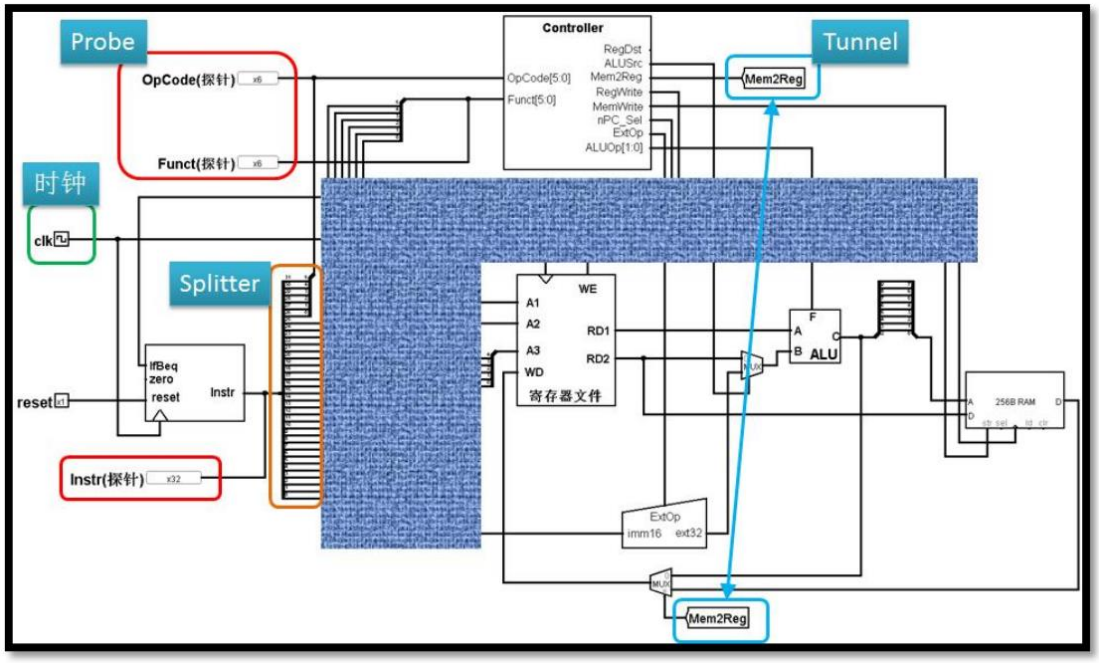
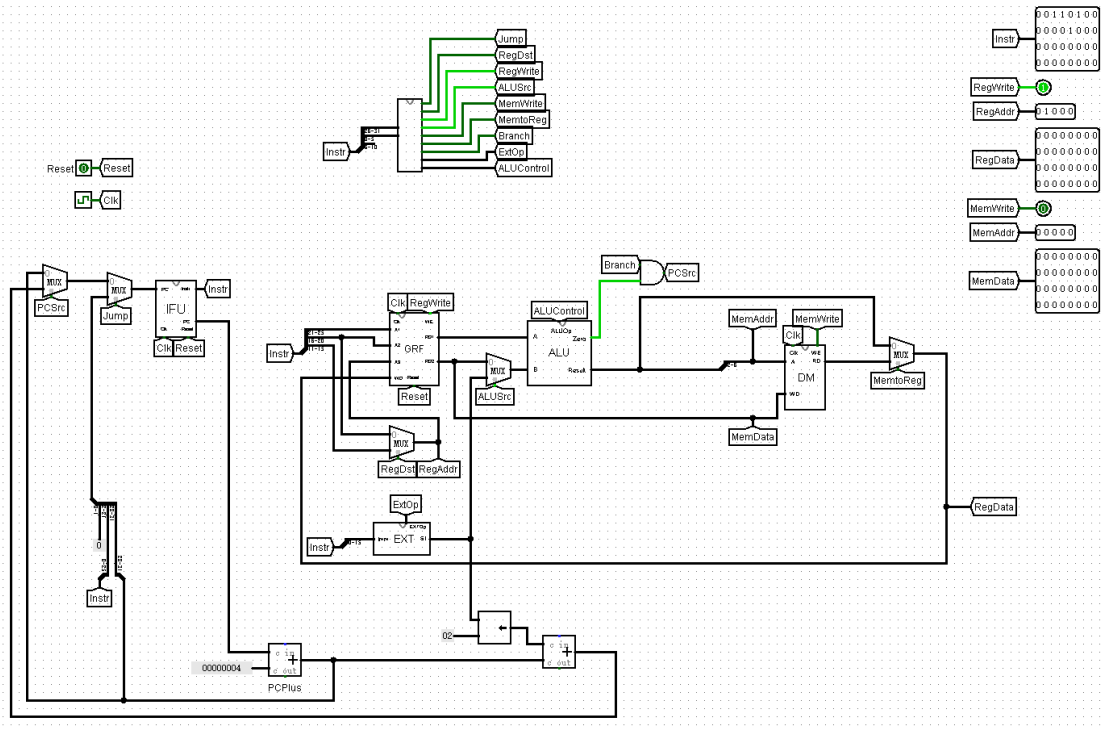
Project3 Logisim完成单周期处理器开发

