CPU 实验报告

特点：

1. 哈佛存储器结构，大端格式；

2. 类 MIPS 精简指令集，支持子程序调用和软中断；

3. 实现了乘除法；

4. 五级流水线，工作频率可达 80MHz（每个时钟周期一条指令，不计流水线冲突）。

一、小组成员

于晃 无 64 负责部分数据通路以及调试

李龙毅 无 64 负责 ALU

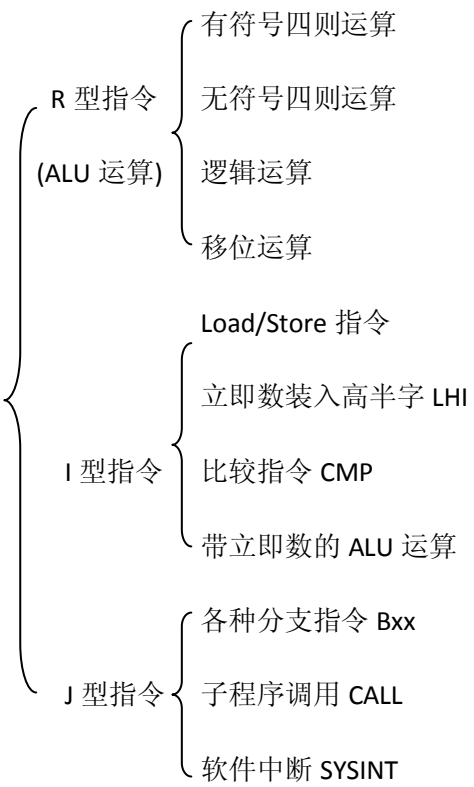
闫轲 无 610 负责控制器、部分数据通路、汇编器以及调试

二、模块原理介绍

指令集介绍

这个指令集是由于晃和闫轲共同协定的，其中于晃提出了初始版本，闫轲根据实际情况进行了改进。

指令集共包括 50 条指令，具体分配如下：



详细的表格控制码、控制信号分配可见压缩包中的 NewInstructions.xls。下面对各指令的设计初衷进行介绍。

- R 型指令
  - ✓ 与老师给的 Z0 指令集不同，这个指令集将加减乘除四则运算都包括进去了，会比只有加减的指令集更易用一些，算是部分实现了 FPU 的指令吧。不过缺点是乘除法的延时会比加减法大一些。
  - ✓ 无符号加减其实与有符号加减并无本质不同，X86 指令集没有无符号加减，MIPS 指令集无符号加减与有符号的唯一区别是无符号加减不置 OV 标志寄存器。这里参考 MIPS 指令集的做法，同时也照顾了指令集的“整齐性”。
  - ✓ 逻辑运算中 NOT 指令其实可以用 XOR 指令来实现，这里单独用了一条指令也是为了易用性和“整齐性”，但 NOT 指令只有两个操作数。
  - ✓ R 型指令的 Op 字段都为 0，具体功能由 Funct 字段指定。实现的功能为将 Rx 中内容和 Ry 中内容进行操作后存入 Rz。

Op	Rx	Ry	Rz	(useless)	Funct
6 位	5 位	5 位	5 位	5 位	6 位

- I 型指令
  - ✓ 根据老师要求，Load/Store 类指令按长度分为字节、（低）半字和字三种，Load 类指令(内存→寄存器)还分为有符号和无符号两种(Store 类指令不需区分)，因此共有 6 个 Load 指令，3 个 Store 指令。其格式为 L/S Rx, Offset, Ry，load 类指令是将 Rx 中地址偏移 Offset 后指向的内存存入 Ry；store 类指令是将 Ry 存入 Rx 中地址偏移 Offset 后指向的内存。
  - ✓ LHI Ry, Imm16 指令实现了立即数装入高半字功能，与 ORI $0, Imm16,Ry 指令接合即可实现装入 32 位立即数的功能。
  - ✓ CMP Rx, Ry 实现了比较两个寄存器值（本质为有符号减）并置标志寄存器的功能，为下面的条件分支指令做准备。
  - ✓ 带立即数的 ALU 运算与寄存器 ALU 运算的唯一区别是：前者不存在 NOT 指令。

