 001000 |  | (rt)  00001 |  |  |
| opcode | 000000 | 000011 | 000000 | 000010 | 000001 | 100000 | 101000 |
|  | sll(nop) | jal | jr | j | bgez | lb | sb |
| ExtOp[1:0] | 00 | xx | xx | xx | xx | 01 | 01 |
| ALUSrc1 | 1 | x | 0 | x | 0 | 0 | 0 |
| ALUSrc2 | 1 | x | 0 | x | 0 | 1 | 1 |
| ALUOp[2:0] | 011 | xxx | 000 | xxx | xxx | 000 | 000 |
| nPC\_sel[2:0] | 000 | 010 | 011 | 010 | 100 | 000 | 000 |
| MemWr | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| MemtoReg  [1:0] | 00 | 10 | xx | xx | xx | 01 | xx |
| RegDst[1:0] | 01 | 10 | xx | xx | xx | 00 | xx |
| RegWr | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| lb |  |  |  |  |  | 1 |  |
| sb |  |  |  |  |  |  | 1 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| func | 101011 | 101010 |  | 001001 |  |  |  |
| opcode | 000000 | 000000 | 001000 | 000000 |  |  |  |
|  | sltu | slt | addi | jalr |  |  |  |
| ExtOp[1:0] | xx | xx | 01 | xx |  |  |  |
| ALUSrc1 | 0 | 0 | 0 | 0 |  |  |  |
| ALUSrc2 | 0 | 0 | 1 | 0 |  |  |  |
| ALUOp[2:0] | 100 | 101 | 000 | 000 |  |  |  |
| nPC\_sel[2:0] | 000 | 000 | 000 | 011 |  |  |  |
| MemWr | 0 | 0 | 0 | 0 |  |  |  |
| MemtoReg  [1:0] | 00 | 00 | 00 | 10 |  |  |  |
| RegDst[1:0] | 01 | 01 | 00 | 10 |  |  |  |
| RegWr | 1 | 1 | 1 | 1 |  |  |  |
| lb |  |  |  |  |  |  |  |
| sb |  |  |  |  |  |  |  |

3、控制信号含义

表格 15 控制信号含义1

|  |  |  |
| --- | --- | --- |
|  | 0 | 1 |
| ALUSrc1 | ALUin1=dataA | ALUin2=dataB |
| ALUSrc2 | ALUin2=dataB | ALUin2=ExtOut |
| MemWr | 不可以向DM中写入数据 | 可以向DM中写入数据 |
| RegWr | 不可以向GRF中写入数据 | 可以向GRF中写入数据 |

表格 16 控制信号含义2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 00 | 01 | 10 | 11 |
| ExtOp | 无符号扩展 | 有符号扩展 | 低位补16位0 | （未定义） |
| RegDst | RegWr=rt | RegWr=rd | RegWr=$31 | （未定义） |
| MemtoReg | RegData=  ALUOut | RegData=  MemOut | RegData=  PC4 | （未定义） |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | nPC\_sel | ALUOp |  |  |
| 000 | 顺序执行指令  PC←PC+4 | 加（无符号） |  |  |
| 001 | beq指令  PC←beqPC | 减（无符号） |  |  |
| 010 | j/jal指令  PC←jPC | 或 |  |  |
| 011 | jr指令  PC←jrPC | 逻辑左移 |  |  |
| 100 | bgez指令  PC←bgezPC | Sltu |  |  |
| 101 |  | Slt |  |  |
| 110 |  |  |  |  |
| 111 |  |  |  |  |

