



《计算机组成原理实验》 实验报告

(实验三)

学 院 名 称 : 数据科学与计算机学院

学 生 姓 名 : 何子彬

学 号 : 15352114

专业（班级） : 15 移动一班

时 间 : 2016 年 5 月 23 日

成绩：

实验二：多周期 CPU 设计

一. 实验目的

- 1、认识和掌握多周期数据通路原理及其设计方法。
- 2、掌握多周期 CPU 的实现方法，代码实现方法。
- 3、编写一个编译器，将 MIPS 汇编程序编译为二进制机器码。
- 4、掌握多周期 CPU 的测试方法。

二. 实验内容

设计一个多周期 CPU，该 CPU 至少能实现算术逻辑运算、存储读取、分支指令和跳转功能操作。
需设计的指令格式包括 R 型指令、I 型指令以及 J 型指令。

三. 实验器材

电脑一台、Xilinx ISE 软件一套

四. 实验原理

1、指令格式分析

a) R 型指令

<i>op</i> (6 位)	<i>rs</i> (5 位)	<i>rt</i> (5 位)	<i>rd</i> (5 位)	<i>reserved</i> (11 位)
-----------------	-----------------	-----------------	-----------------	------------------------

R 型指令各字段名称及含义如下：

- ***op***: 指令的基本操作，通常称为操作码 (*opcode*)。
- ***rs***: 第一个源操作数寄存器的地址。
- ***rt***: 第二个源操作数寄存器的地址。
- ***rd***: 用于存放操作结果的目的寄存器的机制。
- ***reserved***: 预留部分，一般为 0，也可用于指令 *op* 字段中操作的特定变式，一般分为移位 *shift* 和 *func* 功能码

对于 *R* 型指令，一般操作都是算术逻辑操作，不涉及立即数，下面我们逐一分析一下对于每一种功能的指令，在 *R* 型指令内所代表的含义：

1. **一般算术逻辑运算**，两个源操作数都来源于寄存器，运算完毕后，写回到 *rd* 寄存器，步骤如下：
 - 将 *rs* 与 *rt* 寄存器内容读出。
 - 将读出内容放进 *ALU* 执行运算。
 - 将运算结果存到 *rd* 寄存器。
2. **比较大小操作**，比较的两个数都是来源于寄存器，而比较的具体操作就是做差，若操作数 1 小于操作数 2，则将结果 1 写回到 *rd* 寄存器，反之写回结果 0，步骤如下：
 - 将 *rs* 与 *rt* 寄存器内容读出。
 - 将读出的内容放进 *ALU* 执行减法运算。
 - 若 *ALU* 结果小于 0，则将 1 写回到 *rd* 寄存器，反之，将 0 写回。
3. **移位操作**，移位指令 *sll* 中，*rs* 寄存器是暂时未用到的，指令将 *rt* 寄存器的内容左移 *sa* 位后写回到 *rd* 寄存器，所以此时 *ALU* 的两个输入就为 *sa* 和 *rt* 寄存器内容，步骤如下：
 - 将 *rt* 寄存器和 *sa* 分量读出，分别作为 *ALU* 的两个输入端。
 - 将 *rt* 寄存器内容左移 *sa* 位得到结果。
 - 将结果存到 *rd* 寄存器。

b) *I* 型指令

<i>op</i> (6 位)	<i>rs</i> (5 位)	<i>rt</i> (5 位)	<i>constant or address</i> (16 位)
-----------------	-----------------	-----------------	-----------------------------------

I 型指令各字段名称以及含义如下：

- ***op***：指令的基本操作，通常称为操作码 (*opcode*)。
- ***rs***：第一个源操作数寄存器的地址。
- ***rt***：用于存放数据（存放于寄存器或数据存储器）的目的寄存器地址
- ***constant or address***：16 位的立即数或者地址偏移量

对于 *I* 型指令，其可执行的功能很多，包括算术逻辑操作、存储器读写操作、分支操作，下面我们逐一分析一下对于每一种功能的指令，在 *I* 型指令内所代表的含义。

1. **算术逻辑操作**，这种算术逻辑操作相对于 *R* 型指令两个操作数都来源于寄存器，*I* 型指令的两个操作数一个来源于 *rs* 寄存器，另一个来源于立即数，两个操作数运算完后，结果会存放于 *rt* 寄存器，步骤如下：
 - 将 *rs* 寄存器数据读出，将 16 位立即数扩展为 32 位。
 - 两个操作数放进 *ALU* 进行运算。
 - 将运算结果存于 *rt* 寄存器。
2. **存储器读操作**，这时 *rs* 寄存器中的数据表示的是数据存储器中的地址，而立即数表示的是相对于地址的偏移量，所以真正的地址是两者之和，然后将数据存储器中指定地址的单元数据读出后存于 *rt* 寄存器，步骤如下：
 - 将 *rs* 寄存器数据读出，将 16 位立即数扩展为 32 位。
 - 两个操作数放进 *ALU* 进行加法运算得真实地址。
 - 访问数据存储器指定位置，并将数据读出。
 - 将读出的数据存到 *rt* 寄存器。
3. **存储器写操作**，这时 *rs* 寄存器依然表示的是数据存储器的地址，立即数同样表示相对于该地址的偏移量，但是对于写操作而言，*rt* 寄存器不再是目的寄存器了，而是数据传出的寄存器，该寄存器的数据会存放到数据存储器指定的地址，步骤如下：
 - 将 *rs* 寄存器数据读出，将 16 位立即数扩展为 32 位。
 - 两个操作数放进 *ALU* 进行加法运算得真实地址。
 - 将 *rt* 寄存器数据读出，并传到数据存储器，访问数据存储器指定地址。
 - 将数据写入指定位置。
4. **分支指令**，这时 *rs* 和 *rt* 寄存器就变回像 *R* 型指令那样，都是操作数的寄存器，16 位立即数表示将跳到的指令与当前指令的差，比如，立即数为 2，即跳转到后两条指令，不过需要注意的地方有两点，一是这个差值是针对于当前 *pc+4* 的指令的，二是这个差值是以指令为单位的，而指令存储是以字节为单位的，所以真正的偏移量是差值的 4 倍。步骤如下：
 - 将 *rs* 和 *rt* 寄存器内容读出，并放到 *ALU* 做减法操作。
 - 将立即数拓展到 32 位，后左移 2 位。
 - 若减法操作为 0，则跳转；为 1，则不跳转。