Op	Rx/0	Ry	Imm16/Offset/0
6 位	5 位	5 位	16 位

- J 型指令
  - ✓ 条件分支指令（包括相等跳转 BO，不等跳转 BNO，有符号大于跳转 BGT，有符号小于等于跳转 BLET，BGTU，BLETU）本来计划的是设为 I 型指令，即对 Rx 和 Ry 进行比较后，根据标志寄存器 flag 条件跳转到 Addr，但这样一来一条指令要用两次 ALU，一次用来比较，一次用来计算跳转地址，造成了延时；另外很多情况下前面的 ALU 指令已经更改了 flag，不需要再进行比较了。因此将条件分支指令设为 J 型指令，这样一来也扩大了跳转范围。老师给的提示中有符号跳转分为大于跳转和小于跳转，细思之觉此分法不妥，因为这样大于等于跳转就无法表示了。
  - ✓ 子程序调用指令 CALL 与跳转指令很相似，都是跳转，但 CALL 需要保存当前 PC 值以便返回。这里参考 MIPS，在跳转的同时将 PC 存入寄存器$ra。为此 CPU 中设置了专用通路。
  - ✓ 与 CALL 类似，系统中断指令 SYSINT 也需要存 PC 并跳转，不同的是这里假设在程序内存中有一个从 0 号内存开始的、每个表项为 4 个字节的中断向量表，因此 SYSINT 的参数并不是地址偏移量，而是编号。实际上跳转到参数左移 2 位(乘 4)的绝对地址。

✓ 在汇编的过程中，除了 SYSINT 指令外，汇编器都是用当前指令的编号（从 0 开始）减去要跳转到的指令编号，求补码后存入 Addr 字段。CPU 执行的过程中用当前的 PC 值加上 Addr 后即得跳转地址。

Op	Addr
6 位	26 位

注：其实这个指令集还相当不完善，比如，没有设计子程序返回指令 RET，定义数据的伪指令等，但基本的运算、分支和存储都已经实现了。

汇编器介绍（闫轲）

为了测试方便，用 c++编写了一个简易的汇编器，用命令行方式操作：

```
D:\Yan Ke\_Courses\Computer Principles\DataCycle>ns.exe rule.txt in.txt out.txt
done!
```

程序读入一个记录各个指令对应二进制代码的文件 rule.txt，和用汇编语言编写的 in.txt，输出一个二进制代码文件供人工或 verilog 读。输入文件例：

```
        add $1,$2,$3
X1:     sub $3,4,$3
        cmp $3,$1
        blet X1
        sysint 1
```