一、 顶层设计视图





二、 模块定义

1、 IFU（取指令单元）

(1) 基本描述：

① PC 用寄存器实现，应具有复位功能。

② 起始地址：0x00000000。

③ IM 用 ROM 实现，容量为 32bit * 32。

④ 因 IM 实际地址宽度仅为 5 位，故需要使用恰当的方法将 PC 中储存的地址同 IM 联系起来。

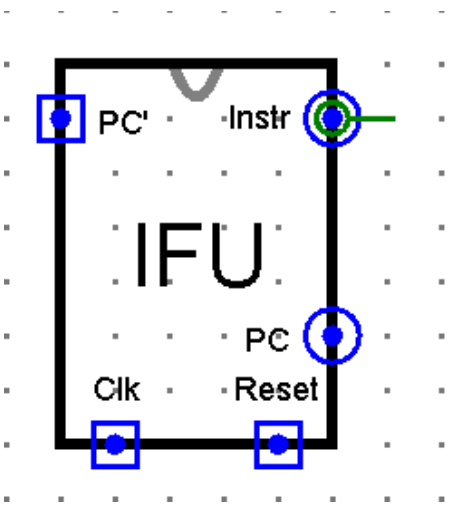
(2) 模块定义：

信号名	方向	描述
Clk	I	时钟信号
Reset	I	复位信号 1：复位 0：无效
PC’	I	更新 PC 值
PC	O	当前 PC 值
Instr[31:0]	O	指令存储器取出的 32 位指令

(3) 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 0x00000000
2	取指令	根据 PC 从 IM 中取出指令

(4) 模块设计：



2、 GRF（通用寄存器组，也称为寄存器文件、寄存器堆）

(1) 基本描述：

① 用具有写使能的寄存器实现，寄存器总数为 32 。

② 0 号寄存器的值始终保持为 0。其他寄存器初始值均为 0，无需专门设置。

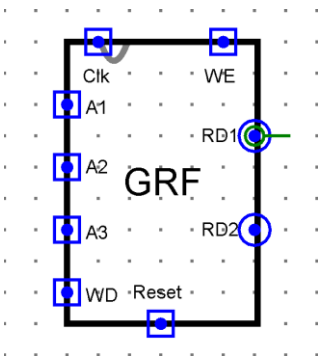
(2) 模块定义：

信号名	方向	描述
Clk	I	时钟信号
Reset	I	复位信号，将 32 个寄存器中的值全部清零 1：复位 0：无效
WE	I	写使能信号 1：可向 GRF 中写入数据 0：不能向 GRF 中写入数据
A1[4:0]	I	5 位地址输入信号，指定 32 个寄存器中的一个，将其中存储的数据读出到 RD1
A2[4:0]	I	5 位地址输入信号，指定 32 个寄存器中的一个，将其中存储的数据读出到 RD1
A3[4:0]	I	5 位地址输入信号，指定 32 个寄存器中的一个，作为写入的目标寄存器
WD[31:0]	I	32 位数据输入信号
RD1[31:0]	O	输出 A1 指定的寄存器中的 32 位数据
RD1[31:0]	O	输出 A2 指定的寄存器中的 32 位数据