c) *J* 型指令

<i>op</i> (6 位)	<i>address</i> (26 位)
-----------------	-----------------------

J 型指令各字段名称以及含义如下：

- ***op***：指令的基本操作，通常称为操作码 (*opcode*)。
- ***address***：一个 26 位表示的地址，需要译码成 32 位。

首先，我们知道，地址是用 32 位二进制表示的数，而由于指令存储器是以字节为单位，这就意味着，对于一条 32 位的指令，需要指令存储器 4 个单位来表示，而我们在表示指令的地址时，一般以 4 个单位中的第一个单位的地址为地址，所以一般来说，指令的地址必定是一个 4 的倍数，若表示为 2 进制数，则后两位必定是 0，所以 *J* 型指令中的 26 位地址，实际上表示的有 28 位的地址，最后的两位被省略了。

接着，我们得到了 28 位的地址，但是真正的地址是 32 位的，而一般跳转指令所跳转的地址范围，不会很大，肯定不会大到使得地址的前 4 位都发生了变化，所以我们把地址的前 4 位来填补即可。

直接跳转的指令不多，就 3 个，下面就这 3 个跳转的指令展开详细阐述。

1. ***j address***，这种是最纯粹的跳转，也称为直接跳转，译码后，直接将 26 位的地址拓展到 32 位，然后直接使 *pc* 等于新的地址，具体步骤如下：
 - 将 26 位的地址拓展到 32 位。
 - 令 *pc* 等于新的地址。
2. ***jal address***，这种被称为跳转和链接，一般 *jal* 后面肯定会有 *jr* 指令，*jal* 指跳到某个地址上继续执行，执行完后，跳回 *pc+4* 继续执行，通常用于函数，与 *j* 指令基本相同，只是多了一步，将 31 号寄存器 *ra* 的内容赋值为 *pc+4*，具体步骤如下：
 - 将 26 位的地址拓展到 32 位。
 - 将写寄存器 *rd* 赋值为 31 号寄存器。
 - 将 *pc+4* 写入 *rd* 寄存器。
 - 令 *pc* 等于新的地址。
3. ***jr rs***，准确来讲，*jr* 指令不属于 *J* 型指令，因为它的形式上不符合 *J* 型指令，但是它的意义确实实现跳转，所以也将它并入跳转指令，上面提到 *jal* 是用于跳转到一个函数的，

那么函数执行完，自然就会跳回原来的地方，所以在 *jal* 中我们将 $pc+4$ 的内容记录在了 31 号寄存器里面，在 *jr* 指令中，就会将 pc 赋值为 31 号寄存器的内容，实现跳转回原函数。

d) 将 *R* 型、*I* 型和 *J* 型指令放在一起看，可以得到以下结论：

- 当指令不是移位指令时，*rs* 寄存器总是作为 *ALU* 的第一个输入。
- *I* 型指令的 16 位立即数总是要被扩展，且总是作为 *ALU* 的一个输入。
- *rt* 寄存器既可以作为源操作数的寄存器，也可以作为目的寄存器。当指令为 *R* 型指令时，*rt* 作为源操作数的寄存器，当指令为 *I* 型指令时，情况比较复杂，当是存储器写操作时，*rt* 寄存器作为数据的输入端，当是分支指令时，与 *R* 型指令相同，其余的情况均作为目的寄存器。
- 若需要写入数据到寄存器，在 *R* 型指令里，这个寄存器是 *rd* 寄存器，在 *I* 型指令里，是 *rt* 寄存器，在跳转指令中，默认为 31 号寄存器，且默认写入内容为 $pc+4$ 。
- 当指令为分支指令且条件满足或指令为跳转指令时，下一个 pc 地址则是跳转地址，其他情况下一个 pc 地址均为 $pc+4$ 。

2、CPU 数据通路分析

经过上面分析后我们得到 *R*、*I* 型和 *J* 型指令有比较多的共通点，这些指令的实现大致相同，而且与具体的指令类型无关。实现每一条指令的前两步都是一样的：

- 1) 程序计数器指向指令所在的指令存储单元，并从中取出指令。
- 2) 通过指令字段内容，选择读取一个或两个寄存器。对于 *I* 型指令(非分支指令)，只需读取一个寄存器，而绝大多数指令要求读取两个寄存器。