- 寄存器格式为：$x
- 立即数支持十进制
- 支持行标号
- 可以检测错误的指令字和行标号

控制器介绍（闫轲）

根据输入的 6 位指令字 op 和 funct 字段，输出以下控制信号：

MP1：选择寄存器堆的输入寄存器序号是指令中的 Rz 字段（当 R 型指令时）或 Ry 字段（当 I 型指令时）。需要流水到回写阶段。

MP2：选择 ALU 的 X 输入是 pc（如跳转指令）或 Dx。

MP3：选择 ALU 的 Y 输入是符号扩展输出（如带立即数的指令）或 Dy。

MP4：选择 Dz 以及 MUX0 的一个输入是符号扩展输出、ALU 输出或数据内存输出。需要流水到访存阶段。

Aluop：ALU 操作码，R 型指令为 funct 字段的后 4 位，带立即数的 ALU 指令为 op 字段的后 4 位，剩下的为加或减。

Memwrite：写内存控制信号，同时控制写与不写，写字节、半字或字。

Memread：读内存控制信号，同时控制读与不读，读字节、半字或字。

Rin：写寄存器控制信号，需要流水到回写阶段。

Fin：置标志寄存器控制信号，当计算地址时为无效电平。

Special_En：当计算乘除法时，结果有 64 位，则置此信号，将高 32 为写入专用寄存器。

Seop：符号扩展控制信号。

jal_ctrl：当执行 CALL 或 SYSINT 指令时，需要将当前 pc 存入寄存器$ra，则置此信号。

注：更详细的控制信号表可见 NewInstructions.xls。

分支控制逻辑单元介绍（闫轲）

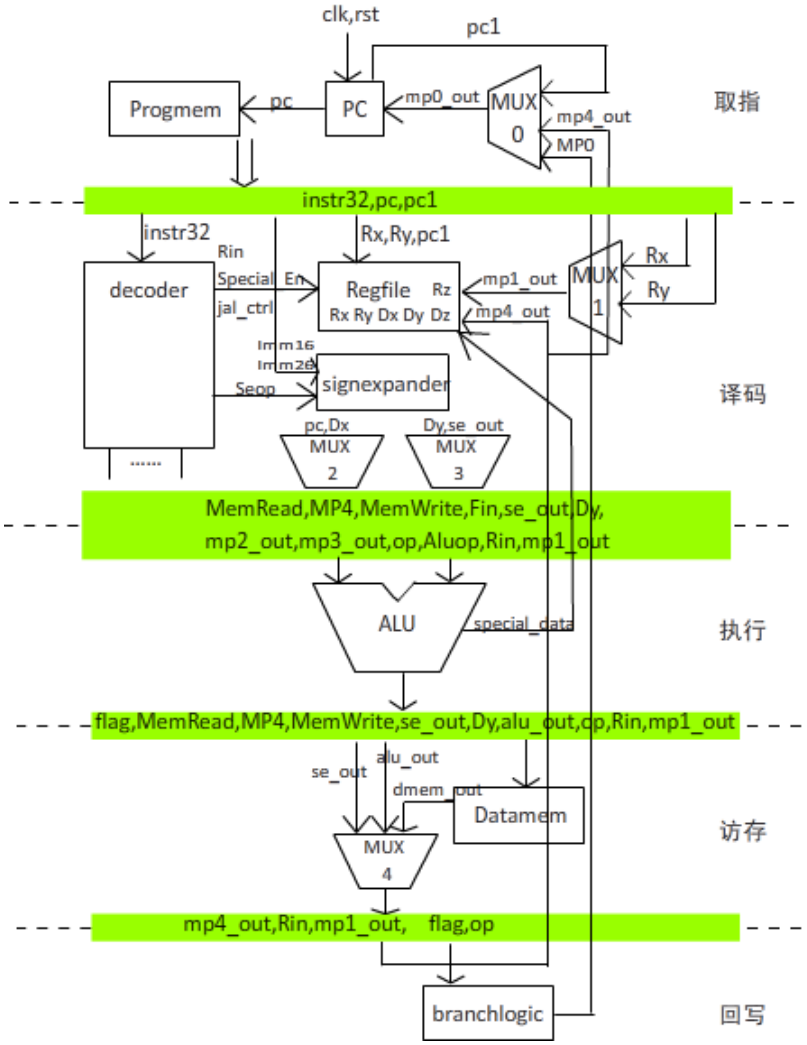
根据 op 字段和 flag 寄存器判断是否跳转。举例：blet 指令（有符号小于等于跳转）出现时，如果 ZR=1，或 NG 和 OV 同时为 0，或 NG 和 OV 同时为 1，则跳转。

仿真内存单元介绍（闫轲）

采用哈佛结构。程序内存单元 Progmem 以字为单位，只读，在编译时用$readmemb 函数读入机器码文件。数据内存单元 Datamem 以字节为单位，采用大端格式，每次时钟下降沿时（不采用上升沿是为了等到输入数据稳定），检测读、写控制信号，若读内存信号有效则将相应内存符号扩展至 32 位后输出；若写内存信号有效则将 32 位输入的相应字节符号扩展至 32 位后存入。

流水线介绍

与老师课件上的实现方法大体相同，每个时钟周期 pc 改变一次（增一或跳转），流水线每级间的寄存器所存储的信号或数据更新一次。具体结构如下图：



三、整体测试

测试程序：

```
ori $0,1,$1

ori $0,6,$3

x0: add $1,$2,$2

addi $1,1,$1

cmp $3,$2

bgt x0

sw $0,0,$1

x1: b x1
```

实现功能：

计算 1+2+3+…，直到结果不小于 5，将和写入 0 号内存（即输出波形中的 DataShow）。最后一句为空循环。

插入空指令、以防冲突后的代码（写与读之间至少 3 条指令，分支指令之后至少 4 条指令）：

```
ori $0,1,$1
ori $0,5,$3
sl $0,$0,$0   ;空指令
sl $0,$0,$0
x0: add $1,$2,$2
addi $1,1,$1
sl $0,$0,$0
sl $0,$0,$0
cmp $3,$2
bgt x0
```