(3) 功能定义：

序号	功能名称	功能描述
1	复位	Reset 信号有效时，所有寄存器存储的数值清零，其行为与 logisim 自带部件 register 的 Reset 接口完全相同
2	读数据	读出 A1，A2 地址对应寄存器中所存储的数据到 RD1，RD2
3	写数据	当 WE 有效且时钟上升沿来临时，将 WD 写入 A3 所对应的寄存器中

(4) 顶层设计：



3、 ALU（算术逻辑单元）

(1) 基本描述：

① 提供 32 位加、减、或运算及大小比较功能。

② 可以不支持溢出（不检测溢出）。

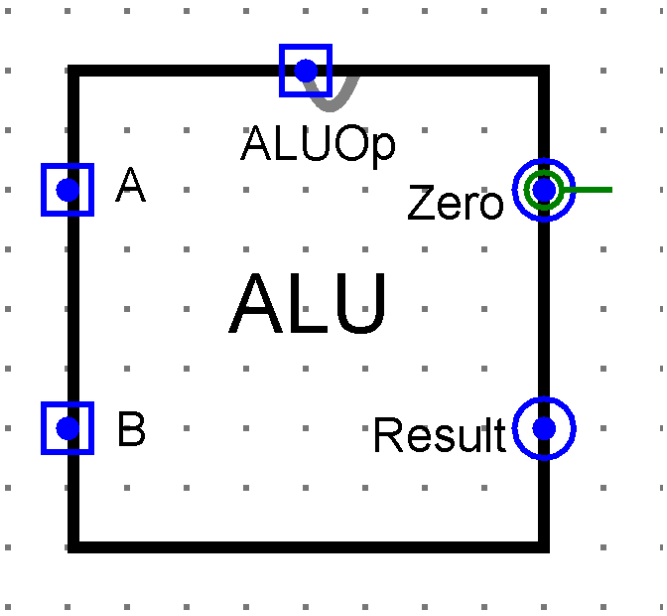
(2) 模块定义：

信号名	方向	描述
A [31:0]	I	参与 ALU 计算的第一个值
B [31:0]	I	参与 ALU 计算的第二个值
ALUOp [3:0]	I	ALU 功能的选择信号 0000：加法运算 0001：减法运算 0010：或运算
Zero	O	零标志位
Result [31:0]	O	ALU 的 32 位运算结果

(3) 功能定义：

序号	功能名称	功能描述
1	加运算	C = A + B
2	减运算	C = A - B
3	与运算	C = A & B

(4) 顶层设计：



4、 DM（数据存储器）

(1) 基本描述：

① 使用 RAM 实现，容量为 32bit * 32。

② 起始地址：0x00000000。

③ RAM 应使用双端口模式，即设置 RAM 的 Data Interface 属性为 Separate load and store ports。

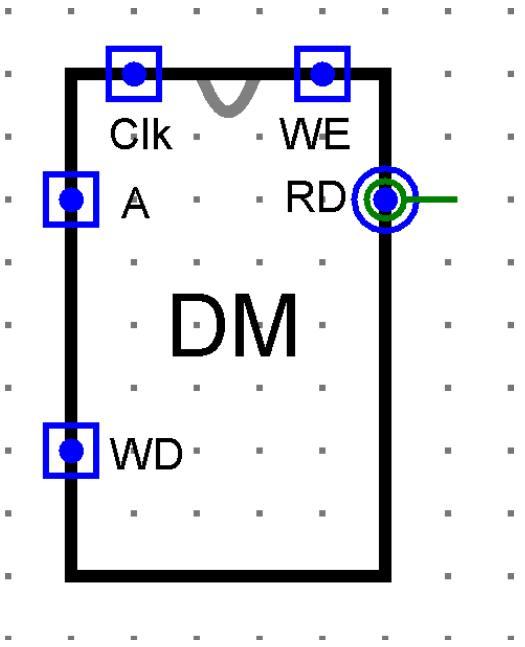
(2) 模块定义：

信号名	方向	描述
clk	I	时钟信号
WE	I	写使能信号
A [4:0]	I	5 位地址输入信号
WD [31:0]	I	32 位被写入存储器中的数据
RD [31:0]	O	32 位被 A 中地址选择并输出的数据

