****

**《计算机组成原理实验》**

**实验报告**

**（实验二）**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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 | 4 | 月 | 23 | 日 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | **成绩** | **:** |  |
| **实验二** | **：** | **单周期*CPU*设计** | | | |

* + - 1. **实验目的**

1. 掌握单周期*CPU*数据通路图的构成、原理及其设计方法；
2. 掌握单周期*CPU*的实现方法，代码实现方法；
3. 认识和掌握指令与*CPU*的关系；
4. 掌握测试单周期*CPU*的方法。
   * + 1. **实验内容**

设计一个单周期*CPU*，该*CPU*至少能实现算术逻辑运算、存储读取、分支指令功能操作。需设计的指令格式包括*R*型指令以及*I*型指令。

* + - 1. **实验器材**

电脑一台、*Xilinx ISE* 软件一套

* + - 1. **实验原理**

1. **指令格式分析**
2. ***R*型指令**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***op*(6位)** | ***rs*(5位)** | ***rt*(5位)** | ***rd*(5位)** | **reserved(11位)** |

*R*型指令各字段名称及含义如下：

* ***op*：**指令的基本操作，通常称为操作吗(*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，所以执行*R*型操作只需以下3步：

* 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。

1. ***I*型指令**

|  |  |  |  |
| --- | --- | --- | --- |
| ***op*(6位)** | ***rs*(5位)** | ***rt*(5位)** | ***constant or address*(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，通常称为操作吗(*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。

1. **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。

1. **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。

1. **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。

1. **将*R*型和*I*型指令放在一起，可以得到以下结论：**

* *rs*寄存器总是作为*ALU*的一个输入。
* *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

1. ***CPU*数据通路分析**

经过上面分析后我们得到*R*和*I*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。

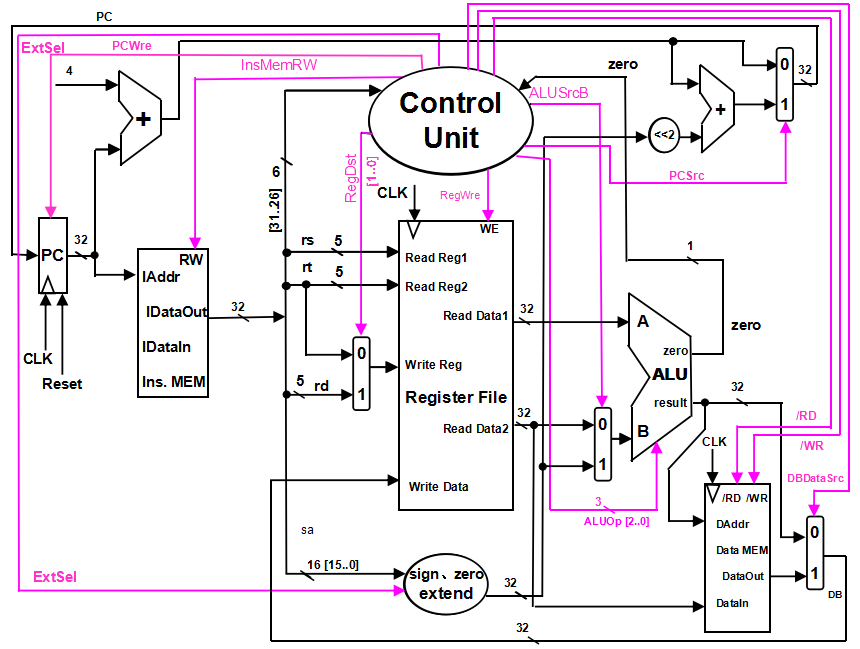


图1：数据通路图

从数据通路图可以看出，一条指令在*CPU*的执行步骤如下：

1. 取指令：根据*pc*中的指令地址，在指令存储器中读取32位指令。
2. 译码：首先判断*op*从而判断是*R*型指令还是*I*型指令，再根据不同的*op*，发出不同的信号到各个选择器模块和存储器模块。
3. 指令执行：取出数据，立即数拓展，*ALU*操作，更新程序计数器。
4. 访问存储器：若指令需要对存储器进行读写操作，则在这一步进行，若不需要，则跳过。
5. 结果写回：指令执行的结果或存储器读出的数据，将在这一步写入到寄存器。分支操作则不需要进行这一步。