1. 测试CPU
2. 测试代码1

#ori

ori $0, $0, 123

ori $a0, $0, 123 #$4=0x0000007b=123

ori $a1, $a0, 456 #$5=0x000001fb=507

#lui

lui $a2, 123 #$6=0x007b0000=8060928

lui $a3, 0xffff

ori $a3, $a3, 0xffff #$7=0xffffffff=-1

nop

#addu

addu $s0, $a0, $a2 #$16=0x007b007b=8061051 ++

addu $s1, $a0, $a3 #$17=0x0000007a=122 +-

addu $s2, $a3, $a3 #$18=0xfffffffe=-2 --

#subu

subu $s3, $a0, $a2 #$19=0xff85007b=-8060805 ++

subu $s4, $a0, $a3 #$20=0x0000007c=124 +-

subu $s5, $a3, $a3 #$21=0x00000000=0 --

#sw

ori $t0, $0, 0x0000

sw $a0, 0($t0) #00000000=0x0000007b

sw $a1, 4($t0)

sw $a2, 8($t0)

sw $a3, 12($t0)

sw $s0, 16($t0)

sw $s1, 20($t0)

sw $s2, 24($t0)

nop

#lw

lw $s0, 24($t0) #$16=$18=-2

lw $s2, 16($t0) #$18=$16=8061051

#beq(up)

ori $s0, $0, 2 #$16=2

ori $s1, $0, 3 #$17=3

ori $t0, $0, 1 #$8=1

up:addu $s0, $s0, $t0 #$16=3 $16=4

beq $s0, $s1, up

#beq(down)

ori $s6, $0, 123 #$22=$4=0x0000007b=123

beq $a0, $a1, loop1

#if($4==$5) 0x0000001c=507 0x00000020=123

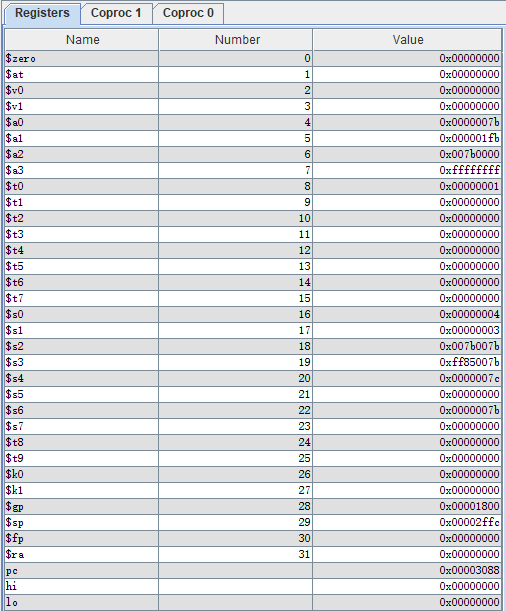
beq $a0, $s6, loop2

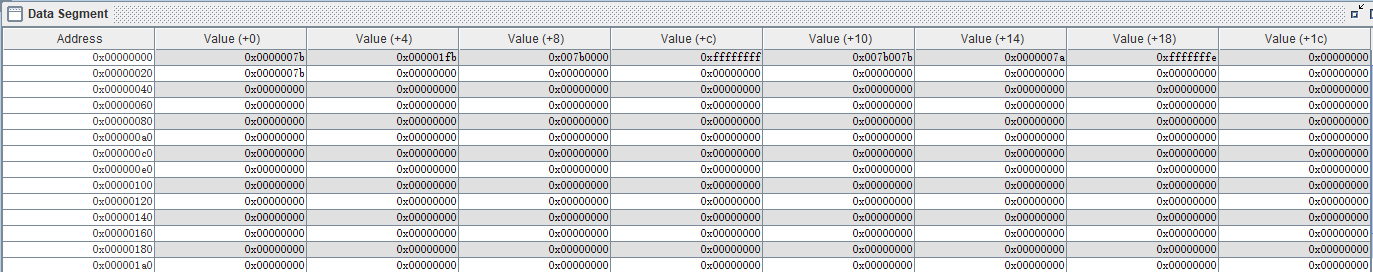
#if($4==$22) 0x0000001c=0 0x00000020=123

loop1:sw $a1, 28($t0)

loop2:sw $s6, 32($0) #00000020=0x0000007b

MIPS结果：





Verilog运行结果:

@00003004: $ 4 <= 0000007b

@00003008: $ 5 <= 000001fb

@0000300c: $ 6 <= 007b0000

@00003010: $ 7 <= ffff0000

@00003014: $ 7 <= ffffffff

@0000301c: $16 <= 007b007b

@00003020: $17 <= 0000007a

@00003024: $18 <= fffffffe

@00003028: $19 <= ff85007b

@0000302c: $20 <= 0000007c

@00003030: $21 <= 00000000

@00003034: $ 8 <= 00000000

@00003038: \*00000000 <= 0000007b

@0000303c: \*00000004 <= 000001fb

@00003040: \*00000008 <= 007b0000

@00003044: \*0000000c <= ffffffff

@00003048: \*00000010 <= 007b007b

@0000304c: \*00000014 <= 0000007a

@00003050: \*00000018 <= fffffffe

@00003058: $16 <= fffffffe

@0000305c: $18 <= 007b007b

@00003060: $16 <= 00000002

@00003064: $17 <= 00000003

@00003068: $ 8 <= 00000001

@0000306c: $16 <= 00000003

@0000306c: $16 <= 00000004

@00003074: $22 <= 0000007b

@00003084: \*00000020 <= 0000007b

1. 测试代码2

#jal+jr

ori $t0, $0, 1 #$8=1

ori $t1, $0, 1 #$9=1

ori $t2, $0, 1 #$10=1

jal dd #$31=0x0000

addu $t1, $t1, $t2

jal end

dd:addu $t0, $t0, $t2

jr $ra

end:

#$8=1

#$9=1

#$10=1

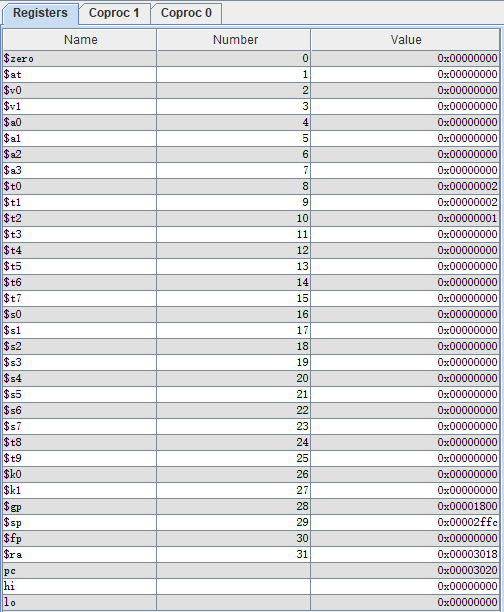
#$31=0x00003010

#$8=2

#$9=2

#$31=0x00003018

MIPS运行结果：



Verilog运行结果：

@00003000: $ 8 <= 00000001

@00003004: $ 9 <= 00000001

@00003008: $10 <= 00000001

@0000300c: $31 <= 00003010

@00003018: $ 8 <= 00000002

@00003010: $ 9 <= 00000002

@00003014: $31 <= 00003018

1. 测试代码3

lui $s0, 0xffee

ori $s0, $s0, 0xccdd

sw $s0, 0($0)

ori $t0, $0, 1

ori $t1, $0, 0

bgez $t0, bj1

bj3: lb $s1, 1($0)

j end

bj1: lb $s2, 3($0)

bgez $t1, bj2

sb $s0, 4($0)

bj2: sb $s0, 6($0)

bgez $t1, bj3

end:

预计输出：$16 <= ffee0000

$16 <= ffeeccdd

\*00000000 <= ffeeccdd

$ 8 <= 00000001

$ 9 <= 00000000

$18 <= ffffffff

\*00000006 <= dd

$17 <= ffffffcc

实际输出：@00003000: $16 <= ffee0000

@00003004: $16 <= ffeeccdd

@00003008: \*00000000 <= ffeeccdd

@0000300c: $ 8 <= 00000001

@00003010: $ 9 <= 00000000

@00003020: $18 <= ffffffff

@0000302c: \*00000006 <= dd

四、思考题

**1、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：

因为DM大小为32bit\*1024，即为以字为单位进行读入和读出，而地址信号addr是以字节为单位的，在执行lw、sw指令时要将addr除以4，即右移2位，即取[11:2]。addr信号是ALU的输出信号ALUOut，是GPR[rs(base)]+sign\_extend(offset)的结果。