(3) 功能定义：

序号	功能名称	功能描述
1	读数据	读出地址 A 中所存储的数据到 RD
2	写数据	当 WE 有效且时钟上升沿来临时，将 WD 写入地址 A

(4) 模块设计：



5、 EXT

(1) 基本描述：

可以使用 logisim 内置的 Bit Extender。

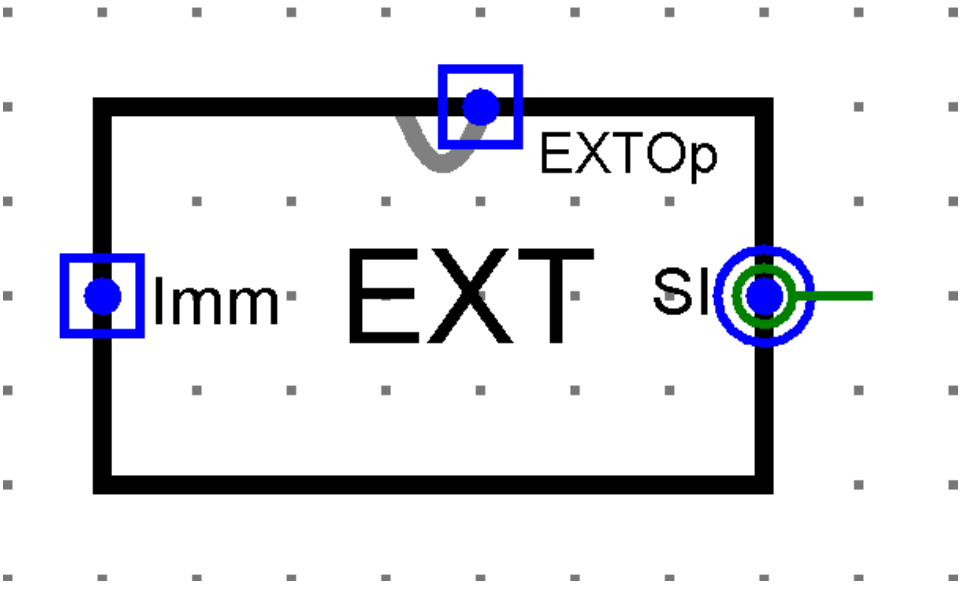
(2) 模块定义：

信号名	方向	描述
Imm [15:0]	I	16 位输入数据
EXTOp [1:0]	I	控制扩展方式 00：符号扩展 01：零扩展 10：加载至高位 11：符号扩展之后，左移两位
SI [31:0]	O	扩展后的 32 位输出数据

(3) 功能定义：

序号	功能名称	功能描述
1	符号扩展	将 16 位输入数据符号扩展为 32 位
2	零扩展	将 16 位输入数据符号置为低 16 位，高 16 位置 0
3	加载至高位	将 16 位输入数据符号置为高 16 位，低 16 位置 0
4	符号扩展之后，左移两位	将 16 位输入数据符号扩展为 32 位之后，左移两位

