哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验报告

开课学期：2020 夏季

课程名称：计算机设计与实践

实验名称：多周期 CPU 设计

实验性质：综合设计型

实验学时：42 地点：线上

学生班级：5 班

学生学号：180110518

学生姓名：胡智胜

评阅教师：

报告成绩：

实验与创新实践教育中心制

2020 年 5 月

注：

本设计报告中各个部分如果页数不够，请大家自行扩页，原则是一定要把报告写详细，能说明设计的成果和特色。报告中应该叙述设计中的每个模块。设计报告将是评定每个人成绩的重要组成部分（**设计内容及报告写作**都作为评分依据）。

完成单周期 CPU 设计与实现的同学根据单周期设计的内容完成实验报告。完成多周期 CPU 设计与实现的同学根据多周期设计的内容完成实验报告。

设计的功能描述（含所有实现的指令描述及模块的功能）

实现了多周期 CPU 的设计，能够运行完整的(24+7)条指令，且作为附加实验个人实现了一个简单的汇编器，能够将在实验一中的用 23 条 MIPS 汇编指令写的汇编程序编译成相应机器码，并在老师给出的 logisim 模型上成功运行，**在本报告的最后附加部分给出了汇编器的设计报告**；

实现的指令描述：

1.R 型指令

- add

指令名：加法指令

汇编格式：add rd, rs, rt

编码格式：

R-类型	op	rs	rt	rd	shamt	func
add	000000	rs	rt	rd	00000	100000

汇编举例：add $4, $2, $3

功能描述：rd ← rs + rt; PC ← NPC (PC + 4)。 32 位整数加法，源操作数分别在 rs, rt 两个通用寄存器中，结果放在 rd 寄存器。

- addu

指令名: 无符号数加法指令

汇编格式：addu rd, rs, rt

编码格式：

R-类型	op	rs	rt	rd	shamt	func
addu	000000	rs	rt	rd	00000	100001

汇编举例：addu $4, $2, $3

功能描述：rd ← rs + rt; PC ← NPC (PC + 4)。 32 位无符号整数加法，源操作数分别在 rs, rt 两个通用寄存器中，结果放在 rd 寄存器。

- sub

指令名：减法指令

汇编格式：sub rd, rs, rt

编码格式：

R-类型	op	rs	rt	rd	shamt	func
sub	000000	rs	rt	rd	00000	100010

汇编举例：sub $4, $2, $3

功能描述：rd ← rs - rt; PC ← NPC (PC + 4)。 32 位整数减法，源操作数分别在 rs, rt 两个通用寄存器中，结果放在 rd 寄存器。

- subu

指令名：无符号减法指令

汇编格式：subu rd, rs, rt

编码格式：

R-类型	op	rs	rt	rd	shamt	func

subu	000000	rs	rt	rd	00000	100011

汇编举例：subu $4, $2, $3

功能描述：rd ← rs - rt; PC ← NPC (PC + 4)。 32 位无符号整数减法，源操作数分别在 rs, rt 两个通用寄存器中，结果放在 rd 寄存器。

- and

指令名： 逻辑与

汇编格式：and rd, rs, rt

编码格式：

R-类型	op	rs	rt	rd	shamt	func
and	000000	rs	rt	rd	00000	100100

汇编举例：and $4, $2, $3

功能描述：rd ← rs and rt; PC ← NPC (PC + 4)。 32 位数按位逻辑与，源操作数分别在 rs, rt 中，结果放在 rd 寄存器。

- or

指令名： 逻辑或

汇编格式：or rd, rs, rt

编码格式：

R-类型	op	rs	rt	rd	shamt	func
or	000000	rs	rt	rd	00000	100101

汇编举例：or $4, $2, $3

功能描述：rd ← rs or rt; PC ← NPC (PC + 4)。 32 位数按位逻辑或，源操作数分别在 rs, rt 中，结果放在 rd 寄存器。

- xor

指令名： 逻辑异或

汇编格式：xor rd, rs, rt

编码格式：

R-类型	op	rs	rt	rd	shamt	func
xor	000000	rs	rt	rd	00000	100110

汇编举例：xor $4, $2, $3

功能描述：rd ← rs xor rt; PC ← NPC (PC + 4)。 32 位数按位逻辑异或，源操作数分别在 rs, rt 中，结果放在 rd 寄存器。

- nor

指令名： 逻辑或非

汇编格式：nor rd, rs, rt

编码格式：

R-类型	op	rs	rt	rd	shamt	func
nor	000000	rs	rt	rd	00000	100111

汇编举例：nor $4, $2, $3

功能描述：rd ← rs nor rt; PC ← NPC (PC + 4)。 32 位数按位逻辑或非，源操作数分别在 rs, rt 中，结果放在 rd 寄存器。

- slt

指令名：小于则设置指令
汇编格式：slt rd, rs, rt
编码格式：

R-类型	op	rs	rt	rd	shamt	func
slt	000000	rs	rt	rd	00000	101010

汇编举例：slt $4, $2, $3
功能描述： if ((rs)<(rt)) then (rd)←1; else (rd)←0; PC ← NPC (PC + 4)。如果 rs 的值小于 rt 值，则设置 rd 为 1，否则 rd 为 0。

- sltu

指令名：无符号小于则设置指令
汇编格式：sltu rd, rs, rt
编码格式：

R-类型	op	rs	rt	rd	shamt	func
sltu	000000	rs	rt	rd	00000	101011

汇编举例：sltu $4, $2, $3
功能描述： if ((rs)<(rt)) then (rd)←1; else (rd)←0; PC ← NPC (PC + 4)。无符号数小于判断，如果 rs 的值小于 rt 值，则设置 rd 为 1，否则 rd 为 0。

- sll

指令名：逻辑左移
汇编格式：sll rd, rt, shamt
编码格式：

R-类型	op	rs	rt	rd	shamt	func
sll	000000	00000	rt	rd	shamt	000000

汇编举例：sll $4, $2, 10
功能描述： (rd)←(rt)<<shamt; PC ← NPC (PC + 4)。逻辑左移，将 rt 寄存器中的 32 位数逻辑左移后赋给 rd，低位用 0 填充，移位的位数是 shamt。

- srl

指令名：逻辑右移
汇编格式：srl rd, rt, shamt
编码格式：

R-类型	op	rs	rt	rd	shamt	func
srl	000000	00000	rt	rd	shamt	000010

汇编举例：srl $4, $2, 10
功能描述： (rd)←(rt)>>shamt; PC ← NPC (PC + 4)。逻辑右移，将 rt 寄存器中的 32 位数逻辑右移后赋给 rd，移位的位数是 shamt。80000000H 逻辑右移 1 位的结果是 40000000H。

- sra

指令名：算术右移

汇编格式：sra rd, rt, shamt
编码格式：

R-类型	op	rs	rt	rd	shamt	func
sra	000000	00000	rt	rd	shamt	000011

汇编举例：sra $4, $2, 10
功能描述： (rd)←(rt)>>shamt; PC ← NPC (PC + 4)。算术右移，将 rt 寄存器中的 32 位数算术右移后赋给 rd，移位的位数是 shamt。算术右移时， 符号位不仅要参与移位，还要保留，如 80000000H 算术右移 1 位的结果是 0C0000000H。

- sllv

指令名: 按寄存器值逻辑左移指令
汇编格式：sllv rd, rt, rs
编码格式：

R-类型	op	rs	rt	rd	shamt	func
sllv	000000	rs	rt	rd	00000	000100

汇编举例：sllv $4, $2, $3
功能描述： (rd)←(rt)<<(rs); PC ← NPC (PC + 4)。按寄存器值逻辑左移指令，将 rt 寄存器中的 32 位数逻辑左移后赋给 rd，低位用 0 填充，移位的位数在 rs 寄存器中。

- srlv

指令名: 按寄存器值逻辑右移指令
汇编格式：srlv rd, rt, rs
编码格式：

R-类型	op	rs	rt	rd	shamt	func
srlv	000000	rs	rt	rd	00000	000110

汇编举例：srlv $4, $2, $3
功能描述： (rd)←(rt)>>(rs); PC ← NPC (PC + 4)。按寄存器值逻辑右移指令，将 rt 寄存器中的 32 位数逻辑右移后赋给 rd，移位的位数在 rs 寄存器中。

- srav

指令名: 按寄存器值算术右移指令
汇编格式：srav rd, rt, rs
编码格式：

R-类型	op	rs	rt	rd	shamt	func
srav	000000	rs	rt	rd	00000	000111

汇编举例：srav $4, $2, $3
功能描述： (rd)←(rt)>>(rs); PC ← NPC (PC + 4)。按寄存器值算术右移指令，将 rt 寄存器中的 32 位数算术右移后赋给 rd，移位的位数在 rs 寄存器中。算术右移时，符号位不仅要参与移位，还要保留。

- jr

指令名: 按寄存器内容转移指令
汇编格式：JR rs
编码格式：

R-类型	op	rs	rt	rd	shamt	func
jr	000000	rs	00000	00000	00000	001000

汇编举例：JR $31 功能描述：(PC)←(rs)；PC ← NPC (PC + 4)。将 rs 寄存器的内容当地址，赋给 PC，从而完成转移，通常可做过程返回语句。实际系统中只用了低 16 位地址线。

2.I 型指令

- addi

指令名：有符号立即数加法指令

汇编格式：ADDI rt, rs, immediate

编码格式：

I-类型	op	rs	rt	immediate
addi	001000	rs	rt	immediate

汇编举例：ADDI $4, $2, -100

功能描述：(rt)←(rs) + (Sign-Extend)immediate; PC ← NPC (PC + 4)。首先将 16 位有符号立即数扩展到 32 位，然后加上 rs 中的数，结果给 rt 寄存器。

- addiu

指令名：无符号立即数加法指令

汇编格式：ADDIU rt, rs, immediate

编码格式：

I-类型	op	rs	rt	immediate
addiu	001001	rs	rt	immediate

汇编举例：ADDIU $4, $2, -100

功能描述：(rt)←(rs) + (Sign-Extend)immediate; PC ← NPC (PC + 4)。首先将 16 位有符号立即数扩展到 32 位，然后加上 rs 中的数，结果给 rt 寄存器。

- andi

指令名：立即数逻辑与指令

汇编格式：ANDI rt, rs, immediate

编码格式：

I-类型	op	rs	rt	immediate
andi	001100	rs	rt	immediate

汇编举例：ADDI $4, $2, 1 功能描述：(rt)←(rs) AND (Zero-Extend)immediate; PC ← NPC (PC + 4)。首先将 16 位立即数零扩展到 32 位，然后同 rs 中的数按位逻辑与，结果给 rt 寄存器。

- ori

指令名：立即数逻辑或指令

汇编格式：ORI rt, rs, immediate

编码格式：

I-类型	op	rs	rt	immediate
ori	001101	rs	rt	immediate

汇编举例：ORI $4, $2, 5 功能描述：(rt)←(rs) ORI (Zero-Extend)immediate; PC ← NPC (PC + 4)。首先将 16 位立即数零扩展到 32 位，然后同 rs 中的数按位逻辑或，结果给 rt 寄存器。

- xori

指令名：立即数逻辑异或指令
汇编格式：XORI rt, rs, immediate
编码格式：

I-类型	op	rs	rt	immediate
xori	001110	rs	rt	immediate

汇编举例：XORI $4, $2, 5 功能描述：(rt)←(rs) XORI (Zero-Extend)immediate; PC ← NPC (PC + 4)。首先将 16 位立即数零扩展到 32 位，然后同 rs 中的数按位逻辑异或，结果给 rt 寄存器。

- sltiu

指令名：小于无符号立即数则设置指令
汇编格式：SLTIU rt, rs, immediate
编码格式：

I-类型	op	rs	rt	immediate
sltiu	001011	rs	rt	immediate

汇编举例：SLTIU $3, $2, 10
功能描述：if ((rs)<(sign_extend)immediate) then (rt)←1; else (rt)←0; PC ← NPC (PC + 4)。如果 rs 的值小于立即数 immediate 值（进行的是符号扩展），则设置 rt 为 1，否则 rt 为 0。

- lui

指令名：立即数赋值指令
汇编格式：LUI rt, immediate
编码格式：

I-类型	op	rs	rt	immediate
lui	001111	00000	rt	immediate

汇编举例：LUI $2, 10
功能描述:: (rt)←immediate<<16 & 0FFFF0000H 即(rt)←immediate×65536; PC ← NPC (PC + 4)。首先 16 位立即数赋给 rt 寄存器的高 16 位，低 16 位用 0 填充。也就是将 16 位立即数乘以 65536 后赋值给 rt 寄存器。