这两步之后，为完成指令而进行的步骤则取决于具体的指令类型，但是对于算术逻辑，存储读写，分支这三种指令，其动作大致相同，与指令类型无关，例如，这三种指令都要使用 *ALU* 模块，算术逻辑指令用 *ALU* 执行运算，存储读写指令用 *ALU* 计算地址，分支指令用 *ALU* 进行比较。但在使用完 *ALU* 之后，不同指令所需的动作就有所不同了，算术逻辑指令或存储读数指令将 *ALU* 或存储器的数据写入寄存器，存储写数的指令需要访问内存，分支指令则根据比较结果决定是否改变下一条指令地址。

根据指令间的同和异，我们制定了 *CPU* 的数据通路图，如图 1，相同的部分决定了整个数据通路图的基本模块和数据流通方向，不同的部分决定了应在哪些“节点”设置多路选择器，来决定数据应该选择哪一路。

一条指令在 *CPU* 的执行步骤通常为取指令，译码，指令执行，访问存储器，结果写回这 5 个步骤，但并不是所有的指令都要全部执行完这 5 个步骤，在单周期 *CPU* 设计中，一条指令在一个时钟周期内完成，不管这条指令是否经历完这 5 个步骤，而多周期 *CPU* 中，每个时钟周期对应的是一个步骤，所以不同的指令，所需要的总时钟周期就不同，范围从 2-5 之间变化。

- 对于 *J* 型跳转的指令，就只有 *IF* 和 *ID* 这两个阶段，因为译码后就可以立刻知道跳转的地址是什么，就可以开始进行下一条指令了。
- 对于 *R* 型指令和一些设计立即数的算术逻辑指令，包含 *IF*、*ID*、*EXE* 和 *WB* 四个步骤。
- 对于分支指令，包含 *IF*、*ID*、*EXE* 三个步骤，因为两个源操作数经过 *ALU* 后便可知道是否相等，也就可以确定下一条指令的地址了。
- 对于 *sw* 指令，包含 *IF*、*ID*、*EXE*、*MEM* 四个步骤，*EXE* 阶段计算存储器地址，*MEM* 阶段将数据存储到存储器。
- 对于 *lw* 指令，包含 *IF*、*ID*、*EXE*、*MEM*、*WB* 五个步骤，是最长的指令。

从上面这五种情况可以看出，任何指令都包含 *IF* 和 *ID* 两个步骤，而对于不同类型的指令，尽管是同一个步骤，但是由于下一个步骤不一样，我们不可以把它们看成是同一个状态，根据状态的转移，我们设计下面的有限状态机来表示这几种指令的状态转移，如图 1。

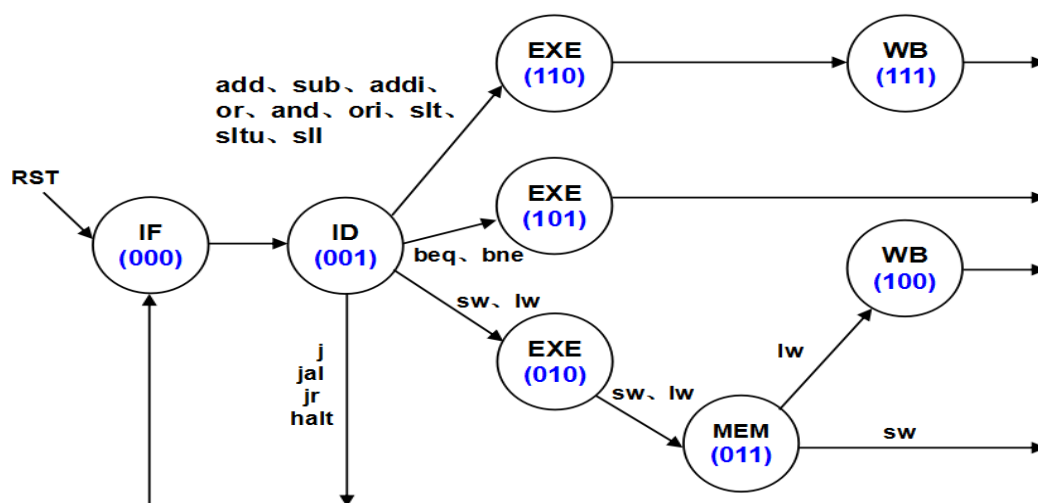


图 1：状态转移图

对于状态机，下一个状态的得到是根据当前状态以及指令类型的，而另外一点与单周期 *CPU* 不同的是，在多周期 *CPU* 的设计中，由于一条指令分成了好几个周期执行，所以我们不能像单周期 *CPU* 那样一下子把所有的控制信号都发出来，而是要根据当前的状态，发送对应的控制信号，这样设计的原因主要是出于这三个控制信号的问题：*PCWre*、*RegWre*、*dataInsWrite*，这三个变量后两者都是寄存器或存储器的写操作，假若，我们不是在 *WB* 或者 *MEM* 阶段才赋值这两个变量，而是在其他的阶段就令他们为 1 的话，那么数据就会不断地被写进寄存器或者存储器，这可能会改变我们原来的数据；而 *PCWre* 是指令读取操作，在单周期 *CPU* 中，一个周期执行一条指令，而读取指令的操作被限定在了时钟的上升沿，而多周期中，若 *PCWre* 一直为高电平，那么就会每个周期都读取一条指令，尽管上一条指令还没完成，就已经改变了指令和操作码，这样我们的状态机就无法工作，因为状态的转移是基于当前状态和操作码的，指令改变了，操作码就改变，状态就无法转移。