不同的指令所经历的步骤不相同，有些指令会执行完所有的5步，而有些指令可能只会执行3到4步，所以这种时间上的差异迫使我们加入延时的操作，即在每个时钟的上升沿进行取指令操作，在每个时钟的下降沿进行写入数据操作，写入数据包括两个地方，结果写回到寄存器，或数据写入存储器

对于不同的指令，所需要的信号是不同的，但是信号的总数是不变的，所以当不需要一些信号的时候，它的值是对指令执行没有影响的，比如add指令中，不需要进行立即数的拓展，所以立即数拓展的控制信号无论是0是1都对其无影响，因为在*ALU*输入信号中已经不会选择立即数作为输入。下面对于不同的指令，其对应信号取值如下表1所示：

**表1：控制信号表**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *PCWre,insMemWR*设初值为1 | | | | | | | | | | |
| 操作 *op* *RegDst* *RegWre* *ALUSrcB* *ALUop* *PCSrc* *ExtSel* *DBdataSrc* *DataMemRW* *PCWre* | | | | | | | | | | |
| *add* | 000000 | 1 | 1 | 0 | 0 | 0 | / | 0 | 0 | 1 |
| *addi* | 000001 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| *sub* | 000010 | 1 | 1 | 0 | 1 | 0 | / | 0 | 0 | 1 |
| *and* | 010001 | 1 | 1 | 0 | 4 | 0 | 1 | 0 | 0 | 1 |
| *sw* | 100110 | / | 0 | 1 | 0 | 0 | 1 | 1 | 1*(write)* | 1 |
| *lw* | 100111 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0*(read)* | 1 |
| *beq* | 110000 | / | 0 | 0 | 1 | 1 | 1 | / | 0 | 1 |
| *ori* | 010000 | 0 | 1 | 1 | 3 | 0 | 0 | 0 | 0 | 1 |
| *or* | 010010 | 1 | 1 | 0 | 3 | 0 | / | 0 | 0 | 1 |

* + - 1. **实验设计**

1. **寄存器模块**

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，所以对于*sw*或分支指令，是不允许进行写入操作的，具体代码如下图2：

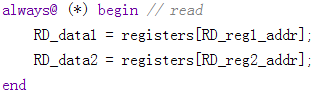
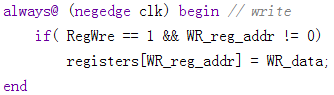
 

图2：寄存器模块

1. **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，控制信号设为1，并且只能在时钟下降沿进行，而读取操作时，控制信号设为0，可以直接读取，但对于那些不需要访问存储器的指令，我们把控制信号设为0，即读取操作，而不是设为1，写入操作，因为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3：

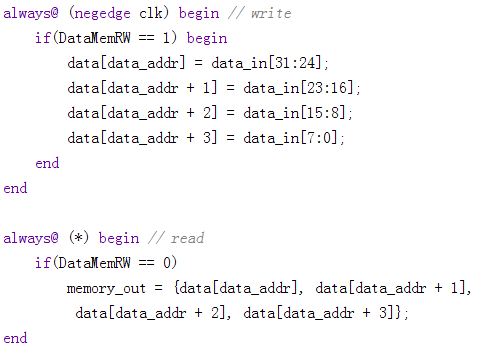


图3：数据存储器模块

1. **控制单元模块**

控制单元在上表各种控制信号已经给出，这里不加以赘述。

1. ***PC*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，当分支指令，条件满足才会进行跳转，所以我们在控制单元中，判断到时分支指令后，就先把*PRWre*设为1，然后再根据*zero*的值进行调整，代码如下，这里特别注意的是，两个连续赋值用的是阻塞赋值，因为这是有顺序的赋值，而其他地方，阻塞和非阻塞均可，如下图4：