- lw

指令名：存储器读（字操作）
汇编格式：LW rt, offset(rs)
编码格式：

I-类型	op	rs	rt	immediate
lw	100011	rs	rt	offset

汇编举例：LW $3, 10($2) 或 LW $3, buff($2)
功能描述：(rt)←Memory[(rs)+(sign_extend)offset]; PC ← NPC (PC + 4)。以 rs 寄存

器的内容为基地址，offset 通过符号扩展后形成 32 位的偏移，将基地址加上偏移形成一个 32 位的地址，以此地址从 RAM 中读出一个字（4 字节）赋给 rt 寄存器。本系统中只使用了低 16 位地址，汇编中，offset 可以是变量名。

- sw

指令名：存储器写（字操作）
汇编格式：SW rt, offset(rs)
编码格式：

I-类型	op	rs	rt	immediate
sw	101011	rs	rt	offset

汇编举例：SW $3, 10($2)
功能描述：Memory[(rs)+(sign_extend)offset]←(rt); PC ← NPC (PC + 4)。以 rs 寄存器的内容为基地址，offset 通过符号扩展后形成 32 位的偏移，将基地址加上偏移形成一个 32 位的地址，将 rt 寄存器的内容写入到 RAM 中该地址开始的一个字（4 字节）单元。本系统中只使用了低 16 位，汇编中，offset 可以是变量名。

- beq

指令名：相等则转移指令
汇编格式：BEQ rt, rs, immediate
编码格式：

I-类型	op	rs	rt	immediate
beq	000100	rs	rt	offset

汇编举例：BEQ $3, $2, 10
功能描述：if ((rt)=(rs)) then (PC)←(PC)+4+((Sign-Extend) offset<<2); else PC ← NPC (PC + 4)。如果 rt 和 rs 的值相等，则转移到新的地址。新地址是当前指令的下一条指令地址（PC+4）加上一个 32 位偏移量。该 32 位偏移量是将 16 位 offset 符号扩展到 32 位，然后左移 2 位（即乘 4）后取低 32 位得到。实际系统中只用了低 16 位地址线。

- bne

指令名：不相等则转移指令
汇编格式：BNE rt, rs, immediate
编码格式：

I-类型	op	rs	rt	immediate
bne	000101	rs	rt	offset

汇编举例：BNE $3, $2, 10
功能描述：if ((rt)≠(rs)) then (PC)←(PC)+4+((Sign-Extend) offset<<2); else PC ← NPC (PC + 4)。如果 rt 和 rs 的值不等，则转移到新的地址。新地址是当前指令的下一条指令地址（PC+4）加上一个 32 位偏移量。该 32 位偏移量是将 16 位 offset 符号扩展到 32 位，然后左移 2 位（即乘 4）后取低 32 位得到。实际系统中只用了低 16 位地址线。

- bgtz

指令名：大于 0 转移指令
汇编格式：BGTZ rs, offset
编码格式：

I-类型	op	rs	rt	immediate
beq	000111	rs	00000	offset

汇编举例：BGTZ $1, 40

功能描述：if ((rs)>0) then (PC)←(PC)+4+((Sign-Extend) offset<<2), rs=$1; else PC ← NPC (PC + 4)。如果 rs 的值大于 0 则跳转到指定分支。

3.J 型指令

- j

指令名：无条件转移指令

汇编格式：J target

编码格式：

J-类型	op	address
j	000010	address

汇编举例：J 100

功能描述：(PC)←((Zero-Extend) address<<2); PC ← NPC (PC + 4)。无条件转移到新的地址。新地址是 26 位 address 零扩展到 32 位，然后左移 2 位（即 乘 4）后取低 32 位得到。实际系统中只用了低 16 位地址线。

- jal

指令名：过程调用指令

汇编格式：JAL target

编码格式：

J-类型	op	address
jal	000011	address

汇编举例：JAL 100

功能描述：($31)←(PC)+4; (PC)←((Zero-Extend)address<<2); PC ← NPC(PC+4)。先将下条指令的地址（(PC)+4）保存在$31（$ra）作为过程的返回地址，然后无条件转移到新的地址。新地址是 26 位 address 零扩展到 32 位，然后左移 2 位（即乘 4） 后取低 32 位得到。实际系统中只用了低 16 位地址线。

多周期 CPU 模块功能：

cpuclk 模块：

该模块负责将 Minisys 的 100MHz 的时钟频率进行分频，以产生能让多周期 CPU 正常运行的时钟频率；

PC 模块：

该模块负责存储下一条要执行指令的地址，一般情况下在第一个周期的开始 PC 存储着当前指令的地址，在第一个周期内将该指令从 IM 指令存储器中取出并存储到 IR 寄存器中，接着 PC 就会更新为下一条指令的地址；

NPC 模块：

该模块接受 CU 控制器的控制信号、PC 传来的当前指令地址、RF 寄存器堆传来的数据、当前指令 IR 传来的立即地址，根据控制信号，正确计算并给出下一条指令的地址 npc；

IM 模块：

该模块是指令存储器，负责存储所有指令的机器码；

ifetc32 模块：

该模快是 PC 模块、NPC 模块、IM 模块的整合，作为多周期 CPU 中的取指单元，负责按照程序里设计好的顺序取出正确的指令，并将取出的指令存放在 IR 寄存器中供 CPU 其它部分分析并执行相应操作；

instruction 模块：

该模块将传入的 IR 寄存器中的指令码切分成各种信号，例如一个 R 型指令中的 op,rs,rt,rd,shamt,func 字段就会被分别给出，方便后续模块使用；

MUX5 模块：

该模块就是一个 5 位信号的 3 选 1 的多路选择器，接受 3 个 5 位的信号，根据控制信号输出其中的一个；

MUX32 模块：

该模块就是一个 32 位信号的 3 选 1 的多路选择器，接受 3 个 32 位的信号，根据控制信号输出其中的一个；

registerfile 模块：

该模快是 CPU 中的寄存器堆,代表了 MIPS 指令系统中 32 个 32 位的通用寄存器,在 RFWr 等于 1 和有效时钟沿到来时将端口 WD 中的数据写到相应的寄存器中，通过给 rd1 和 rd2 端口给寄存器的编码也可读出相应寄存器中的值；

extend 模块：

该模块根据 CU 传来的控制信号,对当前指令 IR 中的立即数部分进行零扩展或符号扩展成 32 位信号；

idecode32 模块：

该模块整合了 instruction 模块、MUX5 模块、MUX32 模块、registerfile 模块、extend 模块，是多周期 CPU 中的译码单元，负责将取得的指令进行译码以及对寄存器堆 RF 进行相应操作；

ALU 模块：

该模块负责对输入进来的三个数据 A、B、C 根据传入的控制信号进相应运算操作，其中 A 和 B 是 32 位，C 则是 5 位，传给 A 和 B 都是直接参与运算的值，传给 C 都是移位运算需要移动的位数，ALU 将给出运算的结果以及根据运算的结果给出相应的标志 less,equal,more 传回 CU；

executs32 模块：

该模块整合了 MUX5、MUX32、ALU 模块，是 CPU 中的执行单元，根据控制信号以及相应的数据完成对应的运算并传出运算的结果和相应标志位；

CU 模块：

该模块是CPU中的控制器,控制器中设计了5个状态代表执行一条指令最多的5个周期，并根据当前指令的OP或FUNC字段的值以及当前所处的状态给出相应的控制信号来控制CPU中其它部件正确地执行操作；

judge 模块：

该模块主要是负责判断 BEQ,BNE,BGTZ 三条条件跳转指令地跳转条件成立与否，例如若此时是 BEQ，且运算器 ALU 传回的 equal 信号为 1，则代表此时跳转条件成立，因此输出 br_true 信号为 1；

control32 模块：

该模块整合了 CU 模块，judge 模块，是 CPU 中的控制单元，负责给出正确的控制信号来使整个 CPU 正确有序地执行操作；

dmemory32 模块：

该模块是 CPU 中的数据存储器 RAM；

设计的主要特色

1.可运行 31 条 MIPS 指令；

2.采用多周期的模式实现 CPU，使复杂度不同的指令的执行周期数不同，能够使大量使用简单指令的程序运行得更快；

3.CPU 的结构采用哈佛结构，具有独立的指令存储器 ROM 和数据存储器 RAM；

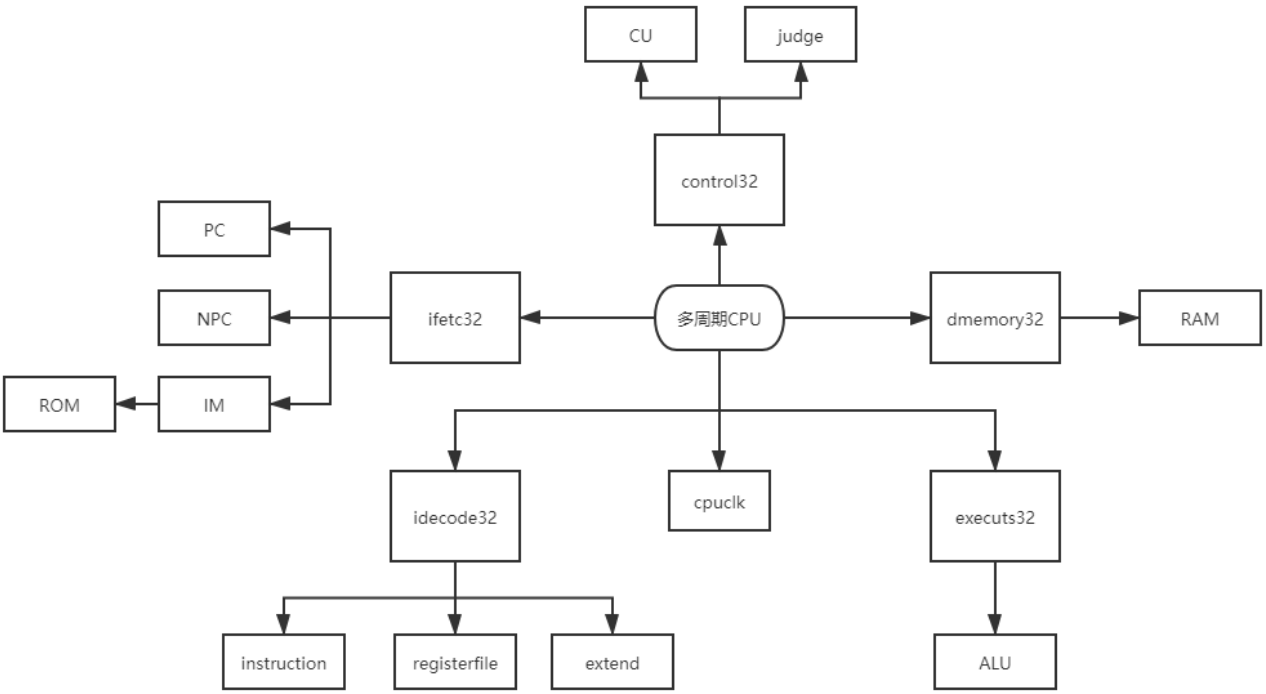
4.CPU 内部有分频结构，能够将 Minisys 实验板平台的时钟源信号的 100MHz 频率降低至 23MHz；

5.设计的简易汇编器能将实验一汇编实验的要求的23条MIPS指令汇编为对应机器码，且能够成功在 logisim 模型上运行实验一中的跑马灯等汇编程序；

设计的体系结构

（包括体系结构框图和对结构图的简要解释）

体系结构框图如下：



多周期 CPU 由六大模块组成，分别是控制模块 control32、取指模块 ifetc32、译码模块 idecode32、执行模块 executs32、数据存储器模块 dmemory32，时钟分频模块 cpuclk 组成；

其中控制模块 control32 中含有控制器模块 CU、三条条件跳转指令(beq,bne,bgtz)条件判断模块 judge，CU 负责多周期 CPU 中的周期状态跳转和输出控制信号到其它部件，judge 模块给出的 br_true 信号可以让 NPC 模块知道条件跳转的条件是否成立；

取指模块 ifetc32 含有指令地址寄存器 PC、下条指令地址计算部件 NPC、指令存储器 IM，其中 PC 负责存储下条指令的地址，NPC 负责计算出下条指令的地址，IM 采用 ROM 存储程序指令的机器码；

译码模块 idecode32 含有指令字段切分部件 instruction、通用寄存器堆 registerfile、位数扩展部件 extend,其中 instruction 就是将当前指令 IR 中的各个字段的信号取出来分别输出，registerfile 则是 MIPS 指令系统中的 32 个 32 位的通用寄存器，extend 则是根据控制信号进行零扩展或符号扩展的部件；