而对于另外的控制信号，因为不涉及改变存储器内容，所以没有太大的影响，就比如 *ALUOp*，尽管不是在 *EXE* 阶段，*ALU* 在计算不会造成影响，只要结果不写入即可。

所以我们可以发现，但凡涉及使能的控制信号，即低电平禁止，高电平允许的，都是设计多周期 *CPU* 关键的控制信号。

下面图 2 是多周期 *CPU* 的数据通路图，考虑了新的指令类型以及多周期 *CPU* 设计的需要，我们在每个阶段都增添了一个寄存器来储存每个状态的数据，然后在每个时钟的上升沿，刷新寄存器。另外考虑了新添的一些指令，如移位和跳转指令，所以增添了 *ALUSrcA* 和 *WrRegDSrc* 的

选择器以及地址的拓展和 *PC* 选择器。同样的，我们规定 *PC* 刷新、状态转移以及寄存器刷新都在时钟的上升沿，而数据的写入在时钟的下降沿。

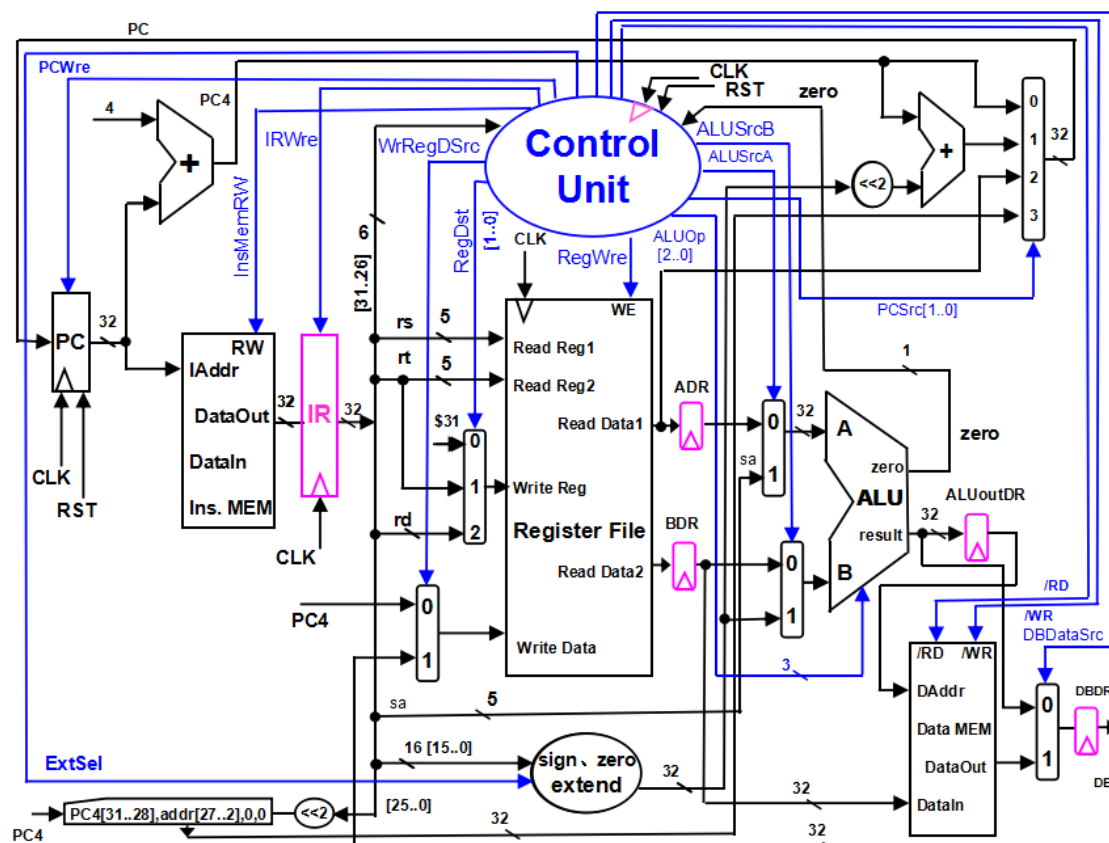


图 2：数据通路图

下表是控制信号的表格，表格内容为两个信息，按照 (信号值/改变时的状态) 来表示，正如上面提到，对于使能的那几个控制信号，*RegWre*、*DataMemWrite* 和 *PCWre*，我们都必须保证它只在对应的状态被赋值为 1，而其他的状态时被赋值为 0，对于 *RegWre* 我们只在 *WB* 状态下才赋值为 1，对于 *DataMemWrite* 我们只在 *MEM* 阶段下才赋值为 1，而对于 *PCWre*，因为 *PC* 模块中，指令的读取是在时钟的上升沿进行的，而状态的切换也是在上升沿进行的，所以我们不可以将 *PCWre* 放到 *ID* 状态下进行修改，因为这样的话，*PCWre* 只会在下一个时钟周期才会生效，所以我们采取在前一条指令的最后一个状态将 *PCWre* 设为 1，在下一条指令的第一个状态将 *PCWre* 设为 0。而对于另外那两个控制信号，因为写入的操作是在时钟的下降沿才进行的，所以我们没有必要像 *PCWre* 这样去处理这两个控制信号。

