****

**《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 | | | | | |
| **专业（班级）** | **：** | 16计算机类4班 | | | | | |
| **学生姓名** | **：** | 陈泓仰 | | | | | |
| **学号** | **：** | 15303009 | | | | | |
| **时间** | **：** | 2017 | 年 | 11 | 月 | 18 | 日 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | **成绩** | **:** |  |
| **实验二** | **：** | **多周期CPU设计与实现** | | | |

* + - 1. **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;
  5. 掌握多周期CPU的实现方法。
     + 1. **实验内容**

1. 本次实验要求我们设计并实现一个多周期CPU，该CPU具有简单的算术逻辑运算（包括加法，减法，与运算，或运算，左移等），读写内存（lw，sw），分支指令（beq，bgtz，bne），跳转指令（j，jr），调用子程序指令（jal）以及停机指令。

指令格式如下：

1. **算术运算指令**
   1. add rd , rs, rt （说明：以助记符表示，是汇编指令；以代码表示，是机器指令）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 100000 | rs(5位) | rt(5位) | rd(5位) | reserved |

功能：rd←rs + rt。**reserved为预留部分，即未用，一般填“0”。**

* 1. addi rt , rs ,**immediate**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 101000 |  | rs(5位) | rt(5位) | **immediate**(16位) |

功能：rt←rs + (sign-extend)**immediate；immediate**符号扩展再参加“加”运算。

* 1. sub rd , rs , rt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 100010 | rs(5位) | rt(5位) | rd(5位) | reserved |

功能：rd←rs - rt

1. **逻辑运算指令**
   1. ori rt , rs ,**immediate**

|  |  |  |  |
| --- | --- | --- | --- |
| 100111 | rs(5位) | rt(5位) | **immediate**(16位) |

功能：rt←rs **|** (zero-extend)**immediate；immediate**做“0”扩展再参加“或”运算。

* 1. and rd , rs , rt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 100100 | rs(5位) | rt(5位) | rd(5位) | reserved |

功能：rd←rs & rt；逻辑与运算。

* 1. or rd , rs , rt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 100101 | rs(5位) | rt(5位) | rd(5位) | reserved |

功能：rd←rs **|** rt；逻辑或运算。

1. **移位指令**
   1. sll rd, rt,sa

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 101101 | 未用 | rt(5位) | rd(5位) | sa | reserved |

功能：rd<－rt<<(zero-extend)sa，左移sa位 ，(zero-extend)sa

1. **比较指令**
   1. slt rd, rs, rt 带符号数

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 101010 | rs(5位) | rt(5位) | rd(5位) | reserved |

功能：if (rs<rt) rd =1 else rd=0, 具体请看表2 ALU运算功能表，带符号

* 1. slti rt, rs,immediate 带符号

|  |  |  |  |
| --- | --- | --- | --- |
| 110010 | rs(5位) | rt(5位) | **immediate**(16位) |

功能：if (rs <(sign-extend)**immediate)** rt =1 else rt=0， 具体请看表2 ALU运算功能表，带符号

1. **存储器读/写指令**
   1. sw rt ,**immediate(**rs) 写存储器

|  |  |  |  |
| --- | --- | --- | --- |
| 101100 | rs(5位) | rt(5位) | **immediate**(16位) |

功能：memory[rs+ (sign-extend)**immediate**]←rt；**immediate**符号扩展再相加。即将rt寄存器的内容保存到rs寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。

* 1. lw rt , **immediate**(rs) 读存储器

|  |  |  |  |
| --- | --- | --- | --- |
| 100011 | rs(5位) | rt(5位) | **immediate**(16位) |

功能：rt ← memory[rs + (sign-extend)**immediate**]；**immediate**符号扩展再相加。

即读取rs寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中的数，然后保存到rt寄存器中。

1. **分支指令**
   1. beq rs,rt,**immediate**

|  |  |  |  |
| --- | --- | --- | --- |
| 000100 | rs(5位) | rt(5位) | **immediate**(16位) |

功能：if(rs=rt) pc←pc + 4 + (sign-extend)**immediate** <<2 else pc ←pc + 4

特别说明：**immediate是从PC+4地址开始和转移到的指令之间指令条数。immediate**符号扩展之后左移2位再相加。为什么要左移2位？由于跳转到的指令地址肯定是4的倍数（每条指令占4个字节），最低两位是“00”，因此将**immediate**放进指令码中的时候，是右移了2位的，也就是以上说的“指令之间指令条数”。

* 1. bne rs,rt,**immediate**

|  |  |  |  |
| --- | --- | --- | --- |
| 000101 | rs(5位) | rt(5位) | **immediate** |

功能：if(rs!=rt) pc←pc + 4 + (sign-extend)immediate <<2 else pc ←pc + 4

特别说明：与beq不同点是，不等时转移，相等时顺序执行。

* 1. bgtz rs,**immediate**

|  |  |  |  |
| --- | --- | --- | --- |
| 000111 | rs(5位) | 00000 | **immediate** |

功能：if(rs>0) pc←pc + 4 + (sign-extend)**immediate** <<2 else pc ←pc + 4

1. **跳转指令**
   1. j addr

|  |  |
| --- | --- |
| 000010 | addr[27..2] |

功能：pc <－{(pc+4)[31..28],addr[27..2],0,0}，无条件跳转。

说明：由于MIPS32的指令代码长度占4个字节，所以指令地址二进制数最低2位均为0，将指令地址放进指令代码中时，可省掉！这样，除了最高6位操作码外，还有26位可用于存放地址，事实上，可存放28位地址了，剩下最高4位由pc+4最高4位拼接上。

* 1. jr rs

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 111001 | rs(5位) | 未用 | 未用 | reserved |

功能：pc <－ rs，跳转。

1. **调用子程序指令**
   1. jal addr

|  |  |
| --- | --- |
| 111010 | addr[27..2] |

功能：调用子程序，pc <－ {(pc+4)[31:28],addr[27:2],0,0}；$31<－pc+4，返回地址设置；子程序返回，需用指令 jr $31。跳转地址的形成同 j addr 指令。

1. **停机指令**
   1. halt

|  |  |
| --- | --- |
| 111111 | 00000000000000000000000000(26位) |

功能：停机；不改变PC的值，PC保持不变。

**注意：这里的指令前六位op码与老师提供的不同。**

1. 最后通过下列测试程序段来验证指令的正确性：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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 | **101000** | **00000** | **00001** | **0000 0000 0000 1000** | **=** | 08010008 |
| **0x00000004** | ori $2,$0,2 | 100111 | 00000 | 00010 | 0000 0000 0000 0010 |  |  |
| **0x00000008** | or $3,**$2**,$1 | 100101 | 00010 | 00001 | 0001 1000 0000 0000 |  |  |
| **0x0000000C** | sub $4,**$3**,$1 | 100010 | 00011 | 00001 | 0010 0000 0000 0000 |  |  |
| **0x00000010** | and $5,**$4**,$2 | 100100 | 00100 | 00010 | 0010 1000 0000 0000 |  |  |
| **0x00000014** | **sll $5,$5,2** | 101101 | 00000 | 00101 | 0010 1000 1000 0000 |  |  |
| **0x00000018** | **beq $5,$1,-2(=,转14)** | 000100 | 00101 | 00001 | 1111 1111 1111 1110 |  |  |
| **0x0000001C** | **jal 0x0000048** | 010101 | 00000 | 00000 | 0000 0000 0001 0010 |  |  |
| **0x00000020** | **slt $8,$12,$1** | 101010 | 01100 | 00001 | 0100 0000 0000 0000 |  |  |
| **0x00000024** | **addi $14,$0,-2** | 101000 | 00000 | 01110 | 1111 1111 1111 1110 |  |  |
| **0x00000028** | **slt $9,$8,$14** | 101010 | 01000 | 01110 | 0100 1000 0000 0000 |  |  |
| **0x0000002C** | **slti $10,$9,2** | 110010 | 01001 | 01010 | 0000 0000 0000 0010 |  |  |
| **0x00000030** | **slti $11,$10,0** | 110010 | 01010 | 01011 | **0000 0000 0000 0000** |  |  |
| **0x00000034** | add $11,**$11**,$8 | 100000 | 01011 | 01000 | **0101 1000 0000 0000** |  |  |
| **0x00000038** | **bne $11,$2,-2 (≠,转34)** | 000101 | 01011 | 00010 | 1111 1111 1111 1110 |  |  |
| **0x0000003C** | addi $2,$2,-1 | 101000 | 00010 | 00010 | **1111 1111 1111 1111** |  |  |
| **0x00000040** | **bgtz $2,-2 (>0,转3C)** | 000111 | 00010 | 00000 | 1111 1111 1111 1110 |  |  |
| **0x00000044** | **j 0x0000054** | 000010 | 00000 | 00000 | **0000 0000 0001 0101** |  |  |
| **0x00000048** | sw $2,4($1) | 101100 | 00001 | 00010 | **0000 0000 0000 0100** |  |  |
| **0x0000004C** | lw $12,4($1) | 100011 | 00001 | 01100 | **0000 0000 0000 0100** |  |  |
| **0x00000050** | **jr $31** | 010100 | 01111 | 00000 | 0000 0000 0000 0000 |  |  |
| **0x00000054** | **halt** | 111111 | 00000 | 00000 | 0000000000000000 | **=** | FC000000 |

1. 通过Basys3板来运行设计的CPU，检验CPU的结果是否正确。
   * + 1. **实验原理**
2. 多周期CPU指的是将整个CPU的执行过程分成几个阶段，每个阶段用一个时钟去完成，然后开始下一条指令的执行，而每种指令执行时所用的时钟数不尽相同，这就是所谓的多周期CPU。
3. 在CPU处理指令时，一般需要经过以下几个步骤：
   1. 取指令（IF）：根据程序计数器PC中的指令地址，从存储器中取出一条指令，同时，PC根据指令字长（这里的指令都是32位）自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把转移地址送入PC。
   2. 指令译码（ID）：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。
   3. 指令执行(**EXE**)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。
   4. 存储器访问(**MEM**)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。
   5. 结果写回(**WB**)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

所以，在多周期CPU中，所有指令都会按照这五个阶段运行，一条指令的执行最长需要五个时钟周期才能完成。但是不同指令运行过程是不同的，有些指令不需要五个时钟周期来运行。对比单周期CPU，单周期CPU每条指令都会执行IF，ID，EXE，MEM，WB五个阶段，这样就会导致时间上的浪费。而多周期会根据指令的类型来决定，节约了时间。

指令运行处理过程图如下：

**取指令**

**IF**

**指令译码**

**ID**

**指令执行**

**EXE**

**存储器访问**

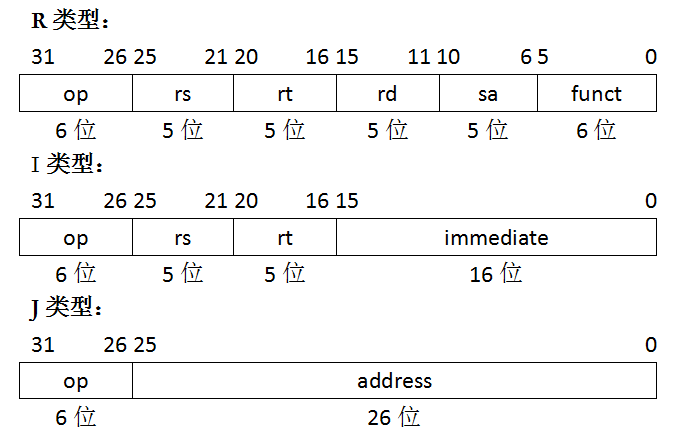
**MEM**

**结果写回**

**WB**

1. 在这次实验中，我的CPU指令格式与MIPS指令格式相同，都是将指令分成3类：R型，I型，J型。其中R型主要是执行算术逻辑运算指令以及访寸指令，I型执行分支指令，而J型指令执行跳转指令。

指令格式如图下所示：



其中，

**op：**为操作码；

**rs：**为第1个源操作数寄存器，寄存器地址（编号）是00000~11111，00~1F；

**rt：**为第2个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

**rd：**为目的操作数寄存器，寄存器地址（同上）；

**sa：**为位移量（shift amt），移位指令用于指定移多少位；

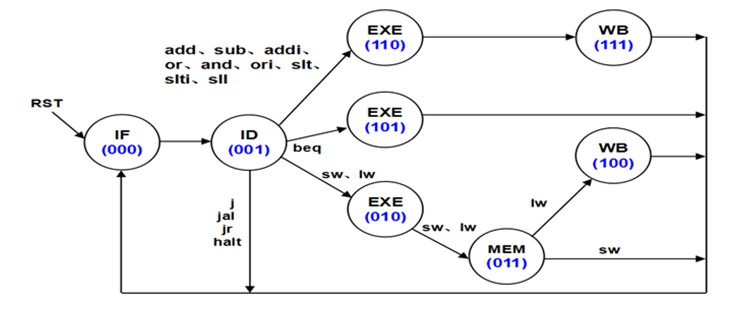
**funct：**为功能码，在寄存器类型指令中（R类型）用来指定指令的功能；

**immediate：**为16位立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载（Load）/数据保存（Store）指令的数据地址字节偏移量和分支指令中相对程序计数器（PC）的有符号偏移量；

**address：**为地址。

1. 最后将写好的CPU烧入basys板中，并验证是否正确。
2. 由于多周期CPU主要是将每条指令的五个阶段分开来实现，那么如何决定当前阶段以及实现当前阶段与下一阶段的切换就是一个值得考虑的问题。
3. 在这里，我们将指令的5个阶段根据指令的执行过程再细分为7个状态：
4. IF，每条指令都会执行的进行相同操作一个更改PC过程。
5. ID
6. EXE1，执行算数逻辑运算指令的计算。
7. EXE2，执行访存指令地址的运算。
8. EXE3，验证是否执行分支跳转指令。
9. MEM1，访存指令的访存阶段。
10. WB，将数据写会寄存器的阶段。

状态转移图：



状态的转移有的是无条件的，例如从IF状态转移到ID就是无条件的；有些是有条件的，例如EXE状态不止一个状态，到底转向哪个状态由该指令功能，即指令操作码决定。每个状态代表一个时钟周期。

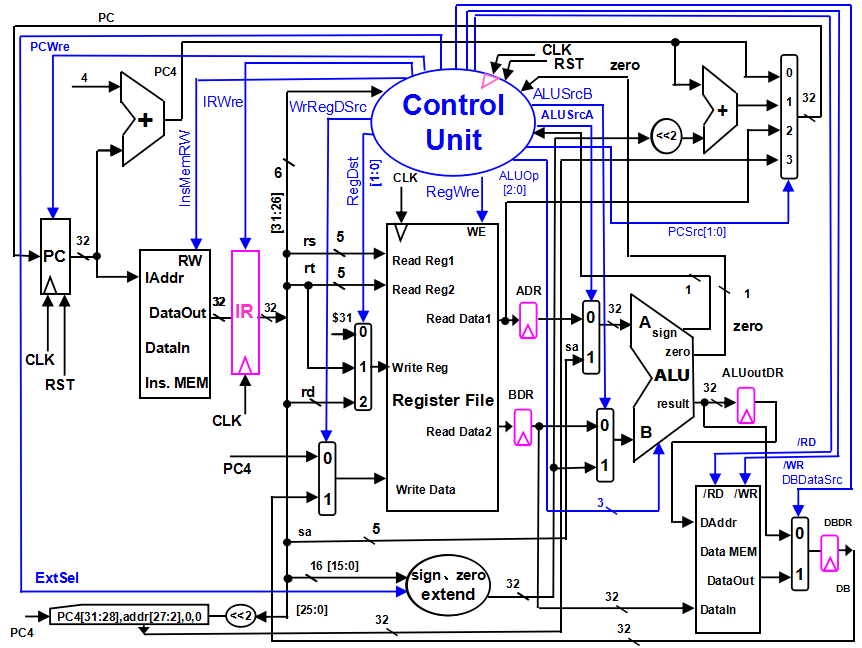
* + - 1. **实验器材**

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

* + - 1. **实验过程与结果**

1. 在设计多周期CPU前，我们必须清楚5种类型的指令的工作原理。

下面附上多周期CPU的数据通路和控制线路图：



1. R型指令：

首先在IF阶段，PC会改变当前指令，跳转到下一条指令。与之同时，指令存储器会根据PC地址的值得到相应的指令，然后在ID阶段将指令存入指令寄存器，并将当前PC地址所对应的指令进行译码。control单元读取指令的高六位，知道这是一个R型指令，所以此时会发出一些信号（线的粗细表示信号位数的大小）来使得寄存器，内存，ALU control进行工作。下面我们通过加法指令，来示范后面几个阶段指令的执行过程。首先控制单元会根据指令的ALU OP码，知道这是一个R型指令，并确定他是一个加法指令。所以ALU control会发出000的信号，指导逻辑计算单元ALU进行加法。接下来，我们看寄存器单元，它会读取指令的21-25，16-20位，11-15位读出俩个源寄存器和目的寄存器的编号（RegDst = 1），然后发出AluSrcA = 0，AluSrcB = 0,使寄存器读出的值先传到ADR，BDR俩个寄存器来进行保存，以确保后面阶段数据的稳定性。并在EXE阶段传到ALU中进行加法计算。由于此时/RD,/WR为1,所以不会对内存进行操作。接着，由于DBDataSrc = 0，所以ALU 将结果送给DBDR寄存器保存下来，最后在WB阶段时，将结果写回目的寄存器中，并将PCWre置为高电平，以确保下一条指令PC的改变。（这里会执行指令的IF，ID，EXE，WB阶段）

1. 访存指令：

这里，以lw指令为例：首先在IF阶段，PC会改变当前指令，跳转到下一条指令。与之同时，指令存储器会根据PC地址的值得到相应的指令，然后在ID阶段将指令存入指令寄存器（IR），并将当前PC地址所对应的指令进行译码。control单元读取指令的前六位，得知这是一个lw指令，由此会发出一系列指令来指引其他单元在下一阶段的工作。首先ALU control发出000信号来指引ALU 单元进行加法运算。然后，RegDst信号为0，指示寄存器单元的目的寄存器是由指令16-20位中得到。寄存器单元继续从指令的21-25位读rs源寄存器的编号，然后读取rs寄存器的值，并送到ADR寄存器存下。与之同时，因为AluSrc B= 1，所以ALU的另一个操作数，是从指令的低16位得到。指令的低16位会经过拓展单元扩展为32位然后送到BDR寄存器中存起来。接着在EXE阶段，ADR与BDR寄存器中的俩个值相加，得到一个内存地址，然后内存单元从这个地址读出值，送到DBDR寄存器中存起来。在这里（WR= 1， RD= 0，DB= 1）最后，在WB阶段时，将结果写回目的寄存器中，并将PCWre置为高电平，以确保下一条指令PC的改变。（lw会执行指令的IF，ID，EXE，MEM，WB阶段；而sw则会执行指令的IF，ID，EXE，MEM阶段）

1. 分支指令：

我们知道，下一条指令的可以为PC + 4，可以为beq的指令所指引，也可以为J型指令结果。这里是beq型指令，control单元先读取机器代码的前6位，知道这是一个beq型指令，然后ALUcontrol发出信号001指引ALU进行减法运算。然后寄存器单元会从机器代码的21-25，16-20，读取rs，rt寄存器的编号，并读取其中的值，送到ADR，BDR寄存器中，并在EXE阶段中进行减法。若结果为0，则发出Sign信号给AluControl，使得PCWre发出信号1，同时改变PCSrs的值使下一条指令等于原PC + 4 + Immediate \*4。（分支指令执行指令的IF，ID，EXE阶段）

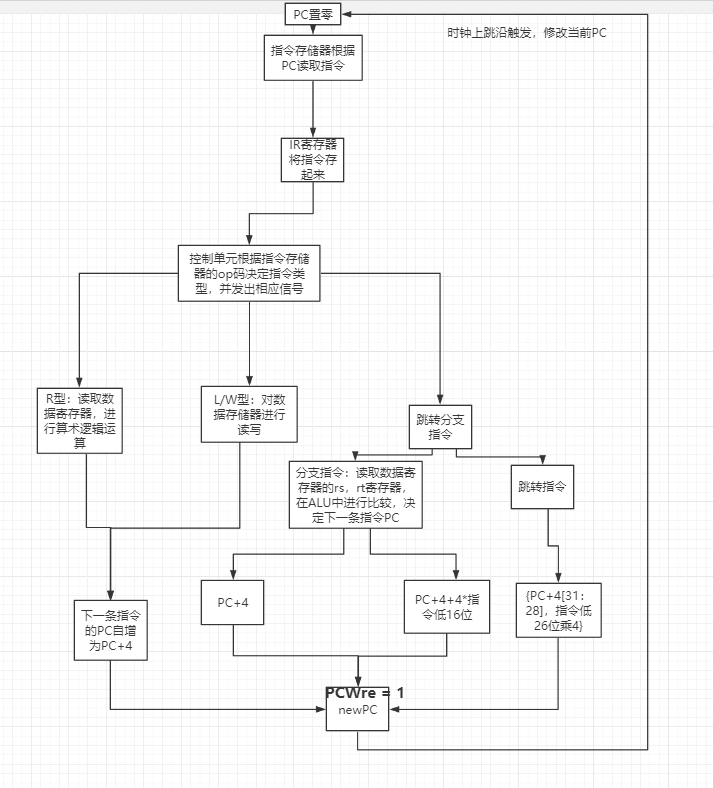
1. 跳转指令：

下一条指令的可以为PC + 4，可以为beq的指令所指引，可以为J型指令结果。这里是J型指令，control单元先读取机器代码的前6位，知道这是一个J型指令，所以会发出一些信号使得内存单元，寄存器单元，ALU control单元无效，所以会将指令的后26位进行俩位左移，PC+4的前四位作为下一条PC的前四位。并置PCWre为高电平。（跳转指令执行指令的IF，ID阶段）

1. 调用子程序指令：

首先在IF阶段，PC会改变当前指令，跳转到下一条指令。与之同时，指令存储器会根据PC地址的值得到相应的指令，然后在ID阶段将指令存入指令寄存器（IR），并将当前PC地址所对应的指令进行译码。Control单元识别到他是一条jal指令，则会将PC+4指令存到第31号寄存器中（WrRegDSrc = 0，RegDst = 2），然后将指令的后26位进行俩位左移，PC + 4的前四位作为下一条PC的前四位，并置PCWre为高电平。（调用子程序指令会执行指令的IF，ID阶段）

1. 指令的执行流程：



1. 这样，我们可以决定多周期CPU的模块实现如下：

|  |  |  |
| --- | --- | --- |
| 部件 | 端口 | 功能 |
| PC | Input:CLK,InputPC[31:0],Reset  Output:OutputPC[31:0] | 当Reset为0时，置PC为0，然后Reset为1且在时钟CLK上调沿触发时，修改PC的值，发出下一条PC的值。 |
| Choose | Input:PCWre,Inputpc[31:0], PCSrc[1:0],AJshift[25:0], Immediate [31:0],  Output :Outputpc[31:0] | 会根据PCWre的值来决定是应该采取何种PC增加方式。 |
| Instruction Memory | Input:IAddress[31:0], InsMemRW,  0utput:op[5:0],rs[4:0], rt[4:0],rd[4:0],immediate[15:0], AJshift[25:0], sa[4:0] | 将PC处所存着的指令读取出来，并分解为op，rs，rt等几个部分输出。 |
| Register File | Input:CLK, Reset, RegWre, ReadReg1[4:0], ReadReg2[4:0],  WriteReg[4:0], WriteData[31:0],  Output:ReadData1[31:0], ReadData2[31:0] | 将rs，rt的值对应的寄存器中的值读取出来，然后在时钟上跳沿将传入WriteData中的值写入rd对应的寄存器中。 |
| Data Memory | Input:CLK, DataAddress[31:0], Datain[31:0], RD, WR,  Output:DataOut[31:0] | 当RD=1时，读取内存，当WR = 1时，将数据在时钟下跳沿写入内存中。指令存储器和数据存储器存储单元宽度一律使用8位，即一个字节的存储单位。不能使用32位作为存储器存储单元宽度。 |
| MuxForReadData1 | Input:AluSrcA,Data[31:0], Sa[4:0],  Output:Result[31:0] | 当AluSrcA = 0，输出rs寄存器所对应的值，否则输出指令中的sa偏移量 |
| MuxForReadData2 | Input:AluSrcB,Data[31:0], ExtendResult[31:0],  Output:Result[31:0] | 当AluSrcB = 0，输出rt寄存器所对应的值，否则输出指令中后16位扩展出来的32位数据。 |
| MuxForWriteReg | Input:RegDst, rt[4:0], rd[4:0],  Output: Result[4:0] | 当RegDst= 0，会将结果写入rt对应的寄存器中，RegDst = 1写入rd所对应的寄存器中，RegDst = 2写入31号寄存器。 |
| MuxForWirteData | Input:DBDataSrc, AluData[31:0], MemData[31:0],  Output: Result[31:0] | 当DB = 0，输出ALU的结果，否则输出内存所读出来的值 |
| SignZeroExtend | Input: InData[15:0], ExtSel,  Output: OutData[31:0] | 当ExtSel = 0为零扩展，否则为符号扩展。 |
| ALU | Input:A[31:0],B[31:0],AluOp[2:0],  Output:Zero, Sign, Result[31:0] | 根据AluOP的值来决定ALU所进行的运算。 |
| ControlUnit | Input :op[5:0], Zero, Sign,  Output:ExtSel,PCWre, InsMemRW, RegDst, RegWre,  AluOp[2:0], AluSrcA, AluSrcB,  RD, WR, DBDataSrc, | 根据OP码决定这是指令类型，并发出相关的信号，指引其他单元的运算。 |
| IR | Input：CLK，Instruction  Output：Instruction | 将都出来的指令存进寄存器中，保证指令的稳定性 |
| ADR（BDR） | Input：CLK，DataA(DataB)  Output:Data1(Data2) | 将从数据寄存器出来的数据存入寄存器中，保证后面状态可以读到这些数据，保证数据性 |
| DBDR | Input：CLK，Data  Output：Data | 将从内存中读出的数据存入寄存器中，保证后面状态可以读到这些数据 |
| AluOutDR | Input：CLK，AluOut  Output：Data | 将ALU模块的数据输出存入寄存器中，同样保证后面状态可以读到这些数据。 |
| MuxForJal | Input：WrRegDSrc，PC4，jal指令存下来的地址 | 当WrRegDSrc = 0，写入寄存器组寄存器的数据来自pc+4(pc4）。  否则，写入寄存器组寄存器的数据来自ALU运算结果或存储器读出的数据。 |

然后我们可以确定CPU中的端口及各个端口信号的作用，通过该表我们可以定义每条指令中，Control Unit 会发出什么信号：

|  |  |  |
| --- | --- | --- |
| **控制信号名** | **状态“0”** | **状态“1”** |
| **RST** | 对于PC，初始化PC为程序首地址 | 对于PC，PC接收下一条指令地址 |
| **PCWre** | PC不更改，相关指令：halt，另外，除‘000’状态之外，其余状态慎改PC的值。 | PC更改，相关指令：除指令halt外，另外，在‘000’状态时，修改PC的值合适。 |
| **ALUSrcA** | 来自寄存器堆data1输出，相关指令：add、sub、addi、or、and、ori、beq、bne、bgtz、slt、slti、sw、lw | 来自移位数sa，同时，进行(zero-extend)sa，即 {{27{0}},sa}，相关指令：sll |
| **ALUSrcB** | 来自寄存器堆data2输出，相关指令：add、sub、or、and、beq、bne、bgtz、slt、sll | 来自sign或zero扩展的立即数，相关指令：addi、ori、slti、lw、sw |
| **DBDataSrc** | 来自ALU运算结果的输出，相关指令：add、sub、addi、or、and、ori、slt、slti、sll | 来自数据存储器（Data MEM）的输出，相关指令：lw |
| **RegWre** | 无写寄存器组寄存器，相关指令：  beq、bne、bgtz、j、sw、jr、halt | 寄存器组寄存器写使能，相关指令：add、sub、addi、or、and、ori、slt、slti、sll、lw、jal |
| **WrRegDSrc** | 写入寄存器组寄存器的数据来自pc+4(pc4），相关指令：jal，写$31 | 写入寄存器组寄存器的数据来自ALU运算结果或存储器读出的数据，相关指令：add、addi、sub、or、and、ori、slt、slti、sll、lw |
| **InsMemRW** | 写指令存储器 | 读指令存储器(Ins. Data) |
| **/RD** | 读数据存储器，相关指令：lw | 存储器输出高阻态 |
| **/WR** | 写数据存储器，相关指令：sw | 无操作 |
| **IRWre** | IR(指令寄存器)不更改 | IR寄存器写使能。向指令存储器发出读指令代码后，这个信号也接着发出，在时钟上升沿，IR接收从指令存储器送来的指令代码。与每条指令都相关。 |
| **ExtSel** | (zero-extend)**immediate**，相关指令：ori； | (sign-extend)**immediate**，相关指令：addi、lw、sw、beq、bne、bgtz； |
|  | | |
| **PCSrc[1..0]** | 00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、  slt、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bgtz(sign=1，  或zero=1)；  01：pc<－pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、  bne(zero=0)、bgtz(sign=0，且zero=0)；  10：pc<－{pc[31:28],addr[27:2],0,0}，相关指令：j、jal；  11：pc<－rs，相关指令：jr； | |
| **RegDst[1..0]** | 写寄存器组寄存器的地址，来自：  00：rt字段，相关指令：addi、ori、slti、lw；  01：rd字段，相关指令：add、sub、or、and、slt、sll；  10：0x1F($31)，相关指令：jal，用于保存返回地址（$31<-pc+4）；  11：未用； | |
| **ALUOp[2..0]** | ALU 8种运算功能选择(000-111)，看功能表 | |

最后，我们定义了ALU中的运算功能：

|  |  |  |
| --- | --- | --- |
| **ALUOp[2..0]** | **功能** | **描述** |
| **000** | **Y = A + B** | **加** |
| **001** | **Y = A – B** | **减** |
| **010** | **Y = B<<A** | **B左移A位** |
| **011** | **Y = A ∨ B** | **或** |
| **100** | **Y = A ∧ B** | **与** |
| **101** | **Y=（A<B）?1: 0** | **比较A与B**  **不带符号** |
| **110** | **if (A<=B &&(A[31] == B[31] ))**  **Y = 1;**  **else if ( A[31] && !B[31) Y = 1;**  **else Y = 0;** | **比较A与B**  **带符号** |
| **111** | **Y = A ⊕ B** | **异或** |

1. 我们对比单周期CPU，可以发现多周期CPU与单周期CPU的实现类似，但是由于多周期CPU中一条指令是分成多个时钟周期来实现的，因此，我们需要一些寄存器来帮助我们存下指令或者数据，以保证后面状态可以读到这些指令或者数据。IR，ADR，BDR，DBDR，AluOutDR就是为了实现这一目的。此外，我们需要考虑到多周期CPU实现上的时序问题。比如在更改PC时，PCWre应该在下一条指令PC来之前置为高电平等。
2. 下面附上相关模块的实现代码与分析：
3. **PC模块（时钟下跳沿触发改变PC的值）：**

always @(negedge CLK or negedge Reset)

begin

if (Reset == 0) begin

Outputpc <= 0;//当Reset置位为0时，下一条PC的输出为0.

end

else begin

if(PCWre == 1)begin

Outputpc = Inputpc; //当时钟上调沿触发时，PC输出值改变

end

end

end

1. **PC选择模块：**

if (PCSrc == 2'b00)

Outputpc <= Inputpc + 4; //当PCSrc=0时，下一条指令为当前指令加4

else if(PCSrc == 2'b01)

Outputpc <= Inputpc + 4 + Immediate \* 4; // 当PCSrc=1，分支指令跳转

else if(PCSrc == 2'b10) begin

Outputpc <= {4'b0000,AJshift,2'b00};//PCSrc = 2，跳转指令跳转

end

else if(PCSrc == 2'b11) begin

Outputpc <= rsdata;//PCSrc = 3，将rs寄存器中的PC值读出来，作为下一条指令PC

1. **InstructionMemory模块：**

reg [7:0] rom [99:0]; // 存储器定义必须用reg类型，存储器存储单元8位长度，共100个存储单元

initial begin // 加载数据到存储器rom。注意：必须使用绝对路径，如：E:/Xlinx/VivadoProject/ROM/（自己定）

$readmemb ("C:/rom\_data.txt", rom); // 数据文件rom\_data（.coe或.txt）。未指定，就从0地址开始存放。

end

always @( InsMemRW or IAddress ) begin

if (InsMemRW == 1) //为0，读存储器。大端数据存储模式

InsDataOut[31:24] = rom[IAddress];

InsDataOut[23:16] = rom[IAddress + 1];

InsDataOut[15:8] = rom[IAddress + 2];

InsDataOut[7:0] = rom[IAddress + 3];

end

1. **IR模块（时钟上跳沿触发，也就是说IF改变PC，ID阶段改变指令寄存器的值，并进行译码。）**

always@(posedge CLK) begin

if(IRWre == 1) begin

op = InsData[31:26];

rs = InsData[25:21];

rt = InsData[20:16];

rd = InsData[15:11];

immediate = InsData[15:0];

sa = InsData[10:6];

AJshift = InsData[25:0];

end

end

1. **RegisterFile模块：**

assign ReadData1 = regFile[ReadReg1]; //根据寄存器的值读值。

assign ReadData2 = regFile[ReadReg2];

initial begin //初始化寄存器的值。

for (i = 0 ; i < 32; i = i+1)

regFile[i] = 0;

end

always@(negedge CLK) begin//在时钟下跳沿触发时，修改rd或者rt对应的寄存器的值

if(RegWre == 1 && WriteReg != 0) begin

regFile[WriteReg] <= WriteData;

end

end

1. **Data Memory模块：**

reg [7:0] ram[60:0]; //reg ram[60:0];

assign DataOut[31:24] = (RD == 0)? ram[DataAddress] : 8'bz;//当RD = 1时，读内存

assign DataOut[23:16] = (RD == 0)? ram[DataAddress + 1] : 8'bz;

assign DataOut[15:8] = (RD == 0)?ram[DataAddress + 2] : 8'bz;

assign DataOut[7:0] = (RD == 0)? ram[DataAddress + 3] : 8'bz;

always@(\*)begin

if(WR == 0) begin

ram[DataAddress] <= Datain[31:24];

ram[DataAddress + 1] <= Datain[23:16];

ram[DataAddress + 2] <= Datain[15:8];

ram[DataAddress + 3] <= Datain[7:0];

end

end

1. **ALU模块：**

reg [31:0] Result;

reg Sign,Zero;//ZERO信号在决定下一条PC的值的时候是很用作用的。

always@(A or B or AluOp)begin//将两个复用器选择出来的结果在ALU中进行计算。

case(AluOp)//通过control unit发出的不同的AluOp码来执行不同的运算

3'b000:begin

Result = A + B;

Sign = Result[31];

Zero = (Result == 0)? 1 : 0;

end

按照如上指令与ALU符号表编写代码，实现不同的运算功能。

endcase

end

1. **D触发器模块：（时钟上跳沿改变当前状态）**

always@(posedge CLK)begin

if(RST == 0) NowStage = 3'b000;

else NowStage = NextStage;

end

1. **决定下一状态的模块（根据每条指令的功能来决定）：**

always@(opcode or NowStage) begin

//if -> id

if(NowStage == 3'b000) NextStage = 3'b001;

//id

if(NowStage == 3'b001)begin

//id -> if jump

if(opcode == 6'b000010 || opcode == 6'b010101 || opcode == 6'b010100 || opcode == 6'b111111)

NextStage = 3'b000;

//id -> alu calculate

else if(opcode == 6'b100000 ||opcode == 6'b100010 ||opcode == 6'b100100 ||opcode == 6'b100101 || opcode == 6'b101000 ||opcode == 6'b100111 || opcode == 6'b101101 ||opcode == 6'b101010 || opcode == 6'b110010)

NextStage = 3'b110;

//id -> lw,sw

else if(opcode == 6'b100011 || opcode == 6'b101100)

NextStage = 3'b010;

//id -> beq

else if(opcode == 6'b000100 || opcode == 6'b000101 ||opcode == 6'b000111)

NextStage = 3'b101;

end

//exe -> alucal -> wb

if(NowStage == 3'b110)begin

NextStage = 3'b111;

end

//exe -> beq -> nextPC

if(NowStage == 3'b101)begin

NextStage = 3'b000;

end

//exe -> lw,sw -> mem

if(NowStage == 3'b010)begin

NextStage = 3'b011;

end

//mem lw sw

if(NowStage == 3'b011)begin

//lw:mem -> Wb

if(opcode == 6'b100011)

NextStage = 3'b100;

//sw:mem -> nextPC

else if(opcode == 6'b101100)

NextStage = 3'b000;

end

if(NowStage == 3'b100) begin

NextStage = 3'b000;

end

if(NowStage == 3'b111)begin

NextStage = 3'b000;

end

end

1. **ControlUnit模块：**

**用stage和Zero信号触发，当stage改变时control信号改变，当ZERO值变时，控制信号也改变。**

always@(stage or Zero)begin

//if

if(stage == 3'b000) begin

if(op == 6'b111111)begin

InsMemRW = 1;

PCWre = 0;

RegWre = 0;

RD = 1;

WR = 1;

IR = 0;

WrRegDSrc = 1;

PCSrc = 0;

end

//j

else if(op == 6'b000010 )begin

PCWre = 0;

PCSrc = 2'b00;

InsMemRW = 1;

RegDst = 2'b01;

RegWre = 0;

IR = 1;

WrRegDSrc = 1;

end

//jal

else if(op == 6'b010101 )begin

PCWre = 0;

PCSrc = 2'b00;

InsMemRW = 1;

RegDst = 2'b10;

RegWre = 0;

IR = 1;

WrRegDSrc = 0;

end

//jr

else if(op == 6'b010100 )begin

PCWre = 0;

PCSrc = 2'b00;

InsMemRW = 1;

RegDst = 2'b01;

RegWre = 0;

IR = 1;

WrRegDSrc = 1;

end

else begin

InsMemRW = 1;

PCWre = 0;

RegWre = 0;

RD = 1;

WR = 1;

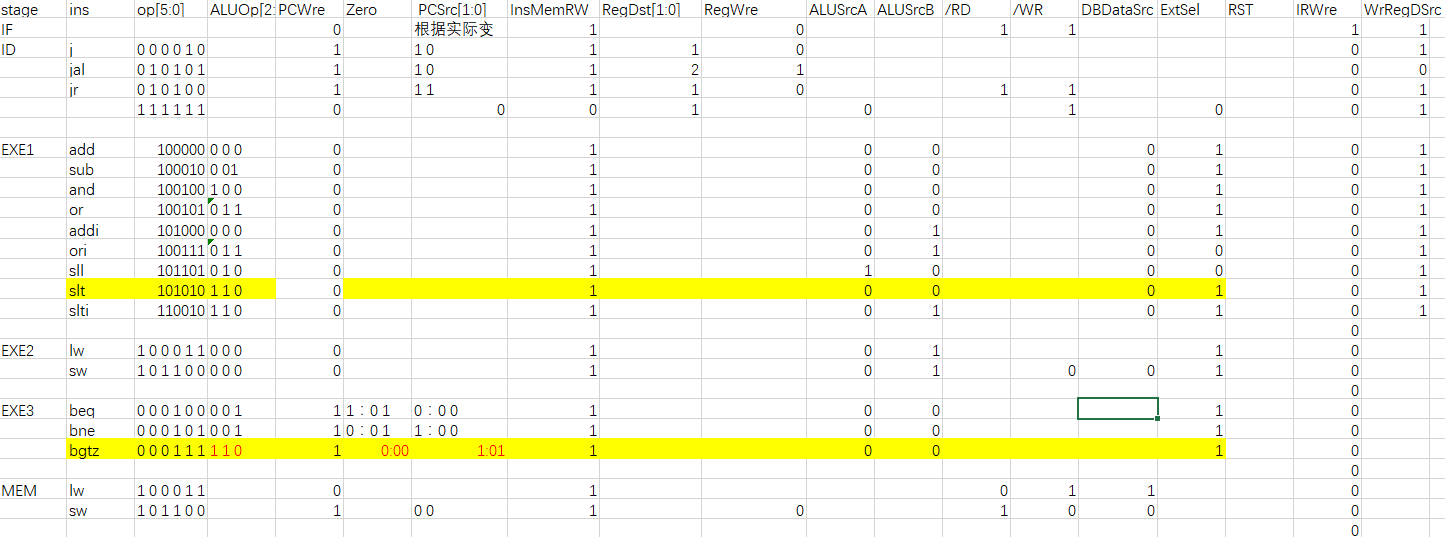
IR = 1;

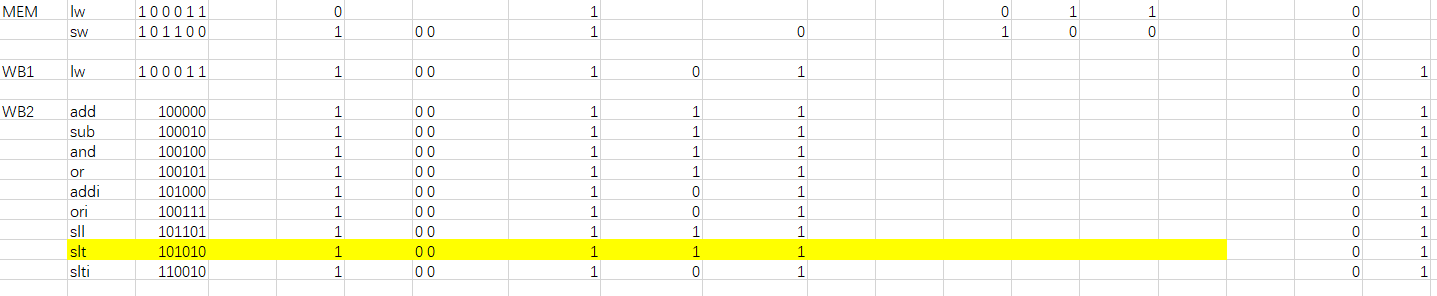
WrRegDSrc = 1;

end

end

只列举IF阶段控制信号的分配。其他阶段控制信号的值如下表所示：





1. **多路复用器（实现都差不多，下面以写寄存器的复用器为例）：**

always @ (rt or rd or RegDst) begin

case(RegDst)

2'b00: Result = rt;

2'b01: Result = rd;

2'b10: Result = 5'b11111;

endcase

end

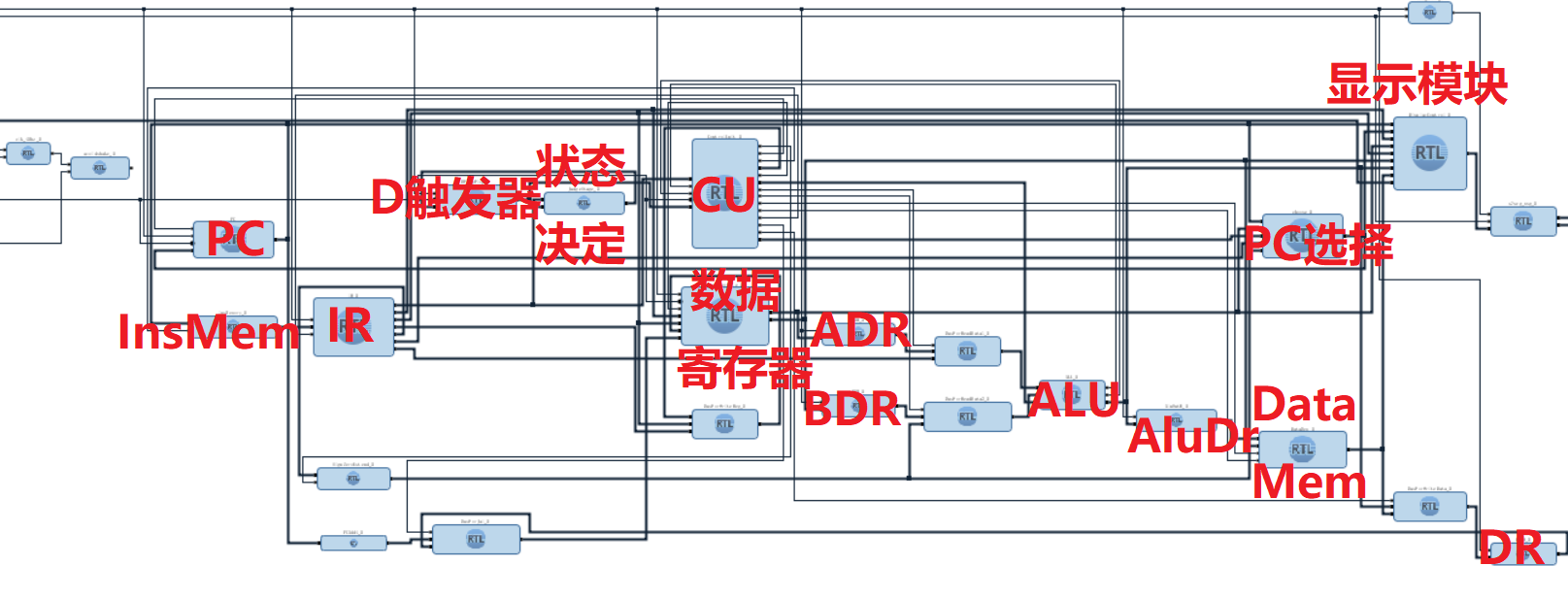
1. **数据寄存器（实现类似，以ADR为例，注意都是下跳沿写入）：**

always@(negedge CLK) begin

AOutput = AInput;

End

1. 顶层模块图：



1. 最后通过仿真模块文件，来实例化CPU，得出CPU波形，并**进行结果的验证**：

我们先将CPU运行的过程计算出来：

|  |  |  |
| --- | --- | --- |
| 地址 | 汇编程序 | 指令结果 |
| 0x00000000 | addi $1,$0,8 | $1 = 8 |
| 0x00000004 | ori $2,$0,2 | $2 = 2 |
| 0x00000008 | or $3,$2,$1 | $3 = 10 |
| 0x0000000C | sub $4,$3,$1 | $4 = 2 |
| 0x00000010 | and $5,$4,$2 | $5 = 2 |
| 0x00000014 | sll $5,$5,2 | $5 = 8 |
| 0x00000018 | beq $5,$1,-2(=,转14) | 跳0X14 |
| 0x00000014 | sll $5,$5,2 | $5 = 32 |
| 0x00000018 | beq $5,$1,-2(=,转14) | $5不等于$1,不跳 |
| 0x0000001C | jal 0x0000048 | 跳到0X48，同时将0X20存到$31 |
| 0x00000048 | sw $2,4($1) | 将$2 = 2存到内存[12] |
| 0x0000004C | lw $12,4($1) | 将内存[12] = 2取出存到$12,$12 = 2 |
| 0x00000050 | jr $31 | 跳到31号寄存器所存的PC值处 |
| 0x00000020 | slt $8,$12,$1 | $8 = 1 |
| 0x00000024 | addi $14,$0,-2 | $14 = -2 |
| 0x00000028 | slt $9,$8,$14 | $9 = 0 |
| 0x0000002C | slti $10,$9,2 | $10 = 1 |
| 0x00000030 | slti $11,$10,0 | $11 = 0 |
| 0x00000034 | add $11,$11,$8 | $11 = 1 |
| 0x00000038 | bne $11,$2,-2 (≠,转34) | 因为$11 不等于 $2,所以跳0X34 |
| 0x00000034 | add $11,$11,$8 | $11 = 2 |
| 0x00000038 | bne $11,$2,-2 (≠,转34) | 因为$11=$2,所以不跳0X34 |
| 0x0000003C | addi $2,$2,-1 | $2 = 1 |
| 0x00000040 | bgtz $2,-2 (>0,转3C) | $2>0,跳0x3c |
| 0x0000003C | addi $2,$2,-1 | $2 = 0 |
| 0x00000040 | bgtz $2,-2 (>0,转3C) | $2不大于0,所以不跳0x3c |
| 0x00000044 | j 0x0000054 | 跳到0x54 |
| 0x00000054 | halt | 停机 |

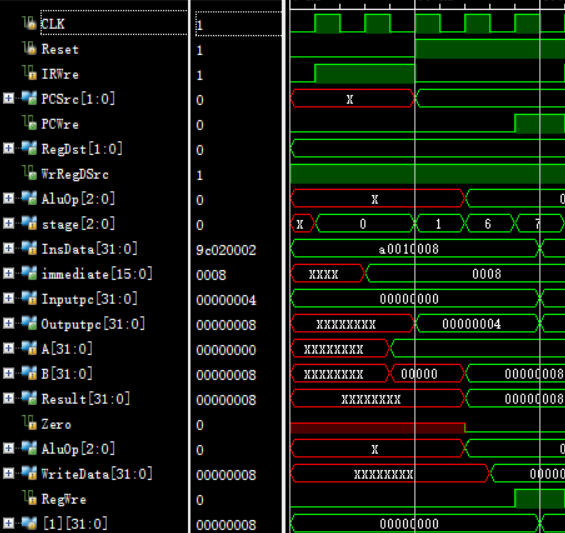
然后运行simulation文件，得到仿真波形，首先先对照PC的执行过程：

IF = 0，ID = 1，EXE = 2（访存指令），MEM = 3（sw），WB = 4(sw,lw),EXE = 5(分支指令)，EXE = 6（ALU），WB = 7（ALU）



|  |  |  |
| --- | --- | --- |
| 0x00000000 | addi $1,$0,8 | $1 = 8 |

波形：



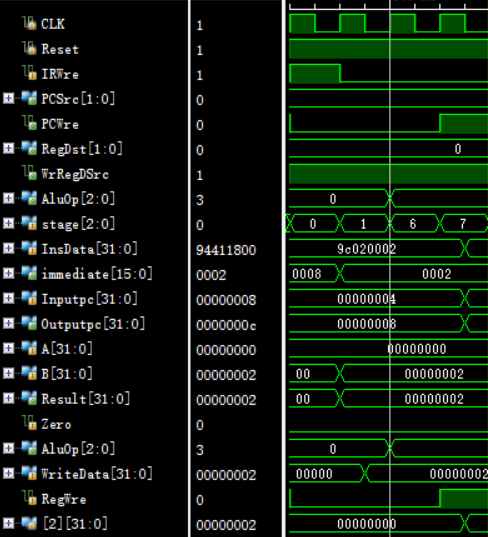
执行效果：把$rs = $0 = 0 和立即数8相加，结果送$rt = $ 1 = 8。

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  立即数 = 8，rs = 0，rt = 1 |
| 6 | 进行计算，同时将ALU输出结果存入AluOutDR中。 |
| 7 | 将RegWre置为1，并慢半个周期写入寄存器。同时置PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x00000004 | ori $2,$0,2 | $2 = 2 |

波形：



执行效果：把$rs = $0 = 0 和立即数2相或，结果送$rt = $ 2 = 2。

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  立即数= 2，rs = 2，rt = 2 |
| 6 | 进行计算，同时将ALU输出结果存入AluOutDR中。 |
| 7 | 将RegWre置为1，并慢半个周期写入寄存器。同时置PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x00000008 | or $3,$2,$1 | $3 = 10 |

波形：



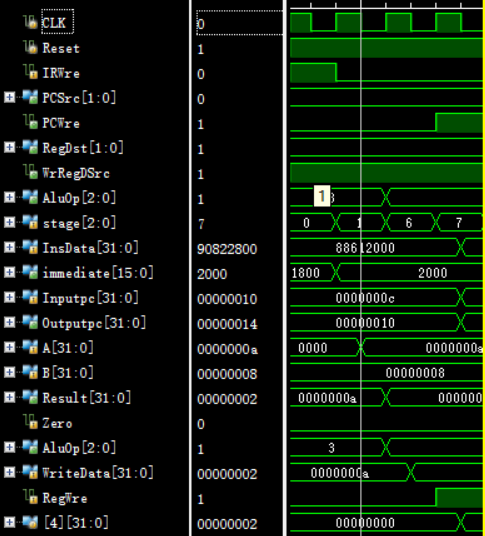
执行效果：将2号寄存器与1号寄存器相或，得到的值赋给3号寄存器，$3 = 0AH

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 2，rt = 1，rd = 3 |
| 6 | 进行计算，同时将ALU输出结果存入AluOutDR中。 |
| 7 | 将RegWre置为1，并慢半个周期写入寄存器。同时置PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x0000000C | sub $4,$3,$1 | $4 = 2 |

波形：



执行效果：将3号寄存器与1号寄存器相减，得到的值赋给4号寄存器，$4 = 2

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 3，rt = 1，rd = 4 |
| 6 | 进行计算，同时将ALU输出结果存入AluOutDR中。 |
| 7 | 将RegWre置为1，并慢半个周期写入寄存器。同时置PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x00000010 | and $5,$4,$2 | $5 = 2 |

波形：



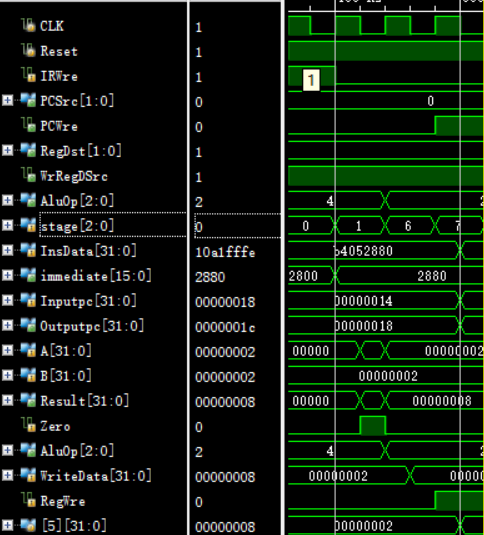
执行效果：将4号寄存器与2号寄存器相与，得到的值赋给5号寄存器，$5 = 2

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 4，rt = 2，rd = 5 |
| 6 | 进行计算，同时将ALU输出结果存入AluOutDR中。 |
| 7 | 将RegWre置为1，并慢半个周期写入寄存器。同时置PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x00000014 | sll $5,$5,2 | $5 = 8 |

波形：



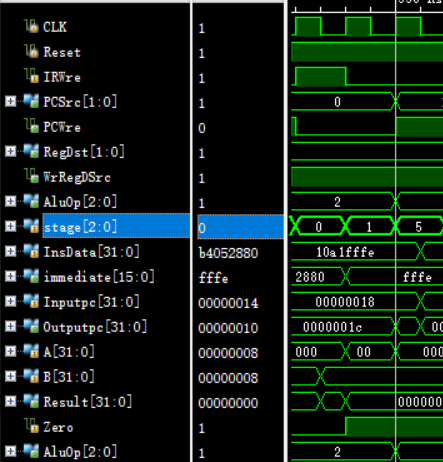
执行效果：将5号寄存器中的值左移俩位，得到的值赋给5号寄存器，$5 = 8

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 5，sa = 2，rt = 5 |
| 6 | 进行计算，同时将ALU输出结果存入AluOutDR中。 |
| 7 | 将RegWre置为1，并慢半个周期写入寄存器。同时置PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x00000018 | beq $5,$1,-2(=,转14) | 跳0X14 |

波形：



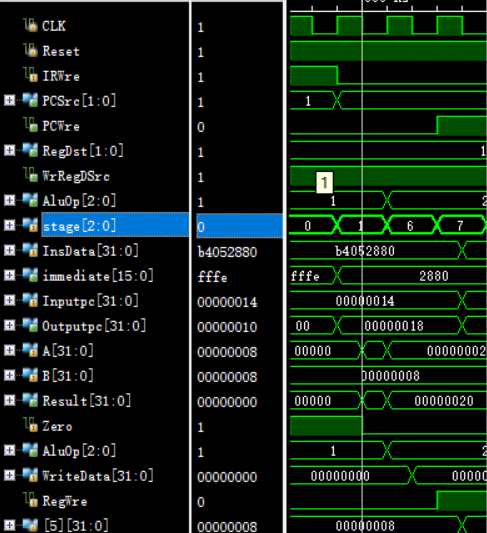
执行效果：将5号寄存器与1号寄存器相比较，若相等则跳到0X14。

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 5，rt = 1，立即数 = 0FFFEH |
| 5 | 置PCWre = 1,并置PCSrc = 1（分支跳转）。 |



|  |  |  |
| --- | --- | --- |
| 0x00000014 | sll $5,$5,2 | $5 = 32 |

波形：



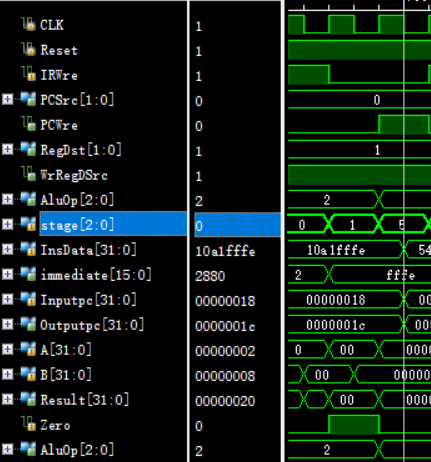
执行效果：将5号寄存器中的值左移俩位，得到的值赋给5号寄存器，$5 = 32

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 5，sa = 2，rt = 5 |
| 6 | 进行计算，同时将ALU输出结果存入AluOutDR中。 |
| 7 | 将RegWre置为1，并慢半个周期写入寄存器。同时置PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x00000018 | beq $5,$1,-2(=,转14) | $5不等于$1,不跳 |

波形：



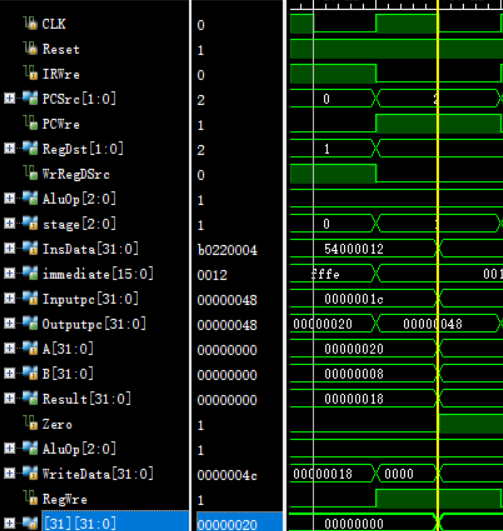
执行效果：将5号寄存器与1号寄存器相比较，若相等则跳到0X14。

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 5，rt = 1，立即数 = 0FFFEH |
| 5 | 置PCWre = 1,并置PCSrc = 0。 |



|  |  |  |
| --- | --- | --- |
| 0x0000001C | jal 0x0000048 | 跳到0X48，同时将0X20存到$31 |

波形：



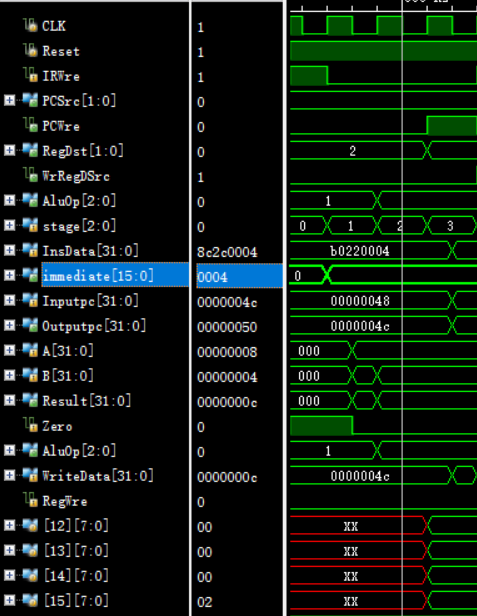
执行效果：跳到0X48，同时将0X20存入31号寄存器。

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。RegDst = 2，RegWre，并在下半个周期写入$31中。  PCSrc = 2，PC跳转指令。PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x00000048 | sw $2,4($1) | 将$2 = 2存到内存[12] |

波形：



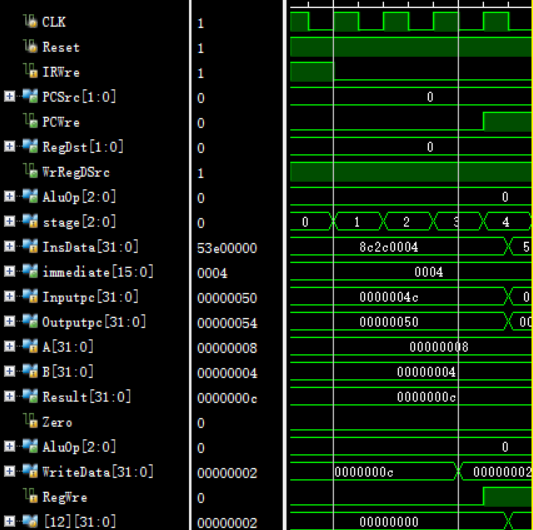
执行结果：将$2 = 2存到[$1 + 4]=[12]中，所以[12,13,14,15] = 00 00 00 02。

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR。WD 置为有效电平。  rs = 1，立即数 = 1，rt = 2 |
| 2 | 计算出目的内存地址。数据写入内存中。并置PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x0000004C | lw $12,4($1) | 将内存[12] = 2取出存到$12,$12 = 2 |

波形：



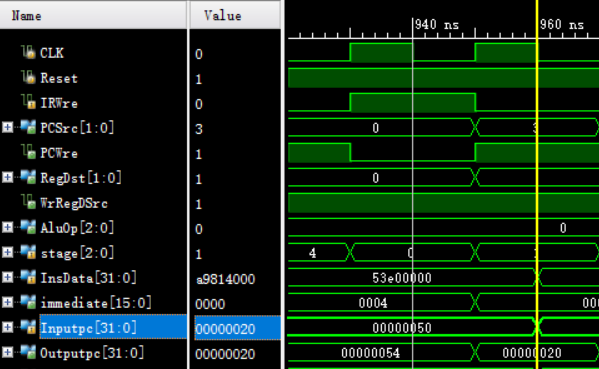
执行效果：将[12,13,14,15]存入$12,$12 = 2。

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 1，立即数 = 1，rt = 2 |
| 2 | 计算出目的寄存器地址，将数据存入AluOutDR中。 |
| 3 | RD置为有效的，访问内存，下半周期存入DBDR中。 |
| 4 | RegDst = 1，RegWre = 1。存入目的寄存器中。同时PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x00000050 | jr $31 | 跳到31号寄存器所存的PC值处 |

波形：



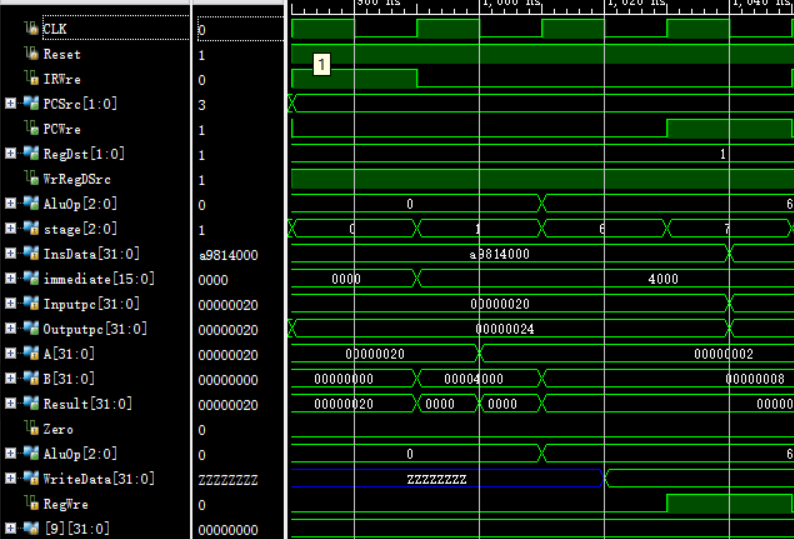
执行效果：跳到rs所对应的寄存器中的地址。

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。PCWre = 1。PCSrc = 3,跳到rs所对应的寄存器中。 |



|  |  |  |
| --- | --- | --- |
| 0x00000020 | slt $8,$12,$1 | $8 = 1 |

波形：



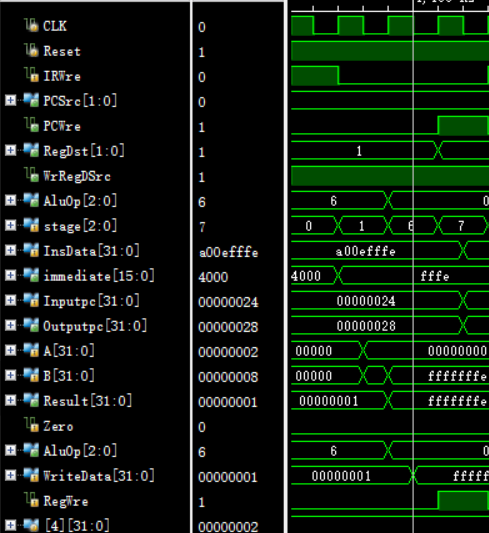
执行过程：将12号寄存器中的值与1号寄存器中的值比较，若$12 < $1,则赋1给8号寄存器，否则赋0。$8 = 0

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为无效电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 12，rt = 1，rd = 8 |
| 6 | 进行计算，同时将ALU输出结果存入AluOutDR中。 |
| 7 | 将RegWre置为1，并慢半个周期写入寄存器。同时置PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x00000024 | addi $14,$0,-2 | $14 = -2 |

波形：



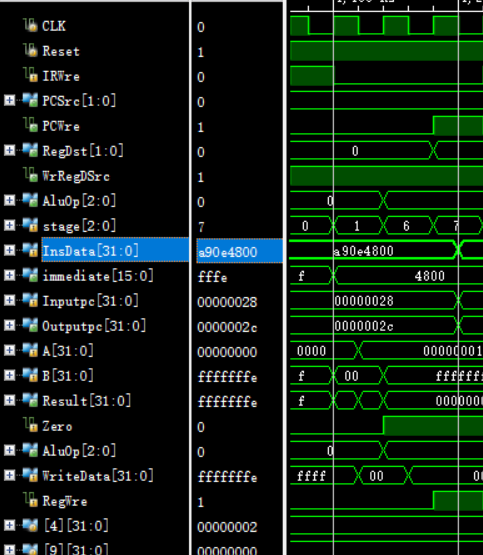
执行过程：将0号寄存器中的值与立即数-2相加，赋给$14,$14 = -2

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为无效电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 0，rt = 14，sa = fffe |
| 6 | 进行计算，同时将ALU输出结果存入AluOutDR中。 |
| 7 | 将RegWre置为1，并慢半个周期写入寄存器。同时置PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x00000028 | slt $9,$8,$14 | $9 = 0 |

波形：



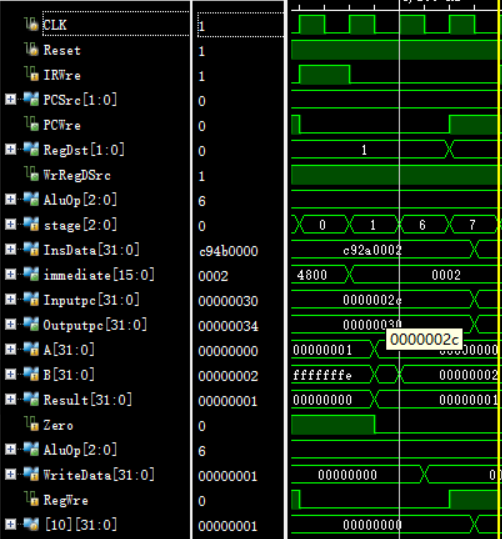
执行效果：将8号寄存器与14号寄存器相比较，若小于则赋1给9号寄存器，否则赋0，$9 = 0

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为无效电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 8，rt = 14，rd = 9 |
| 6 | 进行计算，同时将ALU输出结果存入AluOutDR中。 |
| 7 | 将RegWre置为1，并慢半个周期写入寄存器。同时置PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x0000002C | slti $10,$9,2 | $10 = 1 |

波形：



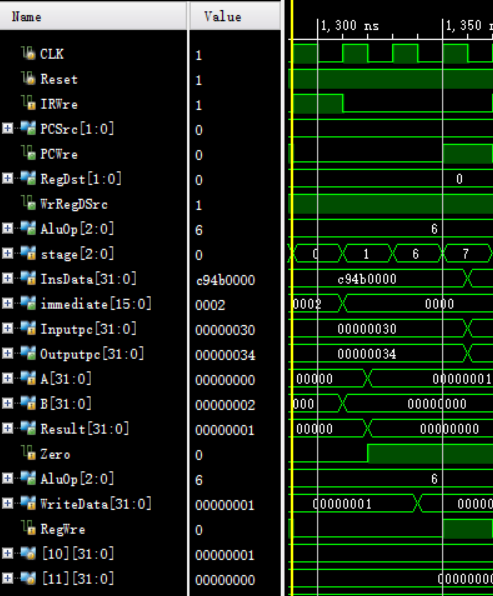
执行效果：将9号寄存器与立即数0相比较，若小于则赋1给10号寄存器，否则赋0，$10 = 1

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为无效电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 9，立即数 = 2，rt = 10 |
| 6 | 进行计算，同时将ALU输出结果存入AluOutDR中。 |
| 7 | 将RegWre置为1，并慢半个周期写入寄存器。同时置PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x00000030 | slti $11,$10,0 | $11 = 0 |

波形：



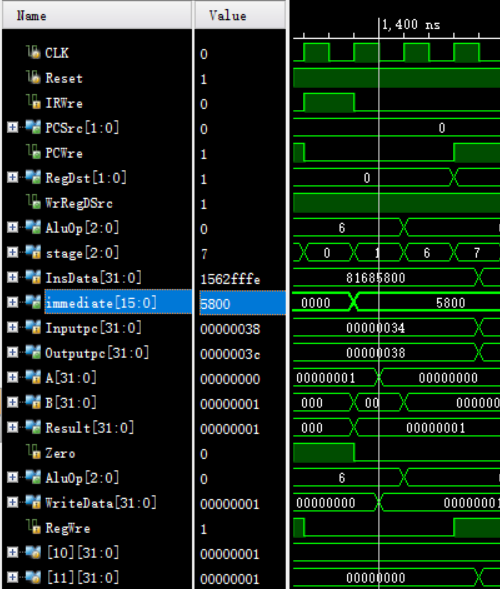
执行效果：将10号寄存器与立即数0相比较，若小于则赋1给11号寄存器，否则赋0，$11 = 0

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为无效电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 10，立即数 = 0，rt = 11 |
| 6 | 进行计算，同时将ALU输出结果存入AluOutDR中。 |
| 7 | 将RegWre置为1，并慢半个周期写入寄存器。同时置PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x00000034 | add $11,$11,$8 | $11 = 1 |

波形:



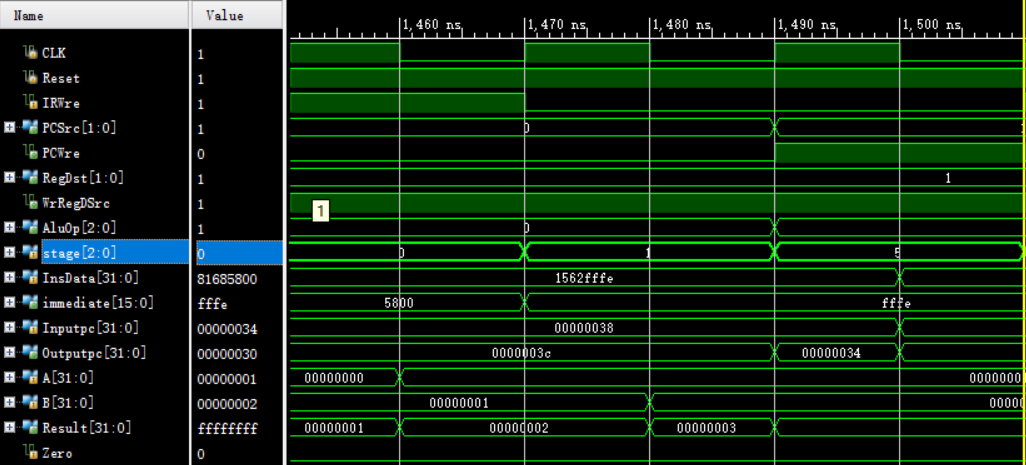
执行效果：将11号寄存器与8号寄存器相加，得到的值赋给11号寄存器，$11 = 1

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为无效电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 11，rt = 8，rd = 11 |
| 6 | 进行计算，同时将ALU输出结果存入AluOutDR中。 |
| 7 | 将RegWre置为1，并慢半个周期写入寄存器。同时置PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x00000038 | bne $11,$2,-2 (≠,转34) | 因为$11 不等于 $2,所以跳0X34 |

波形：



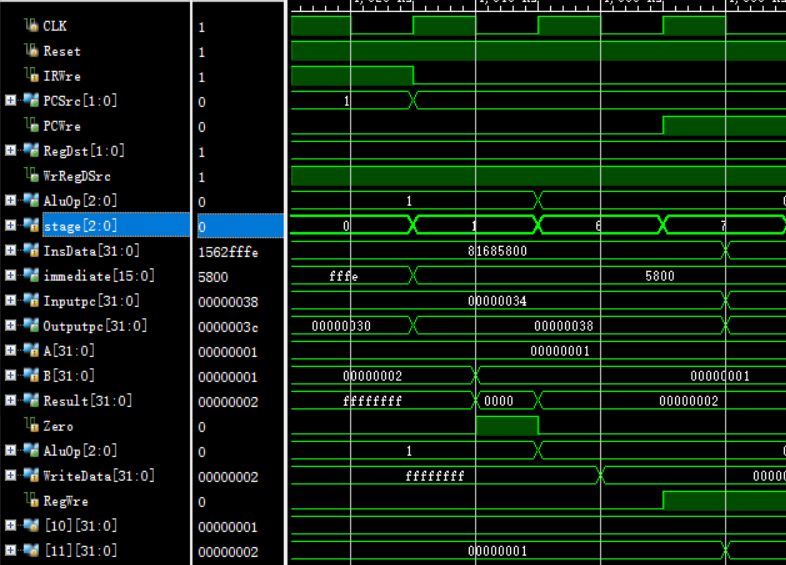
执行效果：将11号寄存器与2号寄存器相比较，若不相等则跳到0X34。

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 11，rt = 2，立即数 = 0FFFEH |
| 5 | 置PCWre = 1,并置PCSrc = 1（分支）。 |



|  |  |  |
| --- | --- | --- |
| 0x00000034 | add $11,$11,$8 | $11 = 2 |

波形：

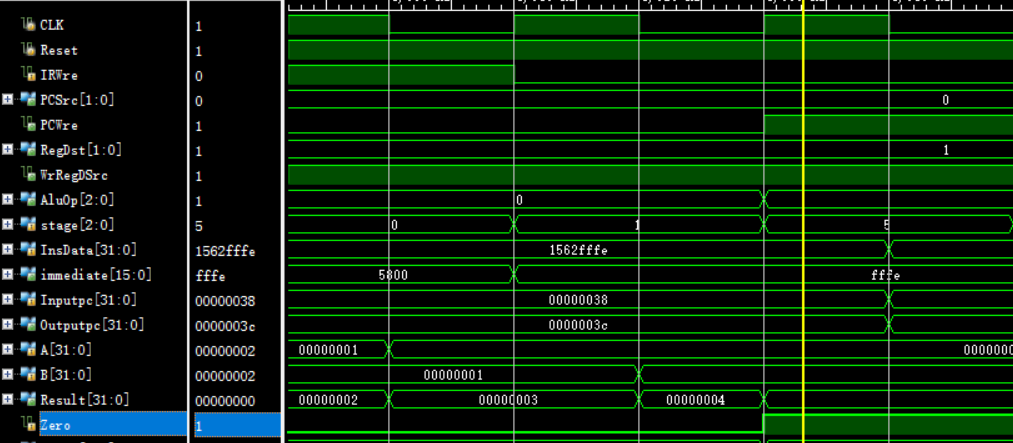


执行效果：将11号寄存器与8号寄存器相加，得到的值赋给11号寄存器，$11 = 2

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为无效电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 11，rt = 8，rd = 11 |
| 6 | 进行计算，同时将ALU输出结果存入AluOutDR中。 |
| 7 | 将RegWre置为1，并慢半个周期写入寄存器。同时置PCWre = 1。 |

|  |  |  |
| --- | --- | --- |
| 0x00000038 | bne $11,$2,-2 (≠,转34) | 因为$11 等于 $2,所以不跳0X34 |

波形：



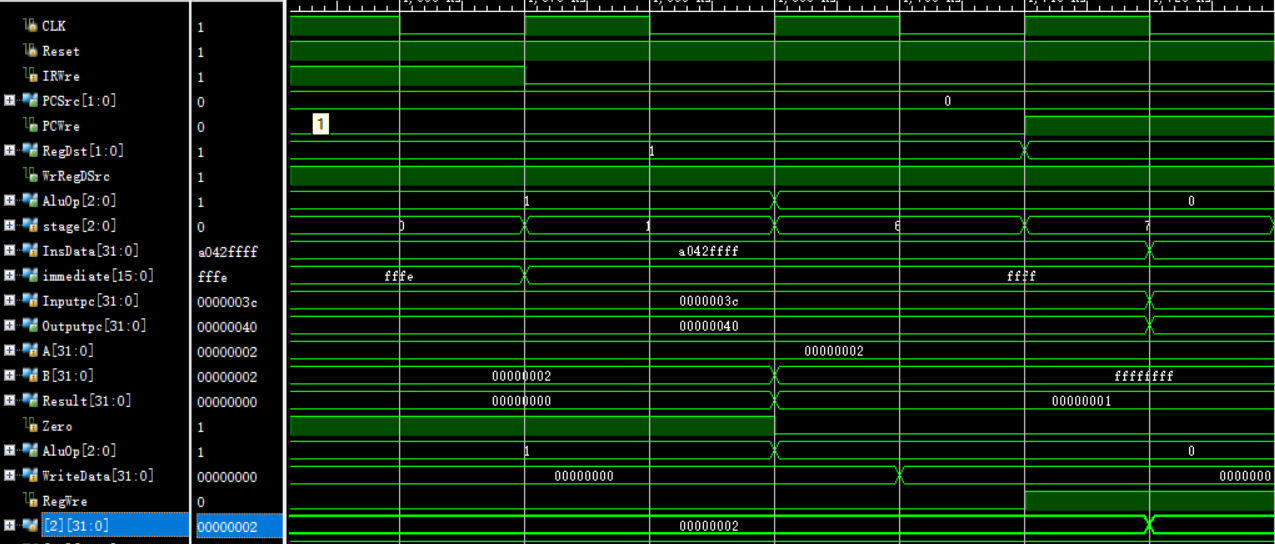
执行效果：将11号寄存器与2号寄存器相比较，若不相等则跳到0X34。

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 11，rt = 2，立即数 = 0FFFEH |
| 5 | 置PCWre = 1,并置PCSrc = 0。 |



|  |  |  |
| --- | --- | --- |
| 0x0000003C | addi $2,$2,-1 | $2 = 1 |

波形：



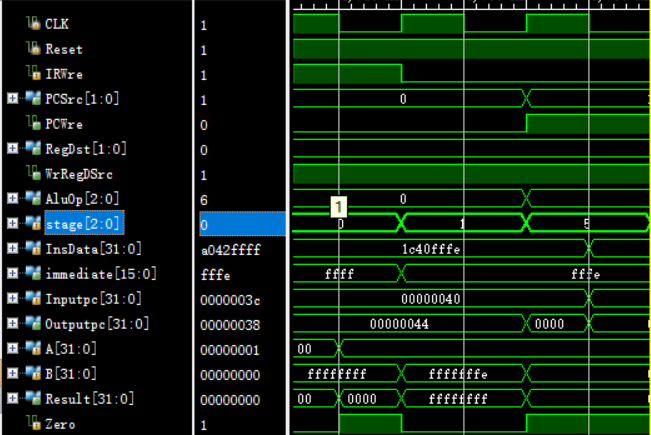
执行效果：将2号寄存器与立即数-1相加，得到的值赋给2号寄存器，$2 = 1

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为无效电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 2，rt =2，立即数 = -1 |
| 6 | 进行计算，同时将ALU输出结果存入AluOutDR中。 |
| 7 | 将RegWre置为1，并慢半个周期写入寄存器。同时置PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x00000040 | bgtz $2,-2 (>0,转3C) | $2>0,跳0x3c |

波形：



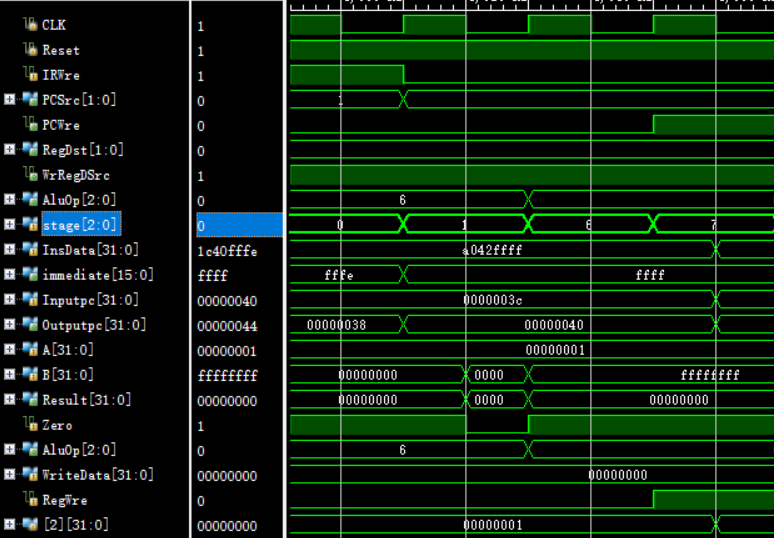
执行效果：$2 > 0 ,跳到0X3c

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 2，rt = 0，立即数 = 0FFFEH |
| 5 | 置PCWre = 1,并置PCSrc = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x0000003C | addi $2,$2,-1 | $2 = 0 |

波形：



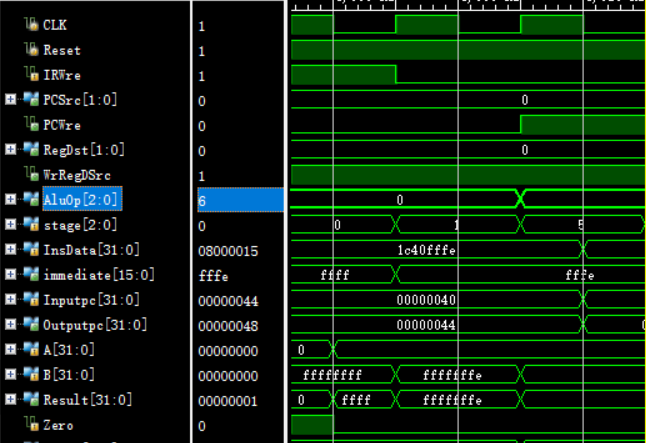
执行效果：将2号寄存器与立即数-1相加，得到的值赋给2号寄存器，$2 = 0

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为无效电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 2，rt =2，立即数 = -1 |
| 6 | 进行计算，同时将ALU输出结果存入AluOutDR中。 |
| 7 | 将RegWre置为1，并慢半个周期写入寄存器。同时置PCWre = 1。 |



|  |  |  |
| --- | --- | --- |
| 0x00000040 | bgtz $2,-2 (>0,转3C) | $2=,不跳0x3c |

波形：



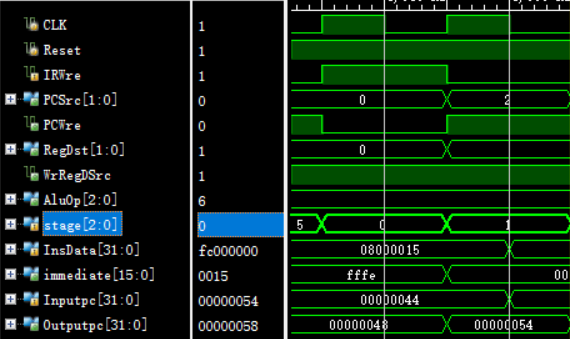
执行效果：$2 = 0 ,不跳到0X3c

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。决定AluOp码，rs，rt，sa，立即数与rd，同时将ALU操作数存入ADR，BDR  rs = 2，rt = 0，立即数 = 0FFFEH |
| 5 | 置PCWre = 1,并置PCSrc = 0。 |



|  |  |  |
| --- | --- | --- |
| 0x00000044 | j 0x0000054 | 跳到0x54 |

波形：



执行过程：跳到0X54。

|  |  |
| --- | --- |
| Stage | 效果 |
| 0 | PC改变，同时将PCSrc与PCWre置为低电平，防止重新写PC值。  将寄存器写信号RegWre，访存信号WR置为无效电平。防止进行寄存器写与内存写的操作。  IRWre = 1, |
| 1 | 指令改变。PCWre = 1。PCSrc = 2(跳转指令)。 |



|  |  |  |
| --- | --- | --- |
| 0x00000054 | halt | 停机 |

停机指令

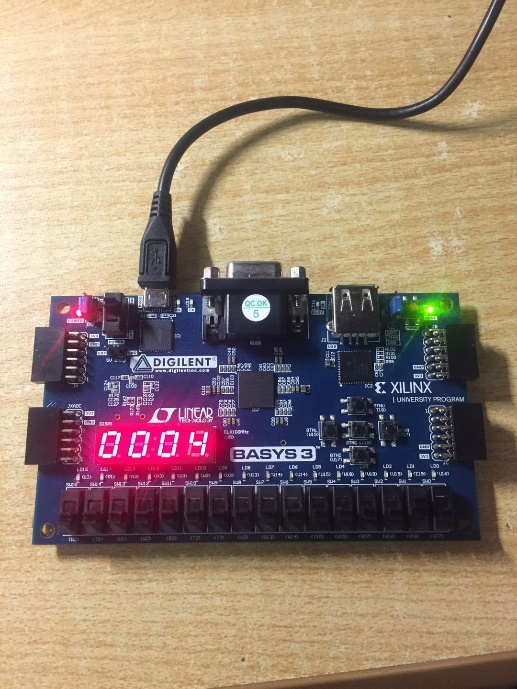