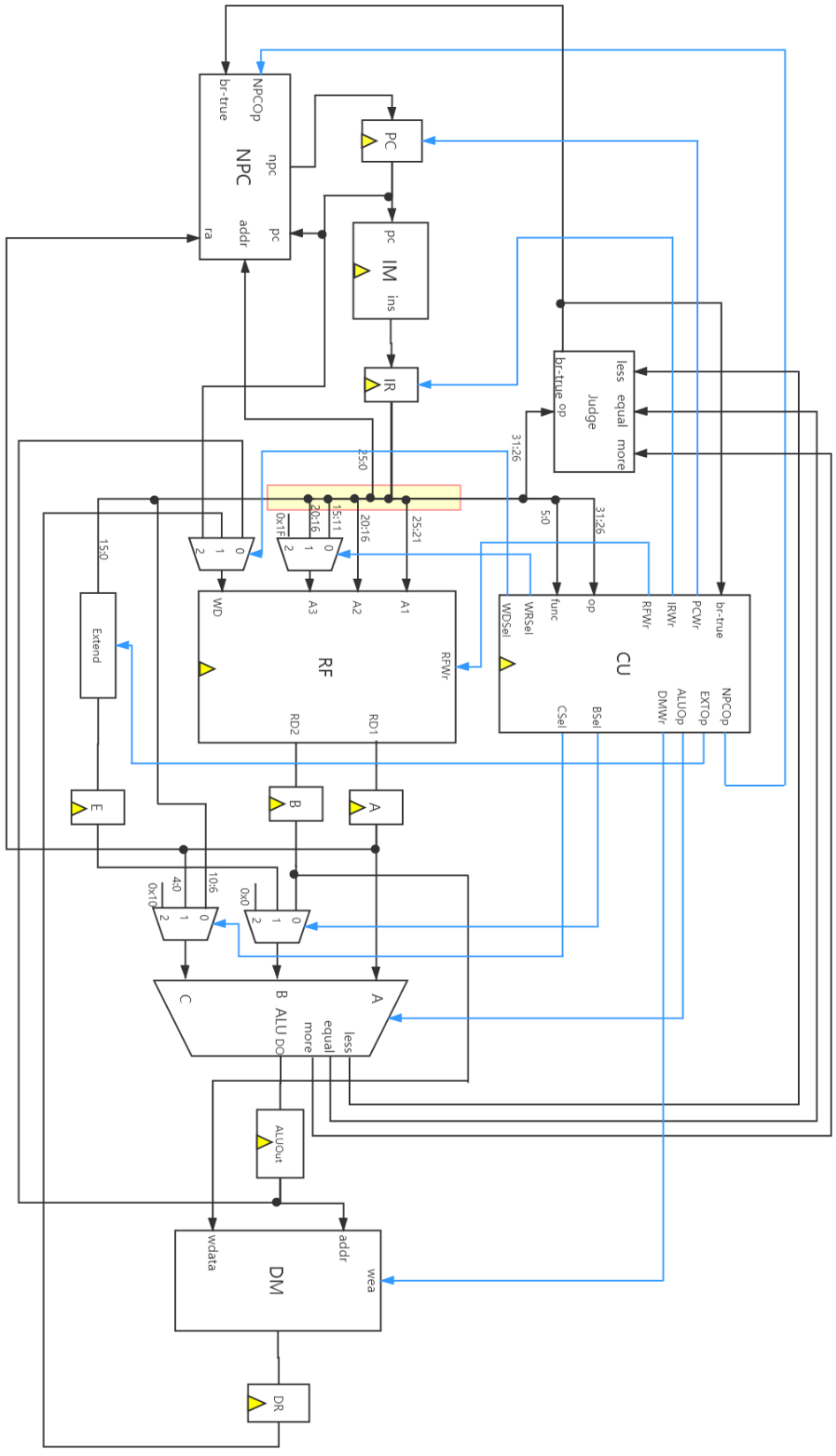
执行模块 executs32 含有 ALU 运算器，是整个 CPU 中对操作数进行运算的部件，除了一般的加减运算、位运算以外还可以进行移位运算，并且能够根据运算的结果输出相应标志位到控制器 CU 中以便产生相应控制信号；

数据存储器模块 dmemory32 采用 RAM 来存储数据（本 CPU 采用数据存储和指令存储分离的哈佛结构）；

分频模块 cpuclk 则是将外界的过高频率的时钟信号进行分频输出频率合适的时钟信号来控制整个 CPU 的正常运转；

数据通路设计

（包含数据通路主要部件说明及数据通路表）



主要部件说明：

PC：

存放下条指令地址的部件，PC 输入的是来自 NPC 部件的 npc；

NPC：

计算下条指令地址的部件，输入的有三种数据信号，一个标志信号，三个数据信号分别是来自 PC 部件的 pc，来自当前指令寄存器 IR 的 addr，来自寄存器堆 RF 的 ra 信号；标志信号是来自 Judge 部件的 br_true 代表条件跳转指令的条件成立与否；NPC 输出的是代表下条指令地址的 npc；

IM：

指令存储器，输入的是指令的地址 pc，输出的是这个地址上指令的机器码 ins；

IR：

当前指令寄存器，存放当前指令机器码的部件；

RF：

存放 32 个通用寄存器值的寄存器堆，输入端口 A1 是第一个寄存器的地址，将会在输出端口 RD1 给出这个寄存器的值，同理 A2 的地址将会在 RD2 给出其值；A3 是写入寄存器时要写入的寄存器的地址，WD 则是要写入的数据值；

A：

存放寄存器堆输出端口 RD1 的值的寄存器；

B：

存放寄存器堆输出端口 RD2 的值的寄存器；

Extend：

将指令中立即数部分进行扩展成 32 位信号的部件，扩展方式有零扩展和符号扩展两种，其输入是来自 IR 寄存器的[15:0]位；

E：

存放 Extend 输出的寄存器；

ALU：

负责进行运算的部件，其运算功能由 CU 给出的 ALUOp 控制信号决定；有三个输入端口 A,B,C，其中 A 和 B 都是 32 位的数据输入端口，C 是 5 位的位移量输入端口；

ALUOut：

存放 ALU 输出结果的寄存器；

DM：

数据存储器，输入端口 addr 输入的是数据的地址，输入端口 wdata 则是在写入数据时要写入的数据值；

DR：

存放数据存储器的读取结果的寄存器；

数据通路表：

见本报告末尾**附录 A.数据通路表**

各部件的设计与实现

（含模块功能，输入、输出信号及变量定义，关键代码等。控制器设计包含控制信号表）

1.PC

1)模块功能：

负责存储下一条要执行的指令的地址；在 PCWr=1 以及时钟**下降**沿时将存储的值更新为输入端口 di 的值；

2)输入信号：

input clk：时钟信号；

input reset：复位信号；

input PCWr：PC 写入信号，为 1 时有效；

input [31:0] di：输入的地址值，代表下一次 PC 要更新的值；

3)输出信号：

output reg [31:0] do：输出的地址值，代表当前 PC 寄存器存储的地址；

4)变量定义：

reg has_started：该变量用于表示 CPU 运行是否已经开始；之所以设置这个变量的原因是在复位信号刚消失时为了保证让第一条指令的执行时间是完整的，因此在面对第一个时钟有效沿以及 PCWr=1 时应该让 PC 先不变化到下一条，而是继续执行原先的指令好让该指令的执行时间是完整的；

5)关键代码：

```
always@(negedge clk)begin
    if(reset==1'b1)begin
        has_started <= 1'b0;
        do <= 32'd0;
    end else begin
        if(has_started == 1'b0 && PCWr == 1'b1)begin
            has_started <= 1'b1;
        end else if(has_started == 1'b1 && PCWr == 1'b1) begin
            do <= di;
        end else begin
            do <= do;
        end
    end
end
```

2.NPC

1)模块功能：

负责计算出下一条要执行的指令的地址；

2)输入信号：

input [31:0] pc：PC 输入的地址，代表当前指令的地址；

input [25:0] addr：当前指令的 addr 部分(25:0 位)，执行 J，JAL 指令会将该值写到 PC；

input [31:0] ra：寄存器堆 RF 读出的 RD2 的值，执行 JR 指令时会将该值写到 PC；

input [1:0] NPCOp：控制器 CU 给出的控制信号，负责控制下一条指令的地址应该如何计算；

input br_true：外界传入的标志，代表 beq,bne,bgtz 三类指令跳转条件成立与否,成立则为 1；

3)输出信号：

output reg [31:0] npc：计算出来的下条指令的地址；

4)变量定义：

wire [31:0] offset：在执行 beq,bne,bgtz 三种指令时代表当前指令和下条指令地址偏移量(是有符号数)；

wire [31:0] b_imm：执行 beq,bne,bgtz 三种指令时要跳转到的地址；

wire [31:0] j_imm：执行 j,jal 三种指令时要跳转到的地址；

5)关键代码：

```
always@(*)begin
    case(NPCOp)
        `PC4:begin
            npc=pc+4;
        end
        `JR:begin
            npc=ra;
        end
        `BT:begin
            npc=br_true==1'b1 ? b_imm : pc+4;
        end
        `J:begin
            npc=j_imm;
        end
    endcase
end
```

3.IM

1)模块功能：

负责存储程序各条指令的机器码;在给出的地址后在下一个时钟有效沿时即可读出给定地址上的指令的机器码；

2)输入信号：

input clock：时钟信号；

input [31:0] PC：PC 给出的当前指令的地址；

3)输出信号：

output [31:0] Instruction：给定地址上的指令的 32 位机器码；

4)变量定义：

wire clk：在本 CPU 中设计的是 IM 在时钟下降沿更新出对应地址上的指令机器码，而本次实验采用的 ROM 的 IP 核是 BlcokMemory，在时钟上升沿时给出新读出的值，因此需要将外部传入的时钟信号反向，因此设置一个反向的时钟信号 clk；

5)关键代码：

```
//分配 64KB ROM,
prgrom instmem(
    .clka(clk), // input wire clka
    .addra(PC[15:2]), // input wire [13 : 0] addra
    .douta(Instruction) // output wire [31 : 0] douta
```

```
);
```

4.instruction

1)模块功能：

负责将传进来的 32 位指令切分出各段信号，方便 CPU 其它部件使用；

2)输入信号：

input [31:0] ins：从 IR 传入的 32 位当前指令机器码；

3)输出信号：

output [5:0] op：指令的 op 字段；

output [4:0] rs：指令的 rs 字段，代表源寄存器的地址；

output [4:0] rt：指令的 rt 字段；

output [4:0] rd：指令的 rd 字段，代表目的寄存器的地址；

output [4:0] shamt：srl,sll,sra 指令中代表移位的位数的字段；

output [5:0] func：指令的 func 字段，用于区分 17 条 R 型指令；

output [15:0] imm：指令的立即数部分；

output [25:0] addr：J 型指令中要跳转的地址；

4)关键代码：

```
assign op=ins[31:26];
assign rs=ins[25:21];
assign rt=ins[20:16];
assign rd=ins[15:11];
assign shamt=ins[10:6];
assign func=ins[5:0];
assign imm=ins[15:0];
assign addr=ins[25:0];
```

5.MUX5

1)模块功能：

5 位信号的 3 选 1 多路选择器；

2)输入信号：

input [4:0] input1：输入的第一个信号；

input [4:0] input2：输入的第二个信号；

input [4:0] input3：输入的第三个信号；

input [1:0] sel：来自 CU 的选择控制信号，决定输出信号的来源；

3)输出信号：

output reg [4:0] out：3 选 1 选择出的输出信号；

4)关键代码：

```
always@(*)begin
    case(sel)
    2'b00:out=input1;
    2'b01:out=input2;
    default:out=input3;
    endcase
end
```

6.MUX32

1)模块功能：

32 位信号的 3 选 1 多路选择器；

2)输入信号：

input [31:0] input1：输入的第一个信号；

input [31:0] input2：输入的第二个信号；

input [31:0] input3：输入的第三个信号；

input [1:0] sel：来自 CU 的选择控制信号，决定输出信号的来源；

3)输出信号：

output reg [31:0] out：3 选 1 选择出的输出信号；

7.registerfile

1)模块功能：

CPU 中的寄存器堆，是 MIPS 指令系统中 32 个通用寄存器组成的寄存器阵列，用于在内存与 CPU 运算部件之间暂存数据；

2)输入信号：

input clk：时钟信号；

input rst：复位信号；

input [4:0] A1：第一个要读出的寄存器的地址；

input [4:0] A2：第二个要读出的寄存器的地址；

input [4:0] A3：要写入的寄存器的地址；

input [31:0] WD：要写入寄存器的值；

input RFWr：寄存器堆写入信号；

3)输出信号：

output reg [31:0] RD1：第一个要读出的寄存器的值；

output reg [31:0] RD2：第二个要读出的寄存器的值；

4)变量定义：

reg[31:0] regfile[31:0]：32 个 32 位通用寄存器阵列；

5)关键代码：

```
always@(*)begin
    RD1=regfile[A1];
    RD2=regfile[A2];
end
integer i;
always@(negedge clk)begin
    if(rst == 1'b1)begin
        for (i = 0; i < 32; i = i+ 1) regfile[i] <= 0;
    end else if(RFWr == 1'b1 && A3)begin
        regfile[A3]<=WD;
    end
end
```