(4) 模块设计：



三、 控制器设计

1、 基本描述：

控制单元基于指令的 opcode 字段（$Instr_{31:26}$）和 funct 字段（$Instr_{5:0}$）计算控制信号。

2、 模块定义：

信号名	方向	描述
Op [5:0]	I	用于识别指令的功能
Func [5:0]	I	用于辅助 op 来识别指令
Jump	O	J 跳转指令的控制信号
RegDst	O	控制写入端地址选择 0：寄存器堆写入端地址选择 Rt 字段 1：寄存器堆写入端地址选择 Rd 字段
RegWrite	O	GRF 的写使能信号 0：无效 1：把数据写入寄存器堆中对应寄存器
ALUSrc	O	控制 ALU 的操作 0：ALU 输入端 B 选择寄存器堆输出 R[rt] 1：ALU 输入端 B 选择 extend(immediate)
MemWrite	O	DM 的写使能信号 0：无效 1：数据存储器 DM 写数据（输入）
MemtoReg	O	控制数据从 ALU 读出还是从 DM 读出 0：寄存器堆写入端数据来自 ALU 输出 1：寄存器堆写入端数据来自 DM 输出
Branch	O	beq 分支指令的控制信号
ExtOp [1:0]	O	EXT 功能的选择信号 00：符号扩展 01：零扩展 10：加载至高位 11：符号扩展之后，左移两位
ALUControl [3:0]	O	ALU 功能的选择信号 0000：加法运算 0001：减法运算 0010：或运算

3、 支持指令集：

(1)addu 指令：

① 功能：无符号加法，不考虑溢出

② 操作：GPR[rd] ← GPR[rs] + GPR[rt]

③ 编码： 000000[31:26] rs[25:11] rt[20:16] rd[15:11] 00000[10:6] 100001[5:0]

④ 控制信号：

Jump	RegDst	RegWrite	ALUSrc	MemWrite	MemtoReg	Branch	ExtOp	ALUControl
0	1	1	0	0	0	0	xx	0000

(2)subu 指令：

① 功能：无符号减法，不考虑溢出

② 操作：GPR[rd] ← GPR[rs] - GPR[rt]

③ 编码： 000000[31:26] rs[25:11] rt[20:16] rd[15:11] 00000[10:6] 100010[5:0]

④ 控制信号：

Jump	RegDst	RegWrite	ALUSrc	MemWrite	MemtoReg	Branch	ExtOp	ALUControl
0	1	1	0	0	0	0	xx	0001

(3)ori 指令：

① 功能：或立即数

② 操作：GPR[rt] ← GPR[rs] OR zero_extend(immediate)

③ 编码：001101[31:26] rs[25:11] rt[20:16] immediate[15:0]

④ 控制信号：

Jump	RegDst	RegWrite	ALUSrc	MemWrite	MemtoReg	Branch	ExtOp	ALUControl
0	0	1	1	0	0	0	01	0010

(4) lw 指令：

① 功能：加载字，从内存中读取 4 个字节

② 操作：GPR[rt] ← GPR[base] + sign_extend(immediate)

③ 编码：100011[31:26] base[25:11] rt[20:16] offset[15:0]

④ 控制信号：

Jump	RegDst	RegWrite	ALUSrc	MemWrite	MemtoReg	Branch	ExtOp	ALUControl
0	0	1	1	0	1	0	00	0000

(5) sw 指令：

① 功能：存储字，向内存中写入 4 个字节

② 操作：Addr ← GPR[base] + sign_extend(immediate)

Memory[Addr] ← GPR[rt]

③ 编码：101011[31:26] base[25:11] rt[20:16] offset[15:0]

④ 控制信号：

Jump	RegDst	RegWrite	ALUSrc	MemWrite	MemtoReg	Branch	ExtOp	ALUControl
0	x	0	1	1	0	0	00	0000

(6) beq 指令：

① 功能：当两个待比较寄存器相等时，跳转到分支地址

② 操作：if (GPR[rs] == GPR[rt])

PC ← PC + 1 + sign_extend(offset || 0^2)

else

PC ← PC + 1

③ 编码：000100[31:26] rs[25:11] rt[20:16] offset[15:0]

④ 控制信号：

Jump	RegDst	RegWrite	ALUSrc	MemWrite	MemtoReg	Branch	ExtOp	ALUControl
0	x	0	0	0	0	1	00	0001

(7)lui 指令：

① 功能：立即数加载至最高位

② 操作：GPR[rt] ← GPR[zero] + immediate || 0^{16}

③ 编码：001111[31:26] 00000[25:11] rt[20:16] immediate[15:0]