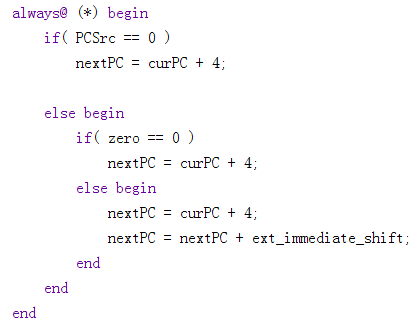


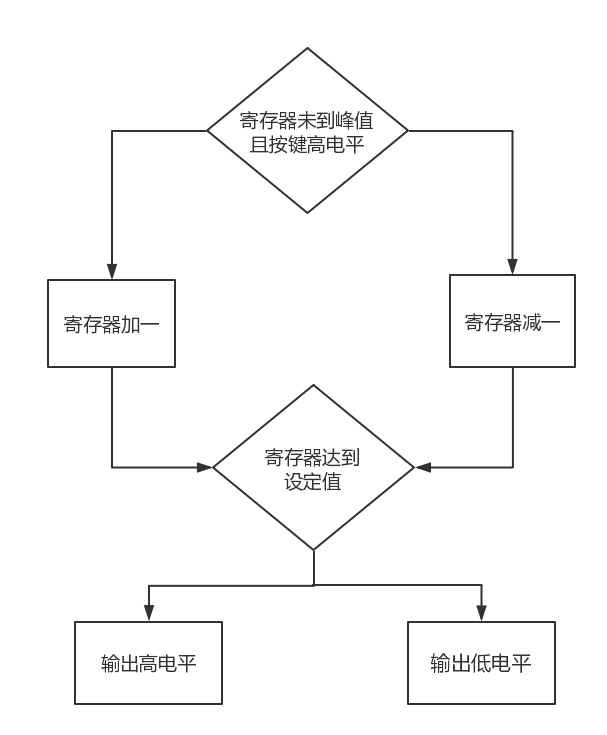
图4：*PC*模块

1. **按键消抖模块**

由于板子上的按钮都是机械按钮，不可避免的会有抖动，所以我们必须采取消抖，消抖的方式有硬件消抖和软件消抖，软件消抖就是采用*if*语句和延时，频繁地检测按键电平的变化，会有一定概率出错。硬件消抖的原理是在按键上加一个电容，当按键电平变化时，传输到*CPU*的信号不会马上发生变化，而是伴随一个电容充电的过程。当*CPU*接收到按键电平时，按键已经经过了抖动的阶段。

具体实现的方式是：利用寄存器来模拟电容充放电的过程，当按键的寄存器还没达到设定的数值并且按钮处于高电平，那电容充电(寄存器加一)，若处于低电平，则电容放电(寄存器减一)；如果寄存器的值达到一定的数值，就输出高电平，否则输出低电平。

当我们按下一个按钮时，一段时间内，高电平所占的比例肯定比低电平要高，所以电容会缓慢的充电到指定的数值，然后输出1；当我们松开一个按钮时，同理，低电平所占的比例会较高，所以电容会缓慢放电，直至低于指定数值，然后输出0。这段缓慢上升和缓慢下降的阶段就是我们所说的抖动阶段，流程图和代码如下图5：



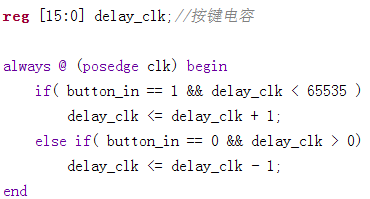
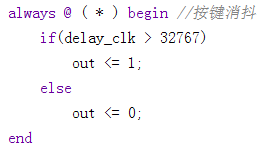
 

图5：按键消抖模块

1. **七段码显示模块**

首先我们应该考虑数码管需要显示什么数字，第一需要显示当前*PC*和下一条指令的*PC*，第二显示*ALU*两个输入，第三显示*ALU*的输出，第四还要显示寄存器的输出值，第五还要显示各个寄存器的值，所以需要开关来进行切换。

其次就是显示电路的原理了，扫描显示，逐一点亮，是一个位置决定内容的点亮方式，一个控制变量从0-3变化分别对应点亮1-4位，这一部分相当于一个194，而对应每一个数字对应数码管每一段的情况，就相当于48，模块图如下图6：