8.extend

1)模块功能：

将指令中的 16 位立即数 imm 扩展为 32 位数据，扩展方式有零扩展和符号扩展；
2)输入信号：
input [15:0] Imm：当前执行指令 IR 中的低 16 位，即指令中的立即数部分；
input EXTOp：CU 传来的控制信号，0 是符号扩展，1 是零扩展；
3)输出信号：
output [31:0] Ext：将输入 imm 扩展成的 32 位信号；
4)关键代码：

```
assign Ext[15:0] = Imm;
assign Ext[31:16] = EXTOp==`SIGN ? (Imm[15]==1'b1 ? 16'hffff : 16'h0000) : 16'h0000;
```

9. ALU

1)模块功能：
CPU 中对数据进行运算的部件，负责实现多组算术运算和逻辑运算；
2)输入信号：
input [31:0] A：参与运算的第一个操作数；
input [31:0] B：参与运算的第二个操作数；
input [4:0] C：移位运算要用到的移位的位数；
input [3:0] ALUOp：CU 传来的控制信号，控制 ALU 执行不同的运算；
3)输出信号：
output reg [31:0] DO：运算的结果；
output equal：传回给 CU 的标志位，如果 DO=0 则为 1，否则为 0
output less：传回给 CU 的标志位，如果 DO<0 则为 1，否则为 0
output more：传回给 CU 的标志位，如果 DO>0 则为 1，否则为 0
4)关键代码：

```
assign equal= DO == 32'd0 ? 1'b1:1'b0;
assign less= DO[31]==1 ? 1'b1:1'b0;
assign more= DO[31]==0 ? 1'b1:1'b0;
always@(*)begin
    case(ALUOp)
    `ADD:begin DO=A+B;end
    `SUB:begin DO=A-B;end
    `AND:begin DO=A&B;end
    `OR:begin DO=A|B;end
    `XOR:begin DO=A^B;end
    `NOR:begin DO=~(A|B);end
    `SLT:begin
        if(A[31]==1 && B[31]==0)begin
            DO=32'd1;
        end else if(A[31]==0 && B[31]==1)begin
            DO=32'd0;
        end else if(A[30:0]<B[30:0])begin
            DO=32'd1;
        end else begin
            DO=32'd0;
```

```
        end
    end
    `SLTU:begin DO= A<B ? 32'd1:32'd0;end
    `SLL:begin DO=B<<C;end
    `SRL:begin DO=B>>C;end
    `SRA:begin DO=($signed(B))>>>C;end
    endcase
end
```

10. CU

1)模块功能：

CPU 中的控制单元，其中设置有状态机根据当前所执行的指令以及当前所处的状态来进行状态的跳转， 同时也会根据当前状态和当前执行的指令给出相应的控制信号到 CPU 中其它部件控制整个 CPU 正确执行操作；

2)输入信号：

input clk：时钟信号；

input rst：复位信号；

input br_true：judge 模块传来的标志信号，为 1 代表条件跳转指令(BEQ,BNE,BGTZ)的跳转条件成立，CU 这时就可以 PCWr 打开，将新地址写入 PC，否则 PCWr 不打开；

input [5:0] op：当前执行指令 IR 中的 op 字段；

input [5:0] func：当前执行指令 IR 中的 func 字段；

3)输出信号：

output reg [1:0] NPCOp：控制 NPC 部件产生对应的下条指令的地址，具体宏定义有 PC4 代表下条指令地址就是 PC+4，JR 代表下条指令地址来自寄存器堆的数据，BT 代表下条指令地址是当前 PC 加上偏移量 offset，J 代表下条指令地址来自 IR 中的 addr 字段；

output reg PCWr：控制 PC 寄存器写入的控制信号，为 1 代表可以写入；

output reg IRWr：控制 IR 寄存器写入的控制信号，为 1 代表可以写入；

output reg RFWr：控制 RF 写入数据的控制信号，为 1 代表可以写入；

output reg EXTOp：控制 EXT 进行符号扩展还是零扩展的控制信号，为 0 代表进行符号扩展，为 1 代表零扩展；

output reg [3:0] ALUOp：控制 ALU 执行相应的运算操作；

output reg DMWr：控制 DM 数据存储器写入的控制信号，为 1 代表可以写入；

output reg [1:0] WRSel：RF 的 A3 端口外的三选一多路选择器的选择信号；

output reg [1:0] WDSel：RF 的 WD 端口外的三选一多路选择器的选择信号；

output reg [1:0] BSel：ALU 的 B 端口外的三选一多路选择器的选择信号；

output reg [1:0] CSel：ALU 的 C 端口外的三选一多路选择器的选择信号；

4)变量定义：

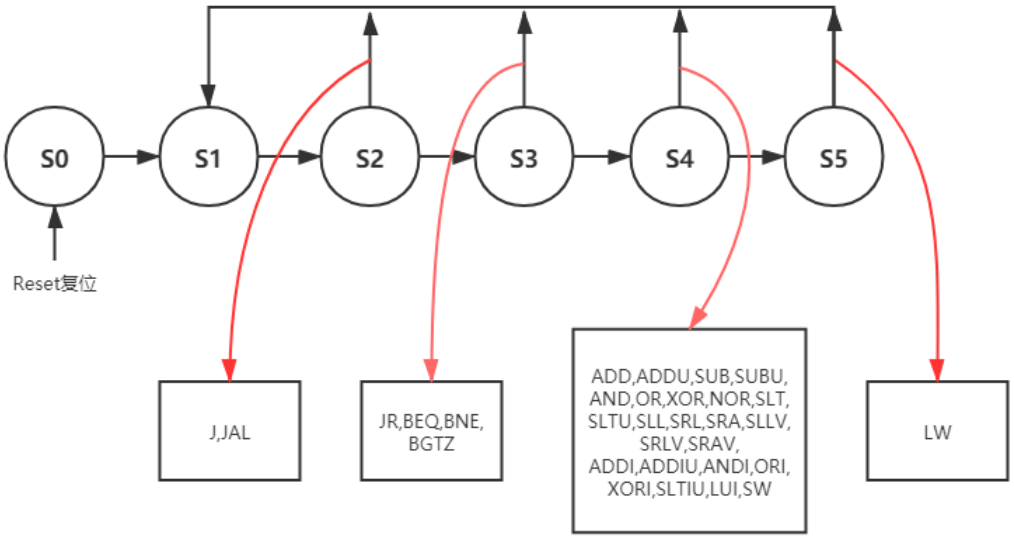
parameter [2:0] s1 = 3'b000,s2=3'b001,s3=3'b010,s4=3'b011,s5=3'b100,s0 = 3'b111;

上面的变量是定义的 6 个变量，其中 s0 是 CPU 刚启动时的状态，其余的 5 个状态则分别代表了一条指令运行时的 5 个周期；

wire T1,T2,T3,T4,T5：5 个周期信号，代表当前处于哪个周期，例如 T1 为 1 则代表当前处在指令执行的第一个周期；

reg [2:0] state, nextState：状态变量，分别是当前所处状态和次态；

5)**状态转换图**：



6)关键代码：

```
assign T1=(state == s1);
assign T2=(state == s2);
assign T3=(state == s3);
assign T4=(state == s4);
assign T5=(state == s5);
```

状态跳转的代码：

```
always@(posedge clk) begin
   if(rst) begin
       state <= s0;
   end else begin
       state <= nextState;
   end
end
```

计算次态的代码：

```
always@(*)begin
    case(state)
        s0:begin
            nextState=s1;
        end
        s1:begin
            nextState=s2;
        end
        s2:begin
            if(op == `J_OP || op == `JAL_OP)begin
                nextState=s1;
            end else begin
                nextState=s3;
            end
```

```
        end
        s3:begin
            if((op == 6'b000000 && func == `JR_FUNC) || op == `BEQ_OP || op == `BNE_OP
|| op == `BGTZ_OP) begin
                nextState=s1;
            end else begin
                nextState=s4;
            end
        end
        s4:begin
            if(op == `LW_OP)begin
                nextState=s5;
            end else begin
                nextState=s1;
            end
        end
        s5:begin
            nextState=s1;
        end
    endcase
end
```

控制信号输出的代码(部分)：

```
case(func)
    `ADD_FUNC:begin
        NPCOp = `PC4;
        PCWr = T1 ? 1'b1 : 1'b0;
        IRWr = T1 ? 1'b1 : 1'b0;
        RFWr = T4 ? 1'b1 : 1'b0;
        ALUOp = `ADD;
        DMWr = 1'b0;
        WRSel = `RD;
        WDSel = `ALUDO;
        BSel = `RD2;
```

6)**控制信号表：**

见本报告末尾**附录 B.控制信号表**

11. judge

1)模块功能：

负责判断 BEQ,BNE,BGTZ 三条条件跳转指令地跳转条件成立与否，例如若此时是 BEQ，且运算器 ALU 传回的 equal 信号为 1，则代表此时跳转条件成立，因此输出 br_true 信号为 1；

2)输入信号：

input [5:0] op：当前执行指令 IR 中的 op 字段，用于判断当前指令是否是 BEQ,BNE,BGTZ；

input equal：ALU 传出的标志位，代表当前两操作数是否相等，相等则为 1；

input less：ALU 传出的标志位，代表两操作数想减的和是否小于 0，小于 0 则为 1；

input more：ALU 传出的标志位，代表两操作数想减的和是大于 0，大于 0 则为 1；
3)输出信号：
output reg br_true：代表条件跳转指令的条件成立与否，成立则为 1，否则为 0；
4)关键代码：

```
always@(*)begin
    case(op)
        `BEQ_OP:begin
            br_true= equal==1'b1 ? 1'b1:1'b0;
        end
        `BNE_OP:begin
            br_true= equal==1'b1 ? 1'b0:1'b1;
        end
        `BGTZ_OP:begin
            br_true= more==1'b1 ? 1'b1:1'b0;
        end
    endcase
end
```

12.ifetc32

1)模块功能：
CPU 中的取指单元，负责取指周期的操作，具体是在指令的执行第一个周期内完成取出新指令，更新 PC，将新指令存入 IR 寄存器；
具体实现则是对 PC,NPC,IM 三个模块以及 IR 寄存器的整合；
2)输入信号：
input clk：时钟信号；
input reset：复位信号；
input br_true：条件跳转指令的条件成立与否的标志信号；
input [25:0] addr：当前执行指令中的 addr 字段；
input [31:0] ra：寄存器堆读出的 RD1 数据；
input [1:0] npcop：CU 给出的对 NPC 下条指令地址的计算进行控制的信号；
input IRWr：CU 给出的对 IR 寄存器写入的控制信号，为 1 时代表可以写入；
input PCWr：CU 给出的对 PC 部件写入新 PC 的控制信号，为 1 时代表可以写入；
3)输出信号：
output reg [31:0] IR：保存的当前执行指令的机器码；
output [31:0] now_pc：当前 PC 部件中存储的地址；
4)变量定义：
wire [31:0] IM_DO：IM 中 RAM 的 read_data 值；
wire [31:0] npc_out：NPC 到 PC 的下条指令地址的信号线 wire；
5)关键代码：

```
always@(posedge clk)begin
    IR<=IRWr?IM_DO:IR;
end
PC u_PC(.clk(clk),.reset(reset),.PCWr(PCWr),.di(npc_out),.do(now_pc));

NPC u_NPC(.pc(now_pc),.addr(addr),.ra(ra),.NPCOp(npcop),.br_true(br_true),.npc(npc_out));

programrom IM(.clock(clk),.PC(now_pc),.Instruction(IM_DO));
```

13.idecode32

1)模块功能：

该模块主要进行指令执行的译码工作，即将寄存器操作数取出到寄存器 A,B，将立即数按控制信号进行扩展后存入寄存器 E，因为其中包含寄存器堆 registerfile，因此也负责维护寄存器堆的工作；