④ 控制信号：

Jump	RegDst	RegWrite	ALUSrc	MemWrite	MemtoReg	Branch	ExtOp	ALUControl
0	0	1	0	0	0	0	10	0000

(8)nop 指令：

① 功能：不执行任何操作

② 操作：无

③ 编码：00000000000000000000000000000000 [31:0]

④ 控制信号：

Jump	RegDst	RegWrite	ALUSrc	MemWrite	MemtoReg	Branch	ExtOp	ALUControl
0	0	0	0	0	0	0	0	xxxx

4、 控制真值表：

指令	Opcode	Funct	Jump	RegDst	RegWrite	ALUSrc	MemWrite	MemtoReg	Branch	ExtOp	ALUControl
addu	000000	100001	0	1	1	0	0	0	0	xx	0000
subu	000000	100011	0	1	1	0	0	0	0	xx	0001
ori	001101	xxxxxx	0	0	1	1	0	0	0	01	0010
lw	100011		0	0	1	1	0	1	0	00	0000
sw	101011		0	x	0	1	1	0	0	00	0000
beq	000100		0	x	0	0	0	0	1	00	0001
lui	001111		0	0	1	1	0	0	0	10	xxxx

Jump = 0

RegDst = addu || subu

RegWrite = addu || subu || ori || lw || lui

ALUSrc = ori || lw || sw || lui

MemWrite = sw

MemtoReg = lw

Branch = beq

ExtOp[0] = ori

ExtOp[1] = lui

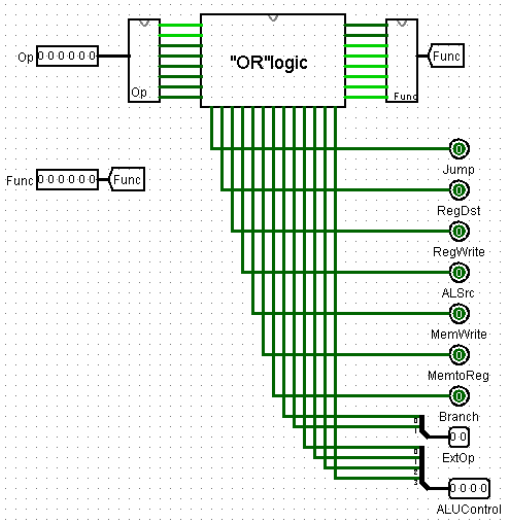
ALUControl[0] = subu || beq

ALUControl[1] = ori

ALUControl[2] = 0

ALUControl[3] = 0

5、 模块设计：



四、 测试 CPU

1、首先测试 addu、subu、ori、lui、nop 这四个指令

```
ori    $1,$0,1
ori    $2,$2,30
ori    $3,$2,50
nop
lui    $4,100
lui    $5,200
lui    $5,300
nop
addu   $6,$1,$2
addu   $7,$5,$6
addu   $7,$7,$7
nop
subu   $8,$7,$2
subu   $9,$8,$0
subu   $9,$9,$9
nop
```

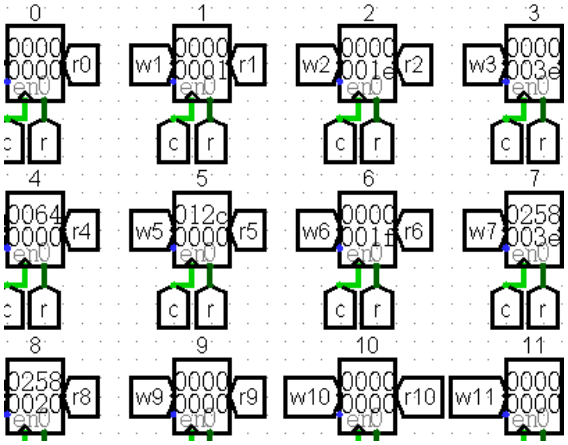
导出的机器码为

34010001	3442001e	34430032	00000000
3c040064	3c0500c8	3c05012c	00000000
00223021	00a63821	00e73821	00000000
00e24023	01004823	01294823	00000000