而对于其他的控制信号，因为不改变存储器的数据，并且不带使能性，所以不需要像上面这样来处理，从下表可以验证到，新添的那几个选择器什么时候用到，*sll* 指令中，*ALUSrcA* 选择端为 1，*jr* 指令中，*RegDst* 选择端为 0，*WRRegDSrc* 选择端为 0。

表 1：控制信号表

指令 码	状态 转移	Reg Dst	WRRe gDsrc	Reg Wre	Ext Sel	ALU SrcA	ALU SrcB	ALU op	DBda taSrc	DataMe mWrite	DataMe mRead	PC Src	PC Wre
<i>add</i>	0.1.6.7	2/1	1/7	0/0 1/7	X	0/6	0/6	0/6	0/6	0/0	X	0/1	0/0 1/7
<i>sub</i>		2/1	1/7	0/0 1/7	X	0/6	0/6	1/6	0/6	0/0	X	0/1	0/0 1/7
<i>addi</i>		1/1	1/7	0/0 1/7	1/1	0/6	1/6	0/6	0/6	0/0	X	0/1	0/0 1/7
<i>or</i>		2/1	1/7	0/0 1/7	X	0/6	0/6	5/6	0/6	0/0	X	0/1	0/0 1/7
<i>and</i>		2/1	1/7	0/0 1/7	X	0/6	0/6	6/6	0/6	0/0	X	0/1	0/0 1/7
<i>ori</i>		1/1	1/7	0/0 1/7	1/1	0/6	1/6	5/6	0/6	0/0	X	0/1	0/0 1/7
<i>slt</i>		2/1	1/7	0/0 1/7	X	0/6	0/6	3/6	0/6	0/0	X	0/1	0/0 1/7
<i>sltu</i>		2/1	1/7	0/0 1/7	X	0/6	0/6	2/6	0/6	0/0	X	0/1	0/0 1/7
<i>sll</i>		2/1	1/7	0/0 1/7	X	1/6	0/6	4/6	0/6	0/0	X	0/1	0/0 1/7
<i>beq</i>	0.1.5	2/1	X	0/0	1/1	0/5	0/5	1/5	X	0/0	X	zero	0/0 1/5
<i>bne</i>		2/1	X	0/0	1/1	0/5	0/5	1/5	X	0/0	X	zero	0/0 1/5
<i>sw</i>	0.1.2.3	X	X	0/0	1/1	0/2	1/2	0/2	X	0/0 1/3	X	0/1	0/0 1/3
<i>lw</i>	0.1.2.3.4	1/1	1/4	0/0 1/4	1/1	0/2	1/2	0/2	1/3	0/0	1/3	0/1	0/0 1/4
<i>j</i>	0.1	X	X	0/0	X	X	X	X	X	0/0	X	3/1	0/0 1/1
<i>jal</i>		0/1	0/1	0/0 1/1	X	X	X	X	X	0/0	X	3/1	0/0 1/1
<i>jr</i>		X	X	0/0	X	X	X	X	X	0/0	X	2/1	0/0 1/1

五. 实验设计

1、寄存器组模块

首先需要设定一个 32×32 的存储器来代表 32 个 32 位的寄存器，并且要将 0 号寄存器初始化为 0。寄存器中数据的读取是组合逻辑电路，而数据的写入是时序逻辑电路，需要在时钟的下降沿才能进行，并且要在控制信号 *RegWre* 为 1 的时候才允许写入操作，所以，我们必须控制不是写入的时候，就不能使 *RegWre* 赋值为 1。具体代码如下图 3：

```

always@ (*) begin // read
    RD_data1 = registers[RD_reg1_addr];
    RD_data2 = registers[RD_reg2_addr];
end

always@ (negedge clk) begin // write
    if( RegWre == 1 && WR_reg_addr != 0)
        registers[WR_reg_addr] = WR_data;
end

```

图 3：寄存器模块

2、数据存储器模块

对于数据存储器，是存储数据的地方，但是不同于寄存器的地方时，它是以字节为单位的，即存储器的一行只有 8 位，所以存储一个 32 位的数据需要 4 行，这也可以解释到，为什么 *sw* 或 *lw* 指令时偏移量总是 4 的倍数。

数据存储器有写入和读取操作，对应 *load* 和 *save* 指令，当需要写入数据的时候，写入控制信号设为 1，并且只能在时钟下降沿进行，但对于那些不需要访问存储器的指令，我们必须要把写入控制信号设为 0，不允许写入操作，因为写入操作会改变存储器的数据，而读取操作不会改变存储器中的数据，读取出来后，由于后面的选择器不会选择存储器中读出的数据，所以也就没有影响，代码如下图 4：

```
always@ (negedge clk) begin
    if(DataMemWrite == 1) begin
        data[data_addr] = data_in[31:24];
        data[data_addr + 1] = data_in[23:16];
        data[data_addr + 2] = data_in[15:8];
        data[data_addr + 3] = data_in[7:0];
    end
end

always@ (*) begin
    if(DataMemRead == 1)
        memory_out = {data[data_addr], data[data_addr + 1],
            data[data_addr + 2], data[data_addr + 3]};
    end
```