2)输入信号：

input clk：时钟信号；

input rst：复位信号；

input [31:0] ins：当前指令寄存器 IR 传来的指令机器码；

input [31:0] ALU_out：ALU 运算部件的运算结果；

input [31:0] PC_DO：PC 部件保存的 PC 值；

input [31:0] DM_RD：DR 寄存器保存的数据存储器读出的值；

input RFWr：CU 发出的控制 RF 写入的信号；

input EXTOp:CU 发出的控制 extend 进行零扩展还是符号扩展的信号；

input [1:0] WRSel：CU 发出的对寄存器堆输入端口 A3 外连接的三选一选择器的选择信号；

input [1:0] WDSel：CU 发出的对寄存器堆写入数据端口 WD 外连接的三选一选择器的选择信号；

3)输出信号：

output [4:0] shamt：当前指令 IR 中的 shamt 字段，代表移位的位数；

output [5:0] op：当前指令 IR 中的 op 字段，操作码；

output [5:0] func：当前指令 IR 中的 func 字段，操作码；

output [25:0] addr：当前指令 IR 中的 addr 字段；

output reg [31:0] A：存储寄存器堆 RD1 端口读出的值；

output reg [31:0] B：存储寄存器堆 RD2 端口读出的值；

output reg [31:0] E：存储位数扩展 extend 部件输出的值；

4)变量定义：

wire [31:0] rf_rd1,rf_rd2：连接寄存器堆到寄存器 A,B 的连线；

wire [31:0] ext：连接位数扩展部件到寄存器 E 的连线；

wire [4:0] rs,rt,rd,wire_A3：连接不同部件之间的连线；

wire [15:0] imm：连接 ins 输出端口 imm 的连线；

wire [31:0] wire_WD：连接 MUX32 多选器的输出到寄存器堆输入端口 WD 的连线；

5)关键代码：

```
always@(posedge clk)begin
    A<=rf_rd1;
```

```
    B<=rf_rd2;
    E<=ext;
end
```

14. executs32

1)模块功能：

CPU 中的执行单元，整合了 ALU 和 ALU 输入端口外的三选一多路选择器，负责执行指令中需要运算的操作；

2)输入信号：

input clk：时钟信号；

input [31:0] rf_rd1：A 寄存器保存的值，即寄存器堆 RD1 端口读出的值；

input [31:0] rf_rd2：B 寄存器保存的值，即寄存器堆 RD2 端口读出的值；

input [31:0] ext：E 寄存器保存的值，即 extend 部件输出的结果；

input [4:0] shamt：当前指令的 shamt 字段，代表位移位数；

input [3:0] ALUOp：CU 传给 ALU 的控制信号，控制 ALU 执行相应运算；

input [1:0] BSel：CU 传给 ALU 的 B 输入端口外的多路选择器的选择信号；

input [1:0] CSel：CU 传给 ALU 的 C 输入端口外的多路选择器的选择信号；

3)输出信号：

output reg [31:0] ALUOut：ALU 运算结果；

output less：ALU 传回控制单元的标志位；

output equal：ALU 传回控制单元的标志位；

output more：ALU 传回控制单元的标志位；

4) 变量定义：

wire [31:0] ALU_out；

wire [31:0] wire_ALUB；

wire [31:0] wire_ALUC；

以上变量都是不同部件之间传递数据的数据线；

5)关键代码：

```
always@(posedge clk)begin
    ALUOut<=ALU_out;
end

MUX32 BMUX(.input1(rf_rd2),.input2(ext),.input3(32'd0),.out(wire_ALUB),.sel(BSel));

MUX5 CMUX(.input1(shamt),.input2(rf_rd1[4:0]),
.input3(5'b10000),.out(wire_ALUC),.sel(CSel));

ALU u_ALU(.A(rf_rd1),.B(wire_ALUB),.C(wire_ALUC),.ALUOp(ALUOp),
.DO(ALU_out),.equal(equal),.less(less),.more(more));
```

15.control32

1)模块功能：

CPU 中的控制单元，整合了 CU 和 judge 模块，是整个 CPU 中发出各种控制信号的部件；

2)输入信号：

input clk：时钟信号；
input rst：复位信号；
input [5:0] op：当前执行指令的 op 字段；
input [5:0] func：当前执行指令的 func 字段；
input less：ALU 传出的计算结果的标志位；
input equal：ALU 传出的计算结果的标志位；
input more：ALU 传出的计算结果的标志位；
3)输出信号：
output [1:0] NPCOp：控制 NPC 产生下条指令地址的计算方式；
output PCWr：控制 PC 寄存器的写入；
output IRWr：控制 IR 寄存器的写入；
output RFWr：控制寄存器堆的写入；
output EXTOp：控制位扩展部件的扩展方式；
output [3:0] ALUOp：控制 ALU 部件执行何种运算；
output DMWr：控制数据存储器的写入；
output [1:0] WRSel：控制寄存器堆写入寄存器地址的三选一选择器的选择信号；
output [1:0] WDSel：控制寄存器堆写入寄存器的数据的三选一选择器的选择信号；
output [1:0] BSel：控制 ALU 的 B 端口外的三选一选择器的选择信号；
output [1:0] CSel：控制 ALU 的 C 端口外的三选一选择器的选择信号；
output br_true：条件跳转的条件成立与否的标志信号；
4)关键代码：

```
CU u_CU(.clk(clk),.rst(rst),.br_true(br_true),.op(op),.func(func),
.NPCOp(NPCOp),.PCWr(PCWr),.IRWr(IRWr),.RFWr(RFWr),.EXTOp(EXTOp),
.ALUOp(ALUOp),.DMWr(DMWr),.WRSel(WRSel),.WDSel(WDSel),.BSel(BSel),.CSel(CSel));

judge u_judge(.op(op),.equal(equal),.less(less),.more(more),.br_true(br_true));
```

16.dmemory32
1)模块功能：
CPU 中的数据存储器；
2)输入信号：
input [31:0] address：要写入的存储单元的地址，来 ALUOut 寄存器；
input [31:0] write_data：要写入的数据，来自寄存器堆的 B 寄存器；
input Memwrite：来自 CU 的控制写入的信号；
input clock：时钟信号；
3)输出信号：
output reg [31:0] DR：数据存储器读出的数据；
4)变量定义：
wire [31:0] read_data：RAM 的 read_data 输出的数据；
wire clk：外界的时钟信号经过反相后的时钟信号；
5)关键代码：

```
always@(posedge clock)begin
    DR<=read_data;
end
```

```
ram ram (
    .clka(clk), // input wire clka
    .wea(Memwrite), // input wire [0 : 0] wea
    .addra(address[15:2]), // input wire [13 : 0] addra
    .dina(write_data), // input wire [31 : 0] dina
    .douta(read_data) // output wire [31 : 0] douta
);
```

设计主要测试结果（仿真截图或下载照片）

（自己实现的仿真部分代码及截图（至少包括除时钟和存储器外的 2 个模块），贴主要说明问题的时序图并对时序进行分析，可以竖贴）

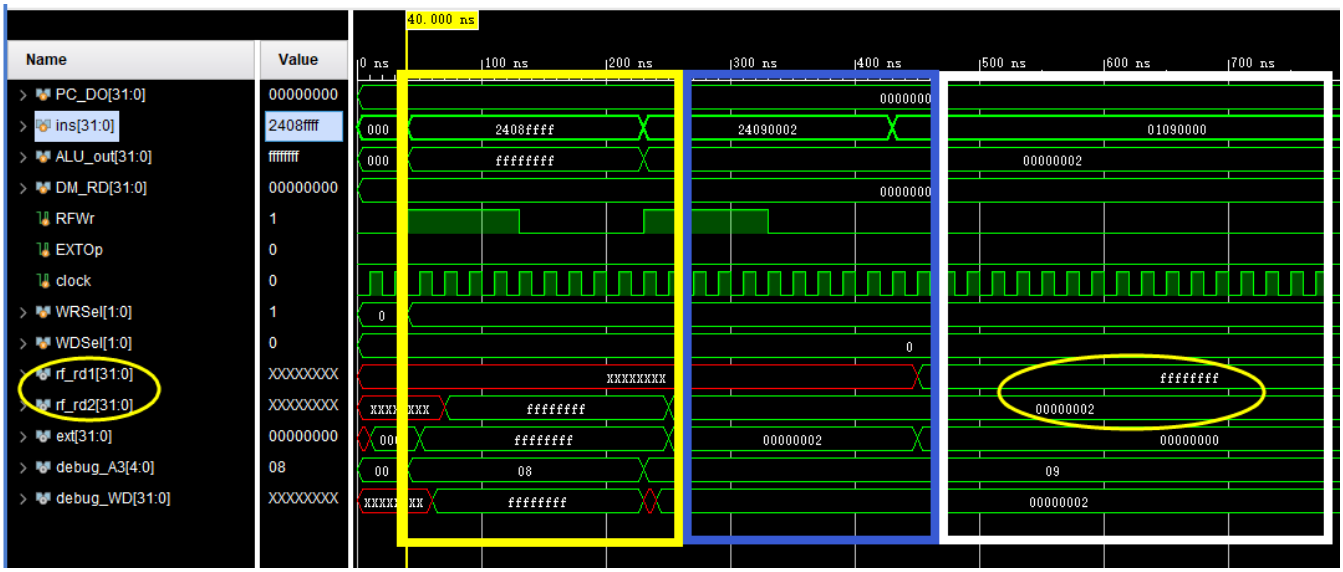
1.对译码单元 idecode32 进行仿真

仿真文件代码：

```
`timescale 1ns / 1ps
module idecode32_sim();
    reg [31:0] PC_DO = 32'h0;
    reg [31:0] ins = 32'h0;
    reg [31:0] ALU_out = 32'h0;
    reg [31:0] DM_RD = 32'h0;
    reg RFWr = 1'b0;
    reg EXTOp = 1'b0;
    reg clock = 1'b0;
    reg [1:0] WRSel = 2'b00;
    reg [1:0] WDSel = 2'b00;

    wire [31:0] rf_rd1,rf_rd2,ext;
    wire [4:0] debug_A3;
    wire [31:0] debug_WD;
    always #10 clock = ~clock;

    //实例化 idecode32
    idecode32 u_idecode32(.clk(clock),.ins(ins),.ALU_out(ALU_out),.PC_DO(PC_DO),.DM_RD(DM_RD),.RFWr(RFWr),
    .EXTOp(EXTOp),.WRSel(WRSel),.WDSel(WDSel),.A(rf_rd1),.B(rf_rd2),
    .E(ext),.debug_A3(debug_A3),.debug_WD(debug_WD));

    initial begin
        #40 begin   ins = 32'h2408ffff; //addiu    $t0,$zero,-1
                    ALU_out = 32'hffffffff;
                    RFWr = 1'b1;
                    WRSel = 2'b01;
                    WDSel = 2'b00;
            end
        #90 begin  RFWr = 1'b0;end
        #100 begin  ins = 32'h24090002; //addiu    $t1,$zero,2
                    ALU_out = 32'h00000002;
                    RFWr = 1'b1;
                    WRSel = 2'b01;
                    WDSel = 2'b00;
             end
        #100 begin  RFWr = 1'b0;end
        #100 begin  ins = {6'd0,5'd8,5'd9,16'd0}; //无意义语句，只把 t0 和 t1 分别读到 rd1 和 rd2 上
             end
    end
```

```
endmodule
```

仿真截图：



分析：

1)40ns 开始，执行“addiu　　$t0,$zero,-1”语句：

从图上黄色方框中可以看到，在该条指令给出后的第一个上升沿，经过符号扩展后的立即数-1 给到了位扩展部件 extend 的寄存器 E(即 idecode32 端口 ext)中，并在随后的下降沿将-1 写入寄存器堆中地址为 8 的寄存器中(从 debug_WD，debug_A3 信号可以看到)，在之后的第一个上升读出的数据才写入到连接到寄存器堆 RD2 外的寄存器 B 中(idecode32 的输出端口 rf_rd2)，rf_rd1 端口输出为-1，代表写入成功；

2)230ns 开始，执行“addiu　　$t1,$zero,2”语句：

从图上蓝色方框中可以看到，在该条指令给出后 debug_A3 则立即变成了 09，即$t1 的地址；并在随后的下降沿将 2 写入到寄存器堆中地址为 9 的寄存器组中，随后 rf_rd1 端口输出为 2，代表写入成功；

3)430ns 开始，执行机器码为{6'd0,5'd8,5'd9,16'd0}的语句：

这条语句除了 rs 字段为 8，rt 字段为 9 以外其它所有位上数字都是 0，因此该指令只是为了从寄存器堆中读出地址为 8 和为 9 上的寄存器的值，即$t0,$t1 中的值；

从图上白色方框中可以看到，在指令 ins 变为 01090000 后,rf_rd1 的值为-1，rf_rd2 的值为 2，即之前对寄存器堆写入的操作执行正确，成功将数据写入到正确的寄存器中；