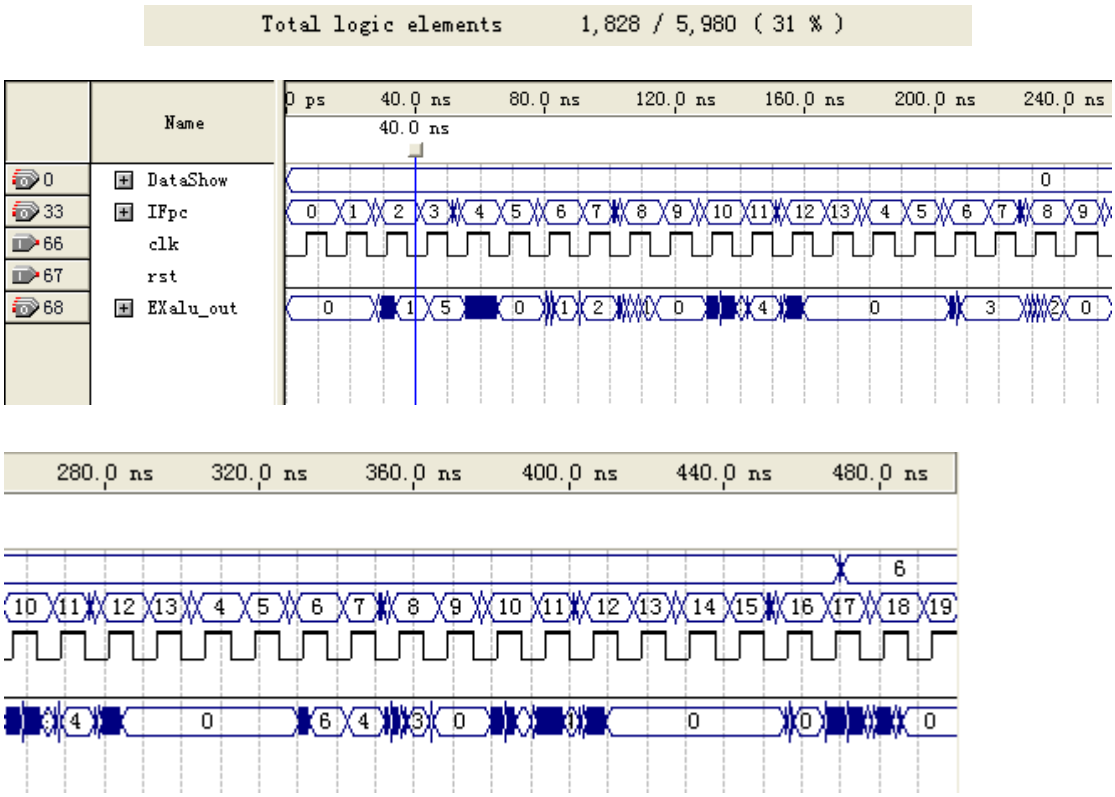
```
sl $0,$0,$0
sl $0,$0,$0
sl $0,$0,$0
sl $0,$0,$0
sw $0,0,$2
x1: b x1
sl $0,$0,$0
sl $0,$0,$0
sl $0,$0,$0
sl $0,$0,$0
```

汇编后的二进制代码：

```
10010100000000010000000000000001
10010100000000110000000000000110
00000000000000000000000000001100
00000000000000000000000000001100
00000000010001000010000000000000
10000000010000100000000000000001
00000000000000000000000000001100
00000000000000000000000000001100
01000000011000100000000000000000
01110100000000000000000000000101
```

```
00000000000000000000000000001100
00000000000000000000000000001100
00000000000000000000000000001100
00000000000000000000000000001100
00100100000000100000000000000000
00000100000000000000000000000000
00000000000000000000000000001100
00000000000000000000000000001100
00000000000000000000000000001100
00000000000000000000000000001100
```

仿真结果：



可以看出，程序最后得出了正确结果：6。

由指令序号 IFpc 看出，程序由 0 号指令运行到 13 号指令（跳转指令后的第 4 条）后发生循环，回到第 4 条指令 add。第三次运行到 13 号指令时不再跳转，之后写入内存。

时钟周期为 12.5ns，若将空指令也算作一条指令，则工作频率 80MHz。不过若将乘除法引入，则应适当降低时钟频率。

四、总结

这次大作业收获甚多，可以说是大学各次大作业中收获最多的一次，进一步学习了 verilog 编程，锻炼了团队合作能力，当然最重要的是完整的学习了一遍 CPU 的构成，CPU、计算机原理在我心中不再神秘。这次实验还锻炼了我的研究能力，看网上介绍的最新 CPU 技术，如超流水线，也能看出些门道了。刚才我在公共汽车上还在琢磨，还有没有什么进一步提高速度的办法呢？暂时想到的办法有当检测到乘除法操作时，可以设置一个暂停信号，从而不让乘除法的长延时影响到整体的时钟频率；前传通路、分支预测都是老师课件上讲过的技术，这次也没有时间实现了，其实我还是挺想挑战一下的。感谢马老师一学期以来富有成效性的教导！