参与运算的寄存器显示应为

Name	Number	Value
$zero	0	0
$at	1	1
$v0	2	30
$v1	3	62
$a0	4	6553600
$a1	5	19660800
$a2	6	31
$a3	7	39321662
$t0	8	39321632
$t1	9	0

实际结果为



2、其次整体测试全部指令

```
.data
arr:    .space 40
.text
ori     $10,$0,4
ori     $11,$0,0
ori     $12,$0,3
sw      $11,arr($11)
addu    $11,$11,$10
sw      $11,arr($11)
addu    $11,$11,$10
sw      $11,arr($11)
addu    $11,$11,$10
beq     $11,$12,loop1
lui     $13,100
loop1: subu   $11,$11,$10
       lw     $14,arr($11)
       subu   $11,$11,$10
       lw     $15,arr($11)
       subu   $11,$11,$10
       lw     $16,arr($11)
```

```
beq    $11,$0,loop2
ori    $17,100
loop2: ori        $18,200
```

导出的机器码为

340a0004 340b0000 340c0003 ad6b0000 016a5821 ad6b0000

016a5821 ad6b0000 016a5821 116c0001 3c0d0064 016a5823

8d6e0000 016a5823 8d6f0000 016a5823 8d700000 11600001

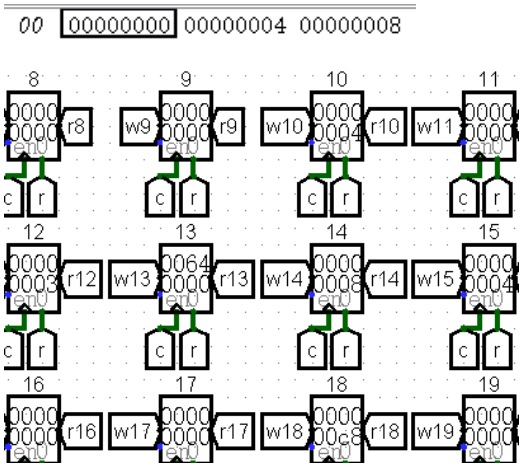
36310064 365200c8

参与运算的寄存器和内存结果应为

Address	Value (+0)	Value (+4)	Value (+8)
0x00000000	0	4	8

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	0
$t1	9	0
$t2	10	4
$t3	11	0
$t4	12	3
$t5	13	6553600
$t6	14	8
$t7	15	4
$s0	16	0
$s1	17	0
$s2	18	200

实际结果为



测试完成

五、 思考题

(1)若 PC（程序计数器）位数为 30 位，试分析其与 32 位 PC 的优劣。

假设主存空间为 4GB，即 2^{32}B，故可使用 32 位的 PC。优点是直接指出地址位置，缺点是耗费晶体管。

而指令字长也为 32 位，它们是按字对齐的，故最多存放 2^{30} 条指令，故可用 30 位的 PC，取址时左移两位。优点是节省晶体管，缺点是取址时需要移位器。

(2)现在我们的模块中 IM 使用 ROM， DM 使用 RAM， GRF 使用寄存器，这种做法合理吗？ 请给出分析，若有改进意见也请一并给出。

合理。

ROM 为只读存储器，数据能够得到保护，不会被改动。符合 IM 在加载指令后的执行过程中不能被改动的特点。

RAM 为随机访问存储器，可以进行读、写两种操作。符合 DM 在面对 sw、lw 两种操作时的读、写功能的实现，同时 DM 不需要特别快的速度，但需要足够的存储空间。缺点为 RAM 不支持跨地址操作，所以无法实现 sb、lb 等非 4 字节操作指令。

寄存器是有限存贮容量的高速存贮部件，它们可用来暂存指令、数据和地址，符合 GRF 的工作需要。

(3)结合上文给出的样例真值表，给出 RegDst， ALUSrc， MemtoReg，RegWrite, nPC_Sel, ExtOp 与 op 和 func 有关的布尔表达式（表达式中只能使用“与、或、非”3 种基本逻辑运算。）