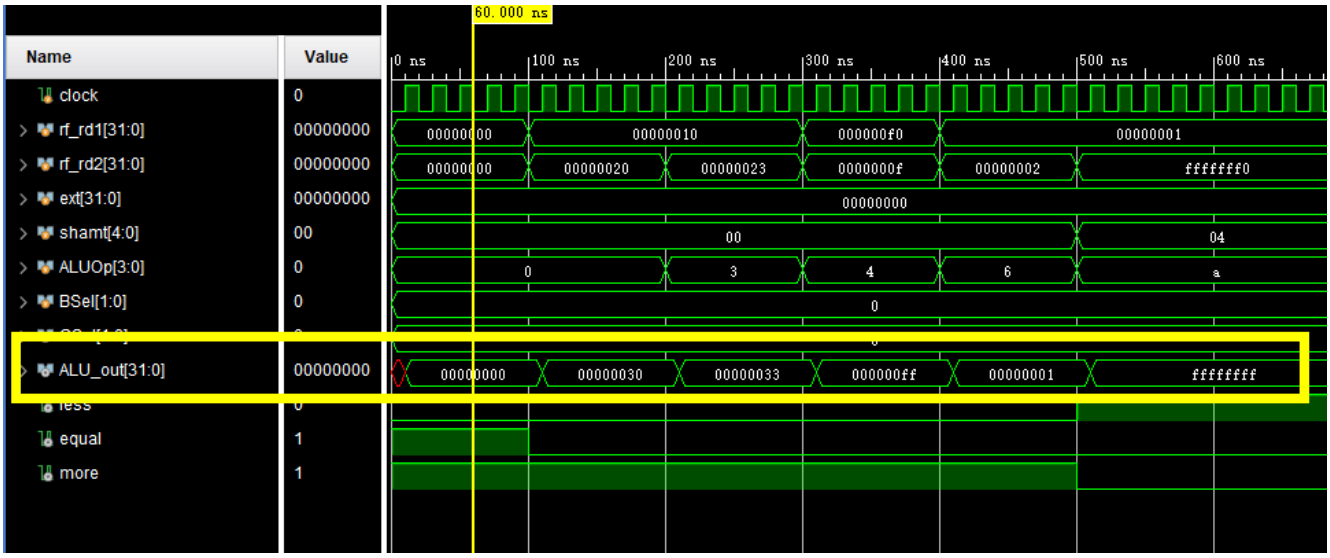
2.对执行单元 executs32 进行仿真

仿真文件代码：

```
`timescale 1ns / 1ps
module executs32_sim();
    reg clock = 1'b0;
    always #10 clock =~ clock;

    reg [31:0] rf_rd1 = 32'd0;
    reg [31:0] rf_rd2 = 32'd0;
    reg [31:0] ext = 32'd0;
    reg [4:0] shamt = 5'd0;
    reg [3:0] ALUOp = 4'd0;
    reg [1:0] BSel = 2'd0;
    reg [1:0] CSel = 2'd0;
    wire [31:0] ALU_out;
    wire less,equal,more;

    //实例化执行单元
    executs32 u_executs32(.clk(clock),.rf_rd1(rf_rd1),.rf_rd2(rf_rd2),.ext(ext),.shamt(shamt),
    .ALUOp(ALUOp),.BSel(BSel),.CSel(CSel),.ALUOut(ALU_out),.less(less),.equal(equal),.more(more));

    initial begin
        //先初始化，让 ALU 的 A 是 RD1，B 是 RD2，C 是 shamt，后续只需要操控这三个寄存器即可，不用去管 ext
        #10 begin BSel = 2'b00; CSel = 2'b00;end
        #90 begin   //先进行加
                rf_rd1 = 32'h10;
                rf_rd2 = 32'h20;
                ALUOp = 4'b0000;
            end
        #100 begin  //再进行或运算
                rf_rd1 = 32'h10;
                rf_rd2 = 32'h23;
                ALUOp = 4'b0011;
            end
        #100 begin  //再进行异或运算
                rf_rd1 = 32'hf0;
                rf_rd2 = 32'h0f;
                ALUOp = 4'b0100;
            end
        #100 begin  //再进行 SLT 运算
                rf_rd1 = 32'h1;
                rf_rd2 = 32'h2;
                ALUOp = 4'b0110;
            end
```

```
        #100 begin  //再进行算术右移运算
                rf_rd2 = 32'hfffffff0;
                shamt = 5'h4;
                ALUOp = 4'b1010;
            end
    end
endmodule
```

仿真截图：



分析：

1)100ns 时执行 0x10+0x20 的加法运算：

可以看到 ALUout 的结果为 0x30，计算正确；

2)200ns 时执行 0x10|0x23 的位或运算：

可以看到 ALUout 的结果 0x33，运算正确；

3)300ns 时执行 0xf0 和 0x0f 的 xor 运算：

可以看到 ALUout 的结果 0xff，运算正确；

4)400ns 时执行 SLT 1,2 运算：

因为 1<2，所以 ALUout 的结果为 1，运算正确；

5)500ns 时执行将 0xfffffff0 算术右移 4 位的运算：

结果为 0xFFFFFFFF，运算正确；

设计过程中遇到的问题及解决方法

（包括设计过程中的错误及测试过程中遇到的问题）

问题 1.CU 的状态转移问题

1)问题描述：

在最初的 CU 设计中，我只针对指令执行最多需要的周期数——5 个(LW 语句需要的周期最多)，设计了 5 个状态 s1-s5,这样当复位信号刚消失时状态是 s1,等到下一个有效沿时状态就成为了 s2，这样的问题就是第一条语句的第一个周期不是完整的，从而使语句执行出错；

2)解决办法：

比原来的 5 个状态多设计一个 s0 初始状态，这个状态不对产生控制信号有作用，只是在复位信号有效时将状态置为 s0，这样当第一个有效沿到来时，状态就会跳转到 s1，从而使第一条语句的第一个周期是完整的，这样就能成功执行第一条指令；

问题 2.JAL 语句测试失败

1)问题描述：

在使用老师提供的测试文件 newcpu_tb.v 对自己最初设计的 CPU 进行测试时，在 JAL 语句时会出现失败。

失败的原因主要是：自己设计的 debug_wb_rf_wdata 是取 A3 地址上的寄存器的值，因此只有将数据写入到 A3 地址上的寄存器后， debug_wb_rf_wdata 才是正确的，而 JAL 语句执行时在第二个周期结束时的上升沿才会将 PC+4 的地址值写回到寄存器堆 RF 中(设计的周期之间是以上升沿作为分隔标志的)，而测试文件也是在上升沿时检测 debug_wb_rf_wdata 和 ref_wb_rf_wdata 是否相同，这样自然会判断为不相同从而测试失败；

2)解决办法：

将寄存器堆 RF 写入的有效沿从上升沿改为下降沿；这样就将 RF 写入的时间放到了一个周期的中间位置，这样在周期结束时 debug_wb_rf_wdata 和 ref_wb_rf_wdata 就会相同，从而使 JAL 语句测试正确；

设计的性能分析

（资源使用情况、主频、功耗数据和自我分析）

在 23MHz 时钟频率的条件下的性能分析：

- **各模块资源使用情况：**

Name	Slice LUTs (63400)	Block RAM Tile (135)	Bonded IOB (285)	BUFGCTRL (32)	PLLE2_ADV (6)	Slice Registers (126800)	F7 Muxes (31700)	Slice (15850)	LUT as Logic (63400)
minisys	1581	29	72	3	1	1271	384	877	1581
u_ifetc32 (ifetc32)	139	14.5	0	0	0		0	72	139
u_PC (PC)	61	0	0	0	0		0	29	61
u_NPC (NPC)	12	0	0	0	0		0	21	12
IM (programrom)	31	14.5	0	0	0		0	15	31
instmem (prgrom)	31	14.5	0	0	0		0	15	31
u_idecode32 (idecode32)	1173	0	0	0	0		384	783	1173
u_registerfile (registerfile)	864	0	0	0	0		384	684	864
u_executs32 (executs32)	9	0	0	0	0		0	45	9
u_ALU (ALU)	9	0	0	0	0		0	34	9
u_dmemory32 (dmemory32)	31	14.5	0	0	0		0	20	31
ram (ram)	31	14.5	0	0	0		0	15	31
U0 (ram_blk_mem_gen_...	31	14.5	0	0	0		0	15	31
u_control32 (control32)	229	0	0	0	0		0	133	229
u_judge (judge)	0	0	0	0	0		0	1	0
u_CU (CU)	229	0	0	0	0		0	133	229
UCLK (cpuclk)	0	0	0	2	1		0	0	0
inst (cpuclk__cpuclk_clk_wiz)	0	0	0	2	1		0	0	0

可见消耗资源数较大的模块是译码模块 idecode32 和控制器模块 control32，其中译码模块又以寄存器堆 registerfile 消耗资源最多；

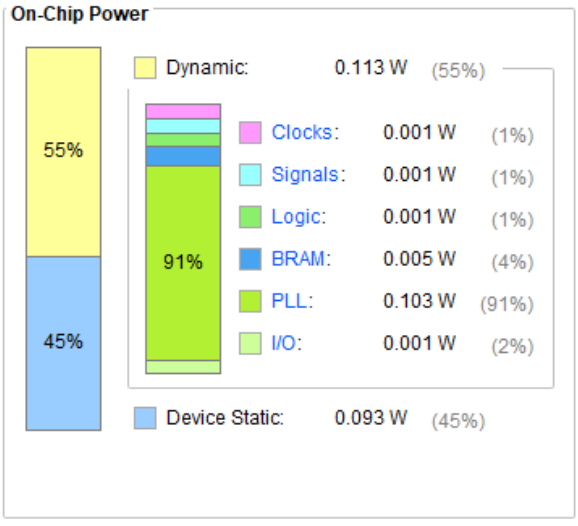
- **功耗数据：**

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.206 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25.6°C**
Thermal Margin:	59.4°C (22.1 W)
Effective ϑJA:	2.7°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity



可见功率消耗较大的部分主要是在 PLL 锁相环中，占了全部 Dynamic 功率消耗的 91%；

- **主频分析：**

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	13.251 ns	Worst Hold Slack (WHS):	0.245 ns	Worst Pulse Width Slack (WPWS):	2.633 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2709	Total Number of Endpoints:	2709	Total Number of Endpoints:	1280

All user specified timing constraints are met.

在 23MHz 的主频下完成 mycpu_tb.v 测试的用时：

```
[ 914799 ns] Test is running, debug_wb_pc = 0x000005ec
[ 915147 ns] Test is running, debug_wb_pc = 0x000005f0
[ 915494 ns] Test is running, debug_wb_pc = 0x000005f4
[ 915581 ns] Test is running, debug_wb_pc = 0x00000100
==================================================

Test end!
----PASS!!!
$finish called at time : 915708296 ps : File "D:/HUZHISHENG/Textbo
run: Time (s): cpu = 00:00:04 ; elapsed = 00:00:12 . Memory (MB):
```

（注意上面的测试用的是老师发出来的最新的“**31 条指令测试包**”中的测试文件，运行时间会比之前的“**单周期测试包**”花的时间更长）

例如：同等频率下运行“**单周期测试包**”里的测试文件花费的时间：

```
[ 744973 ns] Test is running, debug_wb_pc = 0x00000570
[ 745320 ns] Test is running, debug_wb_pc = 0x00000574
[ 745668 ns] Test is running, debug_wb_pc = 0x00000578
[ 746016 ns] Test is running, debug_wb_pc = 0x0000057c
[ 746364 ns] Test is running, debug_wb_pc = 0x00000580
[ 746712 ns] Test is running, debug_wb_pc = 0x00000584
[ 746799 ns] Test is running, debug_wb_pc = 0x00000100
==================================================

Test end!
----PASS!!!
$finish called at time : 746925696 ps : File "D:/HUZHISHENG/Text
```

在 115MHz 时钟频率的条件下的性能分析：

经过不断测试，发现在 PLL 分频调至 115MHz 的情况下 Timing Summary 给出的结果依然为正值，说明本 CPU 设计良好；

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.050 ns	Worst Hold Slack (WHS):	0.157 ns	Worst Pulse Width Slack (WPWS):	2.633 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2709	Total Number of Endpoints:	2709	Total Number of Endpoints:	1280

All user specified timing constraints are met.

```
[ 188642 ns] Test is running, debug_wb_pc = 0x000005f0
[ 188712 ns] Test is running, debug_wb_pc = 0x000005f4
[ 188729 ns] Test is running, debug_wb_pc = 0x00000100
==================================================================

Test end!
----PASS!!!
$finish called at time : 188786547 ps : File "D:/HUZHISHENG/Textbooks/COMP
```

（上面测试依然使用的是“**31 条指令测试包**”中的测试文件）

项目总结

（包括设计的总结和还需改进的内容以及收获）

设计总结：

多周期CPU的设计是我在单周期CPU的基础上经过增加状态机以及按照不同状态不同指令上给出不同控制信号的改变而来，多周期 CPU 设计的难点就是要为每条指令的在不同周期执行的操作进行设计，而要想设计的多周期指令正确完成操作，就要求我们在写代码之前设计好数据通路表和控制信号表，最好把每条指令需要多少周期和各个周期分别完成什么操作都提前记录在一张表上，这样我们在编写实际代码时才会稳步有序地推进设计进展，否则就很可能设计出来的 CPU 功能混乱，不能正确实现功能，而这样也会导致我们后期 debug 和优化工作难以顺利进行；

还需改进的内容：

目前的设计是实现了单周期和多周期的 CPU，不足的地方是目前的 CPU 没有实现异常检测的机制，例如加法指令如果结果溢出了不会有任何特殊处理，而如果实现了溢出检测则还可以为之后的系统中断等功能作铺垫；

如果要想进一步提高指令执行效率则可以实现 CPU 的指令流水架构，通过指令的多级流水，来充分发挥资源的利用率；

收获：

通过本次实验，对 CPU 的架构有了更深的理解，同时也学习了 MIPS 指令集，以及学习运用 MIPS 指令实现走马灯等功能；通过自己设计数据通路表和控制信号表，对 CPU 执行指令的每个步骤都更加清晰，同时也具备了自己设计并实现一个简单的单周期或多周期 CPU 的能力；

拓展实验：汇编器

拓展实验：针对实验一汇编实验用到的 23 条汇编指令设计的汇编器

汇编器功能：

能够将 23 条汇编指令按照 mars4_5 中的处理方式汇编成相应的机器码；

经测试机器码能够正常运行在实验一中老师提供的 logisim 模型上；

注意：

0x3c011001	lui $1,0x00001001	35: lw $s1,data1
0x8c310000	lw $17,0x00000000($1)	

0x3610f000	ori $16,$16,0x0000f000	9: ori $s0,$s0,0xF000
0x8e080c72	lw $8,0x00000c72($16)	14: lw $t0,0xC72($s0)
0x21170000	addi $23,$8,0x00000000	15: addi $s7,$t0,0

在 mars4_5 里面对于 lw 指令有两种处理方式，如果是数据段的数据，lw 就被处理成 lui+lw 语句的格式，如果数据来自 IO 段，则不会添加 lui 指令，本汇编器与之保持一致；

使用本汇编器的特殊规定：

①要求标号 label 不能和汇编指令同行，即标号“xxx:”后面的第一条汇编语句不能和其处在同一行；

②要求代码段前面必须要有“.text”表明代码段的开始；

③要求数据段中的数据必须是 word 类型(占 4 字节，方便计算变量的地址)；

④数据段“.data”需要在“.text”代码段前面；

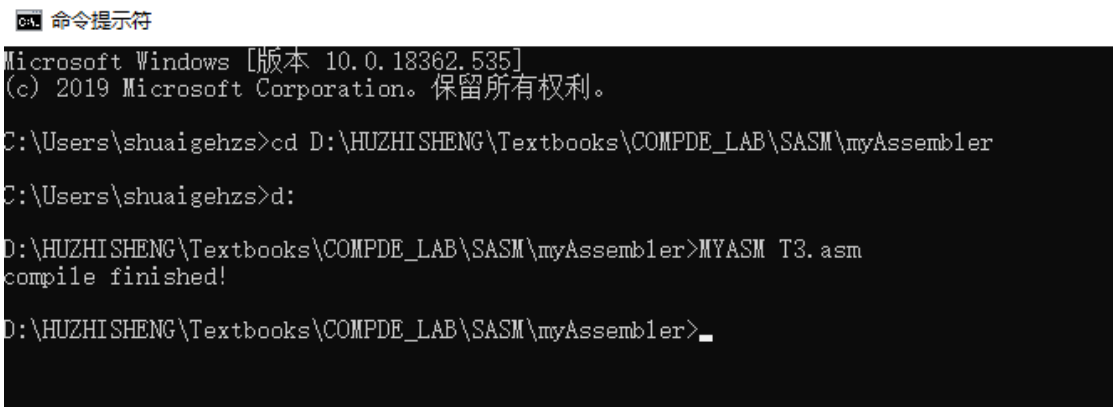
⑤要求 beq, bne, j, jal 等语句中的跳转的地址都必须使用标号；

使用方法：

1)编译 MYASM.cpp 文件；

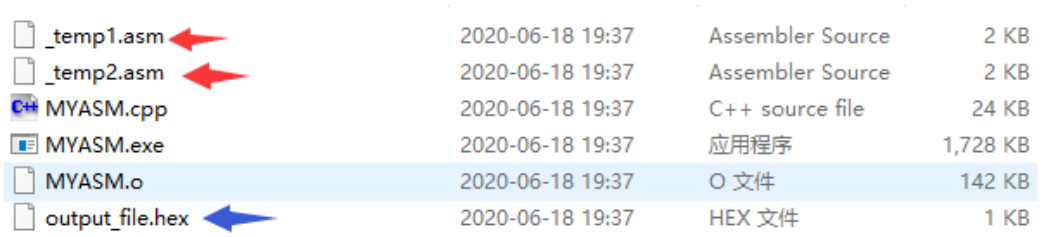
2)将 cmd 当前路径切换到 MYASM.exe 所在位置；

3)在 cmd 中输入“MYASM {你的文件名}”(中间带个空格)，例如下图：



显示 compile finished 则代表汇编完成；

4)生成的文件：



最终生成的 output_file.hex 就是我们想要的 **16 进制**机器码文件；

output_file.hex

```
3c10ffff
3610f000
8e080c72
21170000
00084182
31080003
1100fffb
2108ffff
11000003
2108ffff
11000016
08100032
3c011001
```

最上面的_temp1.asm 和_temp2.asm 两个文件是汇编过程生成的中间文件，可以直接删除掉；

设计思路：

汇编过程扫描整个文件三遍，每一遍扫描都会生成一个新的文件，前两遍生成的是两个中间文件“_temp1.asm”和“_temp2.asm”，第三遍生成的就是目标文件“output_file.hex”；

1)第一遍扫描：

第一遍扫描很简单，就是将原先在代码中的注释去掉：

例如：源文件如下

```
.data
data1:.word 0x00FFF000
data2:.word 0x000FFF00
data3:.word 0xFF000000
.text
initial:
#将IO区域基地址存入s0
lui $s0,0xFFFF
ori $s0,$s0,0xF000
mode_check:
#判断应该进入哪个模式,mode1是指导书中模式1,mode2是指导书中模式2,mode3是指导书中模式3
#在这个设计中,读取[SW23,SW22]的值作为选择,选择01是模式1,10是模式2,11是模式3,如果是00则会熄灭所有LED灯
#如果在执行过程中改变模式选择,则会在当前模式执行完完整的循环后才会跳转到新选择的模式中去
lw $t0,0xC72($s0)
addi $s7,$t0,0          #将输入保存下来,每隔半秒检查是否模式发生了改变,一旦改变则回到mode_check
srl $t0,$t0,6
andi $t0,$t0,0x0003
beq $t0,$zero,mode_check     #选择是00,则循环mode_check,直到选择其它模式为止
addi $t0,$t0,-1
beq $t0,$zero,mode1      #选择是01,进入模式1
addi $t0,$t0,-1
beq $t0,$zero,mode2      #选择是10,进入模式2
j mode3              #选择是11,进入模式3
```

则生成的临时文件如下

```
.data
data1:.word 0x00FFF000
data2:.word 0x000FFF00
data3:.word 0xFF000000
.text
initial:

lui $s0,0xFFFF
ori $s0,$s0,0xF000
mode_check:



lw $t0,0xC72($s0)
addi $s7,$t0,0
srl $t0,$t0,6
andi $t0,$t0,0x0003
beq $t0,$zero,mode_check
addi $t0,$t0,-1
beq $t0,$zero,mode1
addi $t0,$t0,-1
beq $t0,$zero,mode2
j mode3

```

2)第二遍扫描：

第二遍扫描就是**负责记下各种标号的地址**和**去掉多余的空行和每条语句前的缩进。**标号主要是两种：①数据段中的数据；②代码段的标号；对于这两种标号都存储在同一个词典 MAP 中，这个 MAP 是<string,int>型，对于已经去掉注释的临时文件 temp1，我们创建该文件的文件流，每次从文件流里面读出一行，如果改行是空行则直接跳过，否则再处理该行的字符串。因为源代码最开始是数据段“.data”因此此时认为每一行(除了“.data”)都是一个数据的标号,此时只会用到 cur_datap；

当扫描到“.text”则代表开始处理指令段，对于每一行的字符串，在其中寻找“:”，如果有冒号则表明该行是一个标号，则在字典中寻找该标号是否已经存在，已经存在则报错，否则将这个标号和其地址存储字典中。对于每个标号如何知道其代表的地址，我采用 cur_textp 和 cur_datap 两个指针来代表当前地址，cur_textp 是代码的地址，代码地址是从 0x00400000 开始的，因此 curtextp 的初值为 0x00400000；cur_datap 则是数据在数据段的偏移地址，初值为 0；对于每个扫描的不存在“：”的一行，则认为是一个占有 4 字节的语句，因此把对应指针加 4 即可，注意：如果是 lw 语句，还要看其加载的数据是否是数据段的，如果是，因为在其前面会新添加一个 lui 语句，因此相应的地址指针要加 8。

这样在往字典中插入<label,addr>标号以及其地址组成的键值对时，相应的 cur_datap 或 cur_textp 就是我们想要的标号的地址；

经过第二遍扫描就可以得到一个**没有标号行，没有空行，没有缩进，以及所有需要在 lw 语句前添加的 lui 语句也都被添加上**的中间代码文件，例如下图：

temp2.asm

```
lui $s0,0xFFFF
ori $s0,$s0,0xF000
lw $t0,0xC72($s0)
addi $s7,$t0,0
srl $t0,$t0,6
andi $t0,$t0,0x0003
beq $t0,$zero,mode_check
addi $t0,$t0,-1
beq $t0,$zero,model
addi $t0,$t0,-1
beq $t0,$zero,mode2
j mode3
lui $at,0x1001
lw $s1,0x0($at)
lui $at,0x1001
lw $s2,0x4($at)
addi $s3,$zero,24
andi $t1,$s1,0x0FFF
add $t2,$s2,$zero
srl $t2,$t2,20
sll $t2,$t2,12
addi $t3,$zero,0
or $t3,$t1,$zero
or $t3,$t3,$t2
sw $t3,0xC60($s0)
srl $t3,$t3,16
sw $t3,0xC62($s0)
jal wait
srl $s1,$s1,1
sll $s2,$s2,1
addi $s3,$s3,-1
bne $s3,$zero,loop1
```

3)第三遍扫描：

第三遍扫描就是生成机器码的过程，这遍扫描很简单，但是工作很繁琐，具体说来就是根据 temp2 文件中的每行，取出它的操作符后，就根据不同的操作符进行不同的操作。

例如下图代码：

```
}
else if(op_str=="j" || op_str=="jal"){
    int op_num=getOp(op_str,op2num);
    if (op_num == -1){
        cout << "Syntax error at Line " << line_no << "." << endl;
        exit(-1);
    }
    char imm_cstr[30];
    s_stream.getline(imm_cstr,30);
    string imm_str=imm_cstr;
    int p=0;
    while(isspace(imm_str[p]))p++;
    imm_str=imm_str.substr(p,imm_str.size()-p);
    int imm_num=getImm(imm_str,mymap);
    imm_num=imm_num/4;

    string bin_code="";
    bin_code+=getBinStr(op_num,6);
    bin_code+=getBinStr(imm_num,26);
    out_f<<binToHex(bin_code)<<endl;

}
```

上面代码就是如果扫描到的语句是 J 语句或者 JAL 语句所作的操作：先调用 getOp 函数将语句操作符“J”或“JAL”对应的 op 字段数 op_num 取出来，再把改行剩余字符串去掉空格后调用 getImm 函数得到指令的要跳转到的地址 imm_num，最后把 op_num 和 imm_num 转为十六进制字符串再输出到 output_file.hex 中即可。