可以看出*a\_to\_g*每一位都对应一个*LUT，LUT*的输入就是4位二进制数。

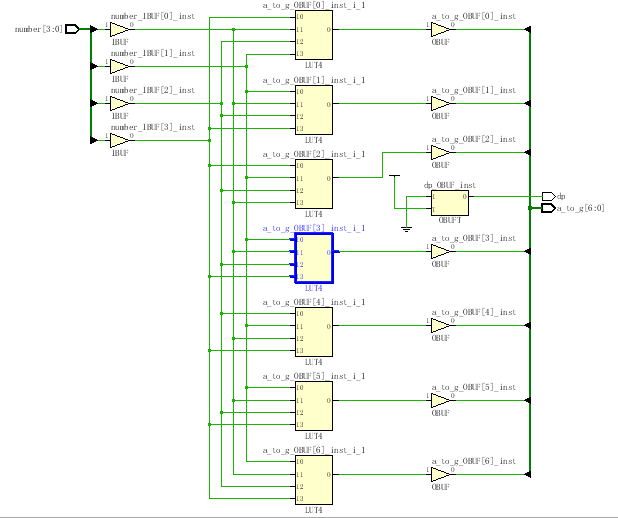
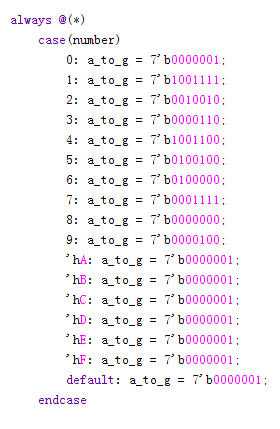


图6：七段码模块

* + - 1. **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** | ori $1,$0,7 | 010000 | 00000 | 00001 | 0000 0000 0000 0111 | = | 40010007 |
| **0x00000004** | addi $2,$0,9 | 000001 | 00000 | 00010 | 0000 0000 0000 1001 | = | 04020009 |
| **0x00000008** | add $3,$1,$2 | 000000 | 00001 | 00010 | 0001 1000 0000 0000 | = | 00221800 |
| **0x0000000c** | sub $4,$2,$1 | 000010 | 00010 | 00001 | 0010 0000 0000 0000 | = | 08412000 |
| **0x00000010** | or $5,$2,$4 | 010010 | 00010 | 00100 | 0010 1000 0000 0000 | = | 48442800 |
| **0x00000014** | beq $1,$4,5 | 110000 | 00001 | 00100 | 0000 0000 0000 0101 | = | c0240005 |
| **0x00000018** | sw $1, 3($2) | 100110 | 00010 | 00001 | 0000 0000 0000 0011 | = | 98410003 |
| **0x0000001c** | and $6,$4,$5 | 010001 | 00100 | 00101 | 0011 0000 0000 0000 | = | 44853000 |
| **0x00000020** | move $7,$1 | 100000 | 00001 | 00000 | 0011 1000 0000 0000 | = | 80203800 |
| **0x00000024** | lw $4,3($2) | 100111 | 00010 | 00100 | 0000 0000 0000 0011 | = | 9c440003 |
| **0x00000028** | beq $4,$7,-6 | 110000 | 00100 | 00111 | 1111 1111 1111 1010 | = | c087fffa |
| **0x0000002c** | addi $2,$3,1 | 000001 | 00011 | 00010 | 0000 0000 0000 0001 | = | 04620001 |
| **0x00000030** | halt | 111111 | 00000 | 00000 | 0000 0000 0000 0000 | = | 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 | $6 | $7 |
| 0 | 07 | 09->11 | 10 | 02->07 | 0b | 02 | 07 |

Vivado仿真后的寄存器结果如下图7，验证了上表的结论：

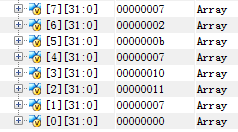
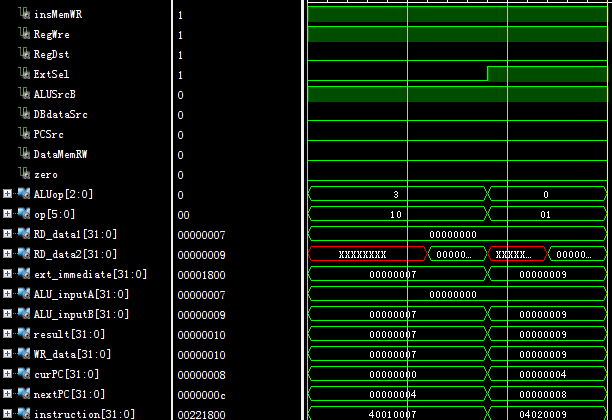