RegDst = $\overline{o5o4o3o2o1o0}$ f5$\overline{f4f3f2}$f0

ALUSrc = $\overline{o5o4}o3o2\overline{o1}o0$ + o5$\overline{o4o2}$o1o0

MemtoReg = o5$\overline{o4o3o2o1}$o0

RegWrite = $\overline{o5o4o3o2o1o0}$ f5$\overline{f4f3f2}$f0 + $\overline{o5o4}o3o2\overline{o1}o0$ + o5$\overline{o4o3o2}$o1o0

nPC_Sel = o5o4o3$\overline{o2}$o1o0

ExtOp = o5$\overline{o4o2}$o1o0

(4)充分利用真值表中的 X 可以将以上控制信号化简为最简单的表达式， 请给出化简后的形式。

RegDst = $\overline{o5o4o3o2o1o0}$ f5$\overline{f4f3f2}$f0

ALUSrc = $\overline{o5o4}o3o2\overline{o1}o0$ + o5$\overline{o4o2}$o1o0

MemtoReg = o5$\overline{o4o3o2o1}$o0

RegWrite = $\overline{o5o4o3o2o1o0}$ f5$\overline{f4f3f2}$f0 + $\overline{o5o4}o3o2\overline{o1}o0$ + o5$\overline{o4o3o2}$o1o0

nPC_Sel = o5o4o3$\overline{o2}$o1o0

ExtOp = o5$\overline{o4o2}$o1o0

(5)事实上，实现 nop 空指令，我们并不需要将它加入控制信号真值表，为什么？请给出你的理由。

nop 指令没有对逻辑电路中的元件进行任何操作，它的存在与否对电路没有影响。如果不将 nop 加入控制信号真值表，则当控制器接收这条指令时，并不能将它识别为任何有效指令。

(6)前文提到，“可能需要手工修改指令码中的数据偏移”，但实际上只需再增加一个 DM 片选信号,就可以解决这个问题。请阅读相关资料并设计一个 DM 改造方案使得无需手工修改数据偏移。

假设 DM 的容量为 256MB，表示数据的地址范围是 0x3000_0000—0x3FFF_FFFF，在进行片选操作时，增加一个把接收到地址的高 4 位与 0x3 进行比较的片选信号，小于 0x3 存在之前的 DM，大于 0x3 存在下一个 DM，而下一个 DM 最大存储地址为 0x0000_0080，故不再需要片选信号。

(7)除了编写程序进行测试外，还有一种验证 CPU 设计正确性的办法——形式验证。 形式验证的含义是根据某个或某些形式规范或属性，使用数学的方法证明其正确性或非正确性。请搜索“形式验证（Formal Verification)"了解相关内容后，简要阐述相比与测试，形式验证的优劣。

形式验证时要确定电路在哪一级电路上的测试是正确的，使用模型检验的方法看两个电路在描述上是否一致。

对组合逻辑来说，不存在状态寄存器，其输出值 Z[t]不依赖于前面的输入值 X[t-i](1≤i≤t)。这时只要对每个输入向量证明其输出向量相同。

对一个时序电路而言，可以把它看成一个有限状态机。电路功能的等价可以用有限状态机的等价来判断。假定有两个状态机 A 和 B，要对它们进行比较。直观的说，当 A 和 B 有相同的接口，而且从相同的初始状态出发，两者对有效输入值序列产生相同的输出值序列，则可以说 A 和 B 等价。

形式验证的优点如下：

①形式验证是对指定描述的所有可能的情况进行验证，覆盖率达到了 100%。

②形式验证技术是借用数学上的方法将待验证电路和功能描述或参考设计直接进行比较，不需要开发测试激励。

③形式验证的验证时间短，可以很快发现和改正电路设计中的错误，可以缩短设计周期。

形式验证的优点如下：

①不能发现代码中的功能错误和时序错误，规模太大的话验证时间更大。

②不能有效的验证电路的性能，如电路的时延和功耗等。