图 4：数据存储器模块

3、控制单元模块

控制单元在上表各种控制信号的值以及对应的状态已经给出，这里不加以赘述。下面给出状态的转移函数，状态的转移是基于当前的状态和指令的操作码，代码如下图 5：

```
case(current_state)
    0 : next_state = 1;
    1 : begin
        if(op == 0 || op == 1 || op == 2 || op == 16 || op == 17 ||
            op == 18 || op == 38 || op == 39 || op == 24)
            next_state = 6;
        else if(op == 52 || op == 53)
            next_state = 5;
        else if(op == 48 || op == 49)
            next_state = 2;
        else if(op == 56 || op == 57 || op == 58 || op == 63)
            next_state = 0;
        end
    2 : next_state = 3;
```

```

3 : begin
    if (op == 48)
        next_state = 0;
    else
        next_state = 4;
    end
4 : next_state = 0;
5 : next_state = 0;
6 : next_state = 7;
7 : next_state = 0;
default : next_state = 0;

```

图 5: 状态转移模块

4、PC 选择模块

正常来说，一条指令执行完了，PC 就会加 4，到下一条指令，如果指令为分支指令，条件满足才会进行跳转，此时 PCSrc 会等于 1，而如果指令为 *j* 指令或者 *jal* 指令，PCSrc 会等于 3，地址会等于拓展后的地址，而如果指令为 *jr* 指令，PCSrc 会等于 2，地址会等于 31 号寄存器的内容，具体代码如下图 6:

```

always@ (*) begin
    if (PCSrc == 0 )
        nextPC = curPC + 4;

    else if (PCSrc == 1 ) begin
        nextPC = curPC + 4;
        nextPC = nextPC + ext_immediate_shift;
    end

    else if (PCSrc == 2 )
        nextPC = RD_data1;

    else if (PCSrc == 3 )
        nextPC = jumpAddr_extend;
    end
end

```

图 6: PC 模块

六. 实验结果分析

本次实验采用以下指令表，并将这个指令表以二进制格式读进指令存储器。

表 2：指令表

地址	汇编程序	指令代码					
		op(6)	rs(5)	rt(5)	rd(5)/immediate(16)	16 进制数代码	
0x00000000	addi \$1,\$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008
0x00000004	ori \$2,\$0,2	010010	00000	00010	0000 0000 0000 0010	=	48020002
0x00000008	or \$3,\$2,\$1	010000	00010	00001	0001 1000 0000 0000	=	40411800
0x0000000c	sub \$4,\$3,\$1	000001	00011	00001	0010 0000 0000 0000	=	04612000
0x00000010	and \$5,\$4,\$2	010001	00100	00010	0010 1000 0000 0000	=	44822800
0x00000014	sll \$7,\$5,2	011000	00000	00101	0011 1000 1000 0000	=	60053880
0x00000018	bne \$7,\$1,-2	110101	00111	00001	1111 1111 1111 1110	=	d4e1fffe
0x0000001c	jal 0x0000040	111010	00000	00000	0000 0000 0001 0000	=	e8000010
0x00000020	slt \$8,\$12,\$1	100110	01100	00001	0100 0000 0000 0000	=	99814000
0x00000024	addi \$14,\$0,-1	000010	00000	01110	1111 1111 1111 1111	=	080effff
0x00000028	slt \$9,\$8,\$14	100110	01000	01110	0100 1000 0000 0000	=	990e4800
0x0000002c	sltu \$10,\$9,\$2	100111	01001	00010	0101 0000 0000 0000	=	9d225000
0x00000030	sltu \$11,\$10,\$0	100111	01010	00000	0101 1000 0000 0000	=	9d405800
0x00000034	add \$11,\$11,\$8	000000	01011	01000	0101 1000 0000 0000	=	01685800
0x00000038	bne \$11,\$2,-2	110101	01011	00010	1111 1111 1111 1110	=	d562fffe
0x0000003c	j 0x000004c	111000	00000	00000	0000 0000 0001 0011	=	e0000013
0x00000040	sw \$2,4(\$1)	110000	00001	00010	0000 0000 0000 0100	=	c0220004
0x00000044	lw \$12,4(\$1)	110001	00001	01100	0000 0000 0000 0100	=	c42c0004
0x00000048	jr \$31	111001	11111	00000	0000 0000 0000 0000	=	e7e00000
0x0000004c	halt	111111	00000	00000	0000000000000000	=	fc000000

经过指令后，我们可以得到\$0 - \$7寄存器的值如下表：

\$0	\$1	\$2	\$3	\$4	\$5	\$7	\$8	\$9	\$10	\$11	\$12	\$14
0	08	02	0a	02	02	08	01	00	01	00->01->02	02	-1