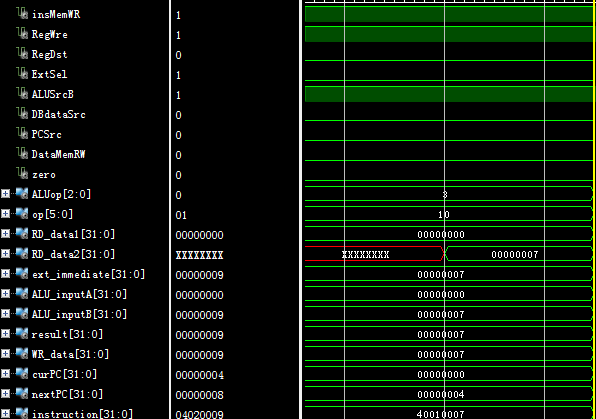
图7：寄存器数据

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。

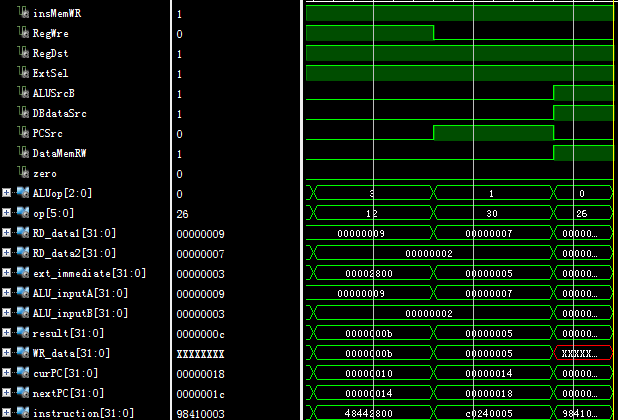
* *R*型指令 add $3,$1,$2，指令16进制表示为00221800，指令地址为0x00000008，下一条指令地址为0x0000000c，由于是*R*型指令，目的寄存器地址是*rd*，所以*RegDst* = 1，*RegWre* = 1，*ALU*的两个输入都来自于寄存器的输出，所以*ALUSrcB* = 0，运算结果返回到寄存器，所以*DBDataSrc* = 0，下图8得以验证，



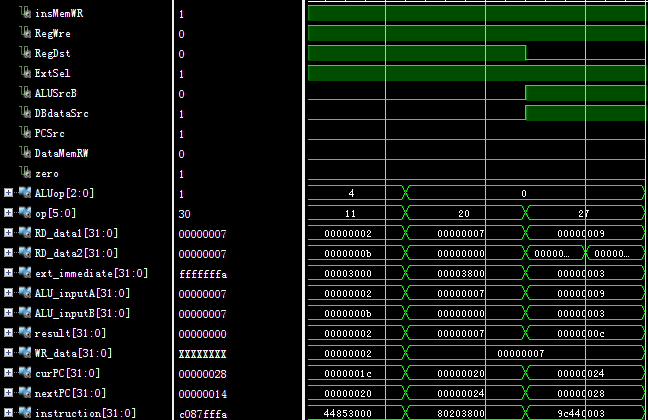
* *I*型立即数指令 addi $2,$0,9，指令译码为04020009，由于涉及立即数，所以*ExtSel* = 1，*ALU*的第二个输入来自于扩展的立即数，所以*ALUSrcB* = 1，由于不用访问数据存储器，所以写入寄存器的数据来源于*ALU*的结果，所以*DBDataSrc* = 0，*PCWre =* 1表示需要写入，而由于是*I*型指令，目的寄存器来自*rt*，所以*RegDst* = 0，仿真结果如下：



* 存储指令 sw $1, 3($2)，指令译码为98410003，由于数据不需要写入寄存器，所以*PCWre* = 0，*ALU*用来进行地址的计算，*ALU*的第二个输入来自立即数，所以*ALUSrcB* = 1，需要写入数据存储器，所以*DataMemRW* = 1。



* 分支指令 beq $4,$7,-6，指令译码为：c087fffa，地址为0x00000028，若条件成立，则跳转至28 + 4 – 6\*4 = 14，所以*ALU*需要进行减法，两个输入对应两个寄存器，所以*ALUSrcB* = 0，并且不需要写入，所以*RegWre* = 0，由于是分支指令，所以*PCSrc* = 1，若*zero* = 1则跳转，仿真结果