经过上述三遍扫描后得到的 output_file.hex 即是汇编出来的十六进制机器码文件。

CPP 源代码：

github 地址：

https://github.com/huzhisheng/MyRepublic/blob/master/MYASM.cpp

附录 A.数据通路表

	所属单元	取指单元					
	部件	PC	NPC			IM（指令存储器）	IR
	输入信号	DI	PC	Imm	RA（寄存器）	A	
R型指令	add	NPC.NPC	PC.DO			PC.DO	IM.D
	addu	NPC.NPC	PC.DO			PC.DO	IM.D
	sub	NPC.NPC	PC.DO			PC.DO	IM.D
	subu	NPC.NPC	PC.DO			PC.DO	IM.D
	and	NPC.NPC	PC.DO			PC.DO	IM.D
	or	NPC.NPC	PC.DO			PC.DO	IM.D
	xor	NPC.NPC	PC.DO			PC.DO	IM.D
	nor	NPC.NPC	PC.DO			PC.DO	IM.D
	slt	NPC.NPC	PC.DO			PC.DO	IM.D
	sltu	NPC.NPC	PC.DO			PC.DO	IM.D
	sll	NPC.NPC	PC.DO			PC.DO	IM.D
	srl	NPC.NPC	PC.DO			PC.DO	IM.D
	sra	NPC.NPC	PC.DO			PC.DO	IM.D
	sllv	NPC.NPC	PC.DO			PC.DO	IM.D
	srlv	NPC.NPC	PC.DO			PC.DO	IM.D
	srav	NPC.NPC	PC.DO			PC.DO	IM.D
	jr	NPC.NPC	PC.DO		A	PC.DO	IM.D
I型指令	addi	NPC.NPC	PC.DO			PC.DO	IM.D
	addiu	NPC.NPC	PC.DO			PC.DO	IM.D
	andi	NPC.NPC	PC.DO			PC.DO	IM.D
	ori	NPC.NPC	PC.DO			PC.DO	IM.D
	xori	NPC.NPC	PC.DO			PC.DO	IM.D
	sltiu	NPC.NPC	PC.DO			PC.DO	IM.D
	lui	NPC.NPC	PC.DO			PC.DO	IM.D
	lw	NPC.NPC	PC.DO			PC.DO	IM.D
	sw	NPC.NPC	PC.DO			PC.DO	IM.D
	beq	NPC.NPC	PC.DO	IR[15:0]		PC.DO	IM.D
	bne	NPC.NPC	PC.DO	IR[15:0]		PC.DO	IM.D
	bgtz	NPC.NPC	PC.DO	IR[15:0]		PC.DO	IM.D
J型指令	j	NPC.NPC	PC.DO	IR[25:0]		PC.DO	IM.D
	jal	NPC.NPC	PC.DO	IR[25:0]		PC.DO	IM.D
综合		NPC.NPC	PC.DO	IR[25:0]	A	PC.DO	IM.D
控制单元	功能选择信号	PCWr	NPCOp			无选择	IRWr
	多路选择器信号	无选择					无选择
	返回信号						

	所属单元	译码单元							
	部件	RF（寄存器堆）				A	B	S_EXT	E
	输入信号	A1（读出1）	A2（读出2）	A3（写入）	WD（写回值）				
R型指令	add	IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF.RD1	RF.RD2		EXT.Ext
	addu	IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF.RD1	RF.RD2		EXT.Ext
	sub	IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF.RD1	RF.RD2		EXT.Ext
	subu	IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF.RD1	RF.RD2		EXT.Ext
	and	IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF.RD1	RF.RD2		EXT.Ext
	or	IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF.RD1	RF.RD2		EXT.Ext
	xor	IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF.RD1	RF.RD2		EXT.Ext
	nor	IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF.RD1	RF.RD2		EXT.Ext
	slt	IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF.RD1	RF.RD2		EXT.Ext
	sltu	IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF.RD1	RF.RD2		EXT.Ext
	sll		IR[20:16]	IR[15:11]	ALUOut	RF.RD1	RF.RD2		EXT.Ext
	srl		IR[20:16]	IR[15:11]	ALUOut	RF.RD1	RF.RD2		EXT.Ext
	sra		IR[20:16]	IR[15:11]	ALUOut	RF.RD1	RF.RD2		EXT.Ext
	sllv	IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF.RD1	RF.RD2		EXT.Ext
	srlv	IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF.RD1	RF.RD2		EXT.Ext
	srav	IR[25:21]	IR[20:16]	IR[15:11]	ALUOut	RF.RD1	RF.RD2		EXT.Ext
	jr	IR[25:21]				RF.RD1	RF.RD2		EXT.Ext
I型指令	addi	IR[25:21]		IR[20:16]	ALUOut	RF.RD1	RF.RD2	IR[15:0]	EXT.Ext
	addiu	IR[25:21]		IR[20:16]	ALUOut	RF.RD1	RF.RD2	IR[15:0]	EXT.Ext
	andi	IR[25:21]		IR[20:16]	ALUOut	RF.RD1	RF.RD2	IR[15:0]	EXT.Ext
	ori	IR[25:21]		IR[20:16]	ALUOut	RF.RD1	RF.RD2	IR[15:0]	EXT.Ext
	xori	IR[25:21]		IR[20:16]	ALUOut	RF.RD1	RF.RD2	IR[15:0]	EXT.Ext
	sltiu	IR[25:21]		IR[20:16]	ALUOut	RF.RD1	RF.RD2	IR[15:0]	EXT.Ext
	lui			IR[20:16]	ALUOut	RF.RD1	RF.RD2	IR[15:0]	EXT.Ext
	lw	IR[25:21]		IR[20:16]	DR	RF.RD1	RF.RD2	IR[15:0]	EXT.Ext
	sw	IR[25:21]	IR[20:16]			RF.RD1	RF.RD2	IR[15:0]	EXT.Ext
	beq	IR[25:21]	IR[20:16]			RF.RD1	RF.RD2		EXT.Ext
	bne	IR[25:21]	IR[20:16]			RF.RD1	RF.RD2		EXT.Ext
	bgtz	IR[25:21]				RF.RD1	RF.RD2		EXT.Ext
J型指令	j					RF.RD1	RF.RD2		EXT.Ext
	jal			0x1F	PC.DO	RF.RD1	RF.RD2		EXT.Ext
综合		IR[25:21]	IR[20:16]	IR[15:11] IR[20:16] 0x1F	ALUOut DR PC.DO	RF.RD1	RF.RD2	IR[15:0]	EXT.Ext
控制单元	功能选择信号 多路选择器信号	无选择	无选择	 WRSel	RFWr WDSel	无选择	无选择	EXTOp 无选择	无选择
	返回信号								

	所属单元	执行单元				数据存储器		
	部件	ALU			ALUOut	DM（数据存储器）	DR	WD
	输入信号	A	B	C		A		
R型指令	add	A	B		ALU.DO		DM.RD	
	addu	A	B		ALU.DO		DM.RD	
	sub	A	B		ALU.DO		DM.RD	
	subu	A	B		ALU.DO		DM.RD	
	and	A	B		ALU.DO		DM.RD	
	or	A	B		ALU.DO		DM.RD	
	xor	A	B		ALU.DO		DM.RD	
	nor	A	B		ALU.DO		DM.RD	
	slt	A	B		ALU.DO		DM.RD	
	sltu	A	B		ALU.DO		DM.RD	
	sll		B	IR[10:6]	ALU.DO		DM.RD	
	srl		B	IR[10:6]	ALU.DO		DM.RD	
	sra		B	IR[10:6]	ALU.DO		DM.RD	
	sllv		B	A[4:0]	ALU.DO		DM.RD	
	srlv		B	A[4:0]	ALU.DO		DM.RD	
	srav		B	A[4:0]	ALU.DO		DM.RD	
	jr				ALU.DO		DM.RD	
I型指令	addi	A	E		ALU.DO		DM.RD	
	addiu	A	E		ALU.DO		DM.RD	
	andi	A	E		ALU.DO		DM.RD	
	ori	A	E		ALU.DO		DM.RD	
	xori	A	E		ALU.DO		DM.RD	
	sltiu	A	E		ALU.DO		DM.RD	
	lui		E	0x10	ALU.DO		DM.RD	
	lw	A	E		ALU.DO	ALUOut	DM.RD	
	sw	A	E		ALU.DO	ALUOut	DM.RD	B
	beq	A	B		ALU.DO		DM.RD	
	bne	A	B		ALU.DO		DM.RD	
	bgtz	A	0x0		ALU.DO		DM.RD	
J型指令	j				ALU.DO		DM.RD	
	jal				ALU.DO		DM.RD	
综合		A	B	IR[10:6]	ALU.DO	ALUOut	DM.RD	B
			E	A[4:0]				
			0x0	0x10				
控制单元	功能选择信号	ALUOp				DMWr		
	多路选择器信号		Bsel	Csel	无选择			
	返回信号	Less,Equal,More						

附录 B. 控制信号表

控制信号取值矩阵（填写）											
指令	NPCOp	PCWr	IRWr	RFWr	EXTOp	ALUOp	DMWr	WRSel	WDSel	Bsel	Csel
add	T1:PC4	T1:1	T1:1	T4:1		T3:ADD		T4:RD	T4:ALUDO	T3:RD2	
addu	T1:PC4	T1:1	T1:1	T4:1		T3:ADD		T4:RD	T4:ALUDO	T3:RD2	
sub	T1:PC4	T1:1	T1:1	T4:1		T3:SUB		T4:RD	T4:ALUDO	T3:RD2	
subu	T1:PC4	T1:1	T1:1	T4:1		T3:SUB		T4:RD	T4:ALUDO	T3:RD2	
and	T1:PC4	T1:1	T1:1	T4:1		T3:AND		T4:RD	T4:ALUDO	T3:RD2	
or	T1:PC4	T1:1	T1:1	T4:1		T3:OR		T4:RD	T4:ALUDO	T3:RD2	
xor	T1:PC4	T1:1	T1:1	T4:1		T3:XOR		T4:RD	T4:ALUDO	T3:RD2	
nor	T1:PC4	T1:1	T1:1	T4:1		T3:NOR		T4:RD	T4:ALUDO	T3:RD2	
slt	T1:PC4	T1:1	T1:1	T4:1		T3:SLT		T4:RD	T4:ALUDO	T3:RD2	
sltu	T1:PC4	T1:1	T1:1	T4:1		T3:SLTU		T4:RD	T4:ALUDO	T3:RD2	
sll	T1:PC4	T1:1	T1:1	T4:1		T3:SLL		T4:RD	T4:ALUDO	T3:RD2	T3:SHAMT
srl	T1:PC4	T1:1	T1:1	T4:1		T3:SRL		T4:RD	T4:ALUDO	T3:RD2	T3:SHAMT
sra	T1:PC4	T1:1	T1:1	T4:1		T3:SRA		T4:RD	T4:ALUDO	T3:RD2	T3:SHAMT
sllv	T1:PC4	T1:1	T1:1	T4:1		T3:SLL		T4:RD	T4:ALUDO	T3:RD2	T3:RD1
srlv	T1:PC4	T1:1	T1:1	T4:1		T3:SRL		T4:RD	T4:ALUDO	T3:RD2	T3:RD1
srav	T1:PC4	T1:1	T1:1	T4:1		T3:SRA		T4:RD	T4:ALUDO	T3:RD2	T3:RD1
jr 2读出到A	T1:PC4 T3:JR	T1:1 T3:1	T1:1								
addi	T1:PC4	T1:1	T1:1	T4:1	T2:SIGN	T3:ADD		T4:RT	T4:ALUDO	T3:EXT	
addiu	T1:PC4	T1:1	T1:1	T4:1	T2:SIGN	T3:ADD		T4:RT	T4:ALUDO	T3:EXT	
andi	T1:PC4	T1:1	T1:1	T4:1	T2:ZERO	T3:AND		T4:RT	T4:ALUDO	T3:EXT	
ori	T1:PC4	T1:1	T1:1	T4:1	T2:ZERO	T3:OR		T4:RT	T4:ALUDO	T3:EXT	
xori	T1:PC4	T1:1	T1:1	T4:1	T2:ZERO	T3:XOR		T4:RT	T4:ALUDO	T3:EXT	
sltiu	T1:PC4	T1:1	T1:1	T4:1	T2:SIGN	T3:SLTU		T4:RT	T4:ALUDO	T3:EXT	
lui	T1:PC4	T1:1	T1:1	T4:1	T2:ZERO	T3:SLL		T4:RT	T4:ALUDO	T3:EXT	T3:SIXTEEN
lw	T1:PC4	T1:1	T1:1	T5:1	T2:SIGN	T3:ADD		T5:RT	T5:DM	T3:EXT	
sw	T1:PC4	T1:1	T1:1		T2:SIGN	T3:ADD	T4:1			T3:EXT	
beq	T1:PC4 T3:BT	T1:1 T3:BR_TRUE	T1:1			T3:SUB				T3:RD2	
bne	T1:PC4 T3:BT	T1:1 T3:BR_TRUE	T1:1			T3:SUB				T3:RD2	
bgtz	T1:PC4 T3:BT	T1:1 T3:BR_TRUE	T1:1			T3:SUB				T3:B0	
j	T1:PC4 T2:J	T1:1 T2:1	T1:1								
jal	T1:PC4 T2:J	T1:1 T2:1	T1:1	T2:1				T2:RA	T2:PCDO		