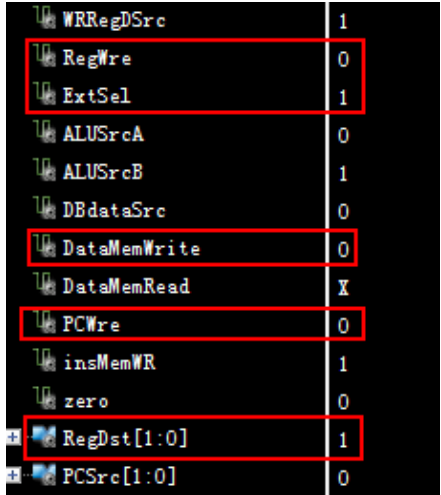
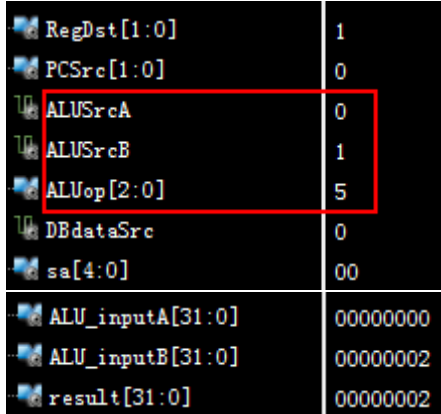
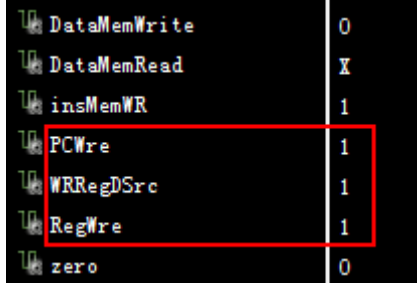
Vivado 仿真后的寄存器结果如下图 7，验证了上表的结论：

[7][31:0]	00000008	Array	[14][31:0]	ffffffff	Array
[6][31:0]	XXXXXXXX	Array	[13][31:0]	XXXXXXXX	Array
[5][31:0]	00000002	Array	[12][31:0]	00000002	Array
[4][31:0]	00000002	Array	[11][31:0]	00000002	Array
[3][31:0]	0000000a	Array	[10][31:0]	00000001	Array
[2][31:0]	00000002	Array	[9][31:0]	00000000	Array
[1][31:0]	00000008	Array	[8][31:0]	00000001	Array

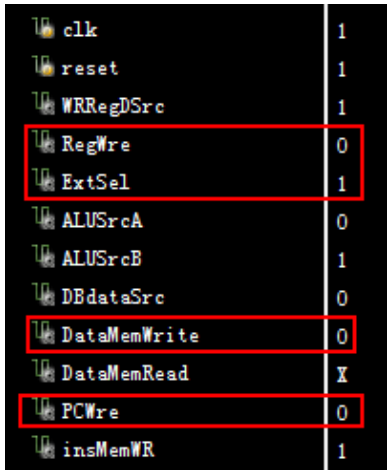
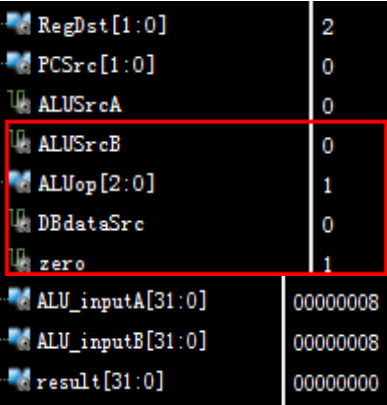
图 7：寄存器数据

下面就 4 条指令，我们来分析以下仿真的结果，与理论是否符合。

- *R* 型指令 *ori* \$2,\$0,2, 指令 16 进制表示为 48020002, 指令地址为 0x00000004, 这是一条 *R* 型指令, 状态的转移为 0->1->6->7, 结果如下表,

<p>状态 1: 控制模块得到指令操作码, 然后发出关于寄存器组的一些控制信号, 可以看出状态 1 下, 几个写使能信号都为 0, <i>PCWre</i> 也为 0, 而 <i>RegDst</i> 为 1</p>	
<p>状态 6: <i>EXE</i> 阶段, 寄存器刷新并且控制模块发出 <i>ALUSrc</i> 的信号, 使得到两个 <i>ALU</i> 的输入, 然后发出 <i>ALUop</i> 计算结果, 对于这种立即数的指令, <i>ALUSrcB</i> 为 1, <i>ALUSrcA</i> 为 0, 因为没有移位, 得到两个输入分别为 0 和 2, 结果为 2, 计算出来结果后, 由于马上就要进入写回阶段, 所以 <i>DBDataSrc</i> 也要马上选择结果放到寄存器准备刷新</p>	
<p>状态 7: 结果需要写回, 所以 <i>RegWre</i> 就被赋值为 1, <i>WRDataSrc</i> 赋值为 1, 表示选择 <i>ALU</i> 或者存储器来的数据, 由于这是这条指令最后的阶段, 所以还需要把 <i>PCWre</i> 设为 1, 为下一条指令做准备。</p>	

- 分支指令, *bne \$7,\$1,-2*, 指令译码为 *d4e1ffe*, 状态的转移是 *0->1->5*, 仿真结果如下:

状态 1: 控制模块得到指令操作码, 然后发出关于寄存器组的一些控制信号, 由于分支指令不需要写回, 所以 <i>RegDst</i> 随便, 同样, 几个写使能信号都为 0	
状态 5: <i>EXE</i> 阶段, 寄存器刷新并且控制模块发出 <i>ALUSrc</i> 的信号, 使得到两个 <i>ALU</i> 的输入, 然后发出 <i>ALUop</i> 计算结果, 对于分支指令, 需要比较两个数的大小, 然后发出 <i>zero</i> 信号, 再确定 <i>PCSrc</i> 的值, 比较的两个数为 8 和 8, 两者相等, 所以 <i>zero</i> 为 1, 由于这个是最后一个阶段, 所以 <i>PCWre</i> 也要设为 1 为下一条指令做准备	

- *load* 指令 *lw \$12,4(\$1)*, 指令译码为 *c42c0004*, 状态转移为 *0->1->2->3->4*, 仿真结果如下:

状态 1: 控制模块得到指令操作码, 然后发出关于寄存器组的一些控制信号, 由于是 <i>load</i> 指令, 所以需要结果写回, 所以 <i>RegDst</i> 为 1, 同时设计立即数, 所以 <i>ExtSel</i> 为 1, 同样, 几个写使能信号都为 0	
状态 2: <i>EXE</i> 阶段, 寄存器刷新并且控制模块发出 <i>ALUSrc</i> 的信号, <i>ALUSrcB</i> 为 1, 所以得到立即数拓展的结果, 然后计算两者之和得到地址, 注意 <i>DbdataSrc</i> 信号不在这个阶段发出。	

状态 3: 访问存储器, *ALU* 得到的真正结果输入到存储器的 *addr_in*, 然后控制信号 *DataMemRead* 设为 1, 这时还需设置 *DbdataSRC* 为 1 表示选择存储器的数据。

<i>ALUop[2:0]</i>	0
<i>DataMemRead</i>	1
<i>DBdataSrc</i>	1
<i>zero</i>	0
<i>sa[4:0]</i>	00

状态 4: 写回阶段, 所以 *RegWre* 就被赋值为 1, *WRDataSrc* 赋值为 1, 表示选择 *ALU* 或者存储器来的数据, 由于这是这条指令最后的阶段, 所以还需要把 *PCWre* 设为 1, 为下一条指令做准备。

<i>insMemWR</i>	1
<i>PCWre</i>	1
<i>WRRegDSrc</i>	1
<i>RegWre</i>	1
<i>RegDst[1:0]</i>	1
<i>PCSrc[1:0]</i>	0

- 跳转指令 *jal 0x0000040*, 指令译码为: *e8000010*, 状态转移为 0->1, 仿真结果如下,

状态 1: 将写寄存器设为 31 号寄存器, 所以 *RegDst* 设为 0, 写的内容设置为 *pc+4*, 所以 *WRdataSrc* 设为 0, 然后 *PCSrc* 设为 3, 表示下一个 *PC* 的结果是地址拓展的结果。同时由于这是最后一步, 所以 *PCWre* 需要设为 1

<i>insMemWR</i>	1
<i>PCWre</i>	1
<i>WRRegDSrc</i>	0
<i>RegWre</i>	1
<i>RegDst[1:0]</i>	0
<i>PCSrc[1:0]</i>	3
<i>ALUSrcA</i>	0

七. 实验心得

这次的多周期 *CPU* 的设计, 总体来说, 是在单周期 *CPU* 上进行修改的, 一开始觉得这个多周期的 *CPU* 并没有太难, 只用在单周期 *CPU* 上, 加几个寄存器, 改下控制模块就好, 但是当我真正做的时候发现, 其实还是有很多的细节需要考虑到。

第一, 如何统一状态的转换和寄存器的刷新以及数据的写入是在上升沿还是下降沿, 一开始我想加入状态的转移和寄存器的刷新都同时在上升沿会不会出现什么问题, 因为大家是同时发出的, 害怕组合逻辑中会出什么错, 但是后来自己也验证了, 是不会出问题, 所以就决定了状态转移和寄存器刷新在时钟上升沿, 而数据的写入在时钟的下降沿。

这个问题解决了之后, 我就遇到第二个问题了, 那么我们要根据不同的状态来发出不同的控制信号, 我又害怕一些不是这个状态的控制信号会影响 *CPU* 的运转, 后来就想到, 只要不是写入的信号或者是使能的信号就不会产生影响, 因为, 尽管 *ALU* 一直在算, 选择器一直在选, 但是这些操作并不会影响存储器或者寄存器组, 所以只要控制好写入信号就好, 所以我一开始就把两个的写入的使能信号在状态 0 的时候全部赋值为 0, 而在具体需要写入的阶段才赋值为 1, 这没有问题, 而问题是 *PCWre*, 我一开始 *PCWre* 是在状态 0 赋值为 1 的, 而我们知道 *PC* 读取指令是在时钟的上升沿, 而状态的转移也在上升沿, 而读取指令的条件是 *PCWre* 等于 1, 两者同时在上升沿进行, 但是 *PCWre* 在上一个时钟为 0, 而尽管在时钟上升沿把它赋值为 1 了, 也只能在下一个时钟才会读取指令了, 所以这样就导致了状态紊乱了, 后来我就打算把 *PCWre* 的赋值放在每一条指令的最后一步, 这样就能保证下一条指令在读取指令的时候是允许的, 而这个问题也解决了。

一个学期的计算机组成原理实验就要结束了, 从一开始接触汇编指令, 接触机器码, 开始了解我们代码背后的东西, 到现在完成了多周期 *CPU* 的设计, 这过程其实不算艰苦, 实验的时长有点长, 3 个星期, 但其实很多人都是到最后一个星期才开始做, 并且验收还持续两个星期, 这样有些人就更加的拖延了, 并且, 一个实验除了第一节课需要讲课之外, 其它的两节课都是不讲课的, 但是我们又必须去到实验室只是为了签到, 这样的安排感觉不太合理, 希望老师下学期可以改变一下, 或者把实验密度提上去, 理论课上还提到了流水的 *CPU* 设计, 但是这个在实验上如果要实现的话, 还是很困难的, 所以觉得老师没有让我们设计流水线是一个明智的决定。