下面提供一个表格，这个表格是将*CPU*烧进*FPGA*板的时候，在七段码中显示的数字含义，00显示的是当前*PC*和下一时刻的*PC*，01代表*ALU*的结果和数据存储器的输出，10代表*ALU*的两个输入，11代表两个读寄存器的内容。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **地址 汇编程序 00 01 10 11** | | | | | |
| **0x00000000** | ori $1,$0,7 | 00 04 | 07 00 | 00 07 | 00 00 |
| **0x00000004** | addi $2,$0,9 | 04 08 | 09 00 | 00 09 | 00 00 |
| **0x00000008** | add $3,$1,$2 | 08 0c | 10 00 | 07 09 | 07 09 |
| **0x0000000C** | sub $4,$2,$1 | 0c 10 | 02 00 | 09 07 | 09 07 |
| **0x00000010** | or $5,$2,$4 | 10 14 | 0b 00 | 09 02 | 09 02 |
| **0x00000014** | beq $1,$4,5 | 14 18 | 05 00 | 07 02 | 07 02 |
|  |  | 14 2c | 00 00 | 07 07 | 07 07 |
| **0x00000018** | sw $1, 3($2) | 18 1c | 0c 00 | 09 03 | 09 07 |
| **0x0000001C** | and $6,$4,$5 | 1c 20 | 02 00 | 02 0b | 02 0b |
| **0x00000020** | move $7,$1 | 20 24 | 07 00 | 07 00 | 07 00 |
| **0x00000024** | lw $4,3($2) | 24 28 | 0c 07 | 09 03 | 09 02 |
| **0x00000028** | beq $4,$7,-6 | 28 14 | 00 00 | 07 07 | 07 07 |
| **0x0000002C** | addi $2,$3,1 | 2c 30 | 11 00 | 10 01 | 10 09 |

* + - 1. **实验心得**

这次的单周期*CPU*的设计，重新捡回了上学期的*Verilog*和*vivado*，虽说上学期用*Verilog*打过一个小小的*project*，但是今次再次打的时候，还是很卡壳，特别是这次的*CPU*设计，变量很多，所以模块间的关系要理清楚，不然很容易就会出现*bug*，下面说说这次实验中碰到的几个问题吧。

第一个问题出现在写完整个*CPU*后进行仿真的报错，然后由于*vivado*的错误提示真的是很不清楚，最后我在文件夹里面找到了仿真文件的错误信息，然后发现原来变量名在实例化的时候，与仿真文件的变量名不相同，就这样仿真成功了。

第二个问题，仿真成功后，我浏览了一下变量表，除了在仿真文件中被我初始化的*clock*和*reset*变量外，其他全是*X*，这时我也很蒙，不知道怎样解决，后来我就想*CPU*工作的流程，第一步要做的就是取指令，但是我发现*PCWre*信号一直都是*X*，但是理应是1才对，所以我就想是我的初始化出现了问题，我原来*PCWre*的初始化是在顶层模块中进行的，后来我就将它放到了控制模块里面，就这样这个问题就解决了，*PCWre*变成了1。

第三个问题又来了，*PCWre*等于1了，但是为什么指令还是*X*呢？然后我就查了一下指令存储器，发现也是全X，所以我就把关注点放到了指令存储器，后来发现原来是路径名称打错了，但是*vivado*也是没有报错，后来把路径修改完后，仿真的所有*bug*就全部修改完了，然后我就开始逐条指令验证是否正确。

验证指令的正确性，是一件很费时的工作，对于每条指令，你要检查读写寄存器地址是否正确，是否需要符号位拓展，*ALU*的输入输出是什么，各种控制信号的值，这个过程大概花了我1个小时，就在检查到最后的一条指令的时候，又发现了问题了，在负数的符号位拓展的时候，fffa被拓展成000ffffa，就觉得很奇怪，因为我的代码是让原16位前加4个f，即16个1，但是结果却只加了一个f，很懵逼，后来我就改成二进制的解法，16个1，又神奇般地可以了，所以至今不知道问题是什么。

最后的问题出现在了烧板子的时候，由于顶层模块实在太多变量了，后来把我就把一个分频的时钟给搞错了，但是*vivado*也没有报错，我觉得很神奇，因为我压根没有设置过那个变量，但是我在后面实例化的时候就用到了这个变量，所以理应报错才对。

这次的实验就先告一个段落了，五一回来的多周期*CPU*设计才是真正的大难题。