**2、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针对程序计数器PC、寄存器堆GRF32和数据内存DM1024复位。

原因：清楚上一次执行留下的所有痕迹，保证在下一次执行开始时PC为0x00003000，GRF32和DM1024都为0。

**3、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case语句对每一个指令生成控制信号（以addu,jr,ori,jal四个指令为例）

case(op)

6'b000000:

case(fc)

6'b100001: begin //addu

RegDst = 2'b01;

RegWr = 1'b1;

end

6'b001000: begin //jr

RegDst = 2'b00;

RegWr = 1'b0;

end

endcase

6'b001101: begin //ori

RegDst = 2'b00;

RegWr = 1'b1;

end

6'b000011: begin //jal

RegDst = 2'b10;

RegWr = 1'b0;

end

endcase

1. 利用assign语句书写控制信号布尔表达式（或）

与（4）类似，但没有`define 及addu,subu等各种指令的wire定义

1. 利用assign语句书写指令表达式及控制信号表达式（与+或）

与（4）类似，但没有`define

1. 上述（1）--（3）均可利用宏定义进行优化

`define R\_OP 6'b000000

`define ADDU\_FC 6'b100001

`define SUBU\_FC 6'b100011

`define ORI 6'b001101

`define LW 6'b100011

`define SW 6'b101011

`define BEQ 6'b000100

`define LUI 6'b001111

`define SLL\_FC 6'b000000

`define JAL 6'b000011

`define JR\_FC 6'b001000

wire addu,subu,ori,lw,sw,beq,lui,sll,jal,jr;

assign addu = (op==`R\_OP)&(fc==`ADDU\_FC);

assign subu = (op==`R\_OP)&(fc==`SUBU\_FC);

assign ori = (op==`ORI);

assign lw = (op==`LW);

assign sw = (op==`SW);

assign beq = (op==`BEQ);

assign lui = (op==`LUI);

assign sll = (op==`R\_OP)&(fc==`SLL\_FC);

assign jal = (op==`JAL);

assign jr = (op==`R\_OP)&(fc==`JR\_FC);

assign ALUSrc1 = sll;

assign ALUSrc2 = ori|lw|sw|lui|sll;

assign MemWr = sw;

assign RegWr = addu|subu|ori|lw|lui|sll|jal;

assign ExtOp = {lui,lw|sw};

assign ALUOp = {ori|sll,sll|subu};

assign nPC\_sel = {jal|jr,beq|jr};

assign RegDst = {jal,addu|subu|sll};

assign MemtoReg = {jal,lw};

**4、根据你所列举的编码方式，说明他们的优缺点。**

答：

1. 优点：简单直观，直接翻译表格。缺点：指令和控制信号的增加会导致代码量的线性增加。
2. 优点：相比于（1）代码量明显减少。缺点：相比于（3）不够直观。
3. 优点：与+或的布尔表达式使得可读性提高，而且代码量随指令和控制信号的增加小幅度增加。缺点：相比于（4）有一点繁琐。
4. 优点：宏定义使得代码可读性和可维护性再度提高。缺点：目前没发现。

**5、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：

在忽略溢出的情况下，add(addi)指令仍然是把计算结果的低32位写入$rd之中（即else部分），与addu(addiu)的行为相同。

temp ← (GPR[rs] 31 ||GPR[rs] 31..0 ) +

(GPR[rt] 31 ||GPR[rt] 31..0 )

if temp 32 ≠ temp 31 then

SignalException(IntegerOverflow)

else

GPR[rd] ← temp

**6、根据自己的设计说明单周期处理器的优缺点。**

答：

优点：实现简单，一个周期处理一条指令，在当前周期只需要考虑当前指令的行为。

缺点：考虑到组合逻辑的输出和输入间存在延迟，较长的数据通路会使得时钟周期只能设置在较慢水平。

**7、简要说明jal、jr和堆栈的关系。**

答：

Jal, jr常用于函数调用，尤其在递归程序的实现之中，我们进入函数和跳转回原来位置的时候不可避免的要使用jal,jr。一般调用函数时会用jal指令将函数结束时的跳回的位置存入$ra，然后将$ra的值存到堆栈之中，在函数调用结束的时候就将堆栈中存储的数据以及返回地址信息回写到$ra之中，再用jr指令进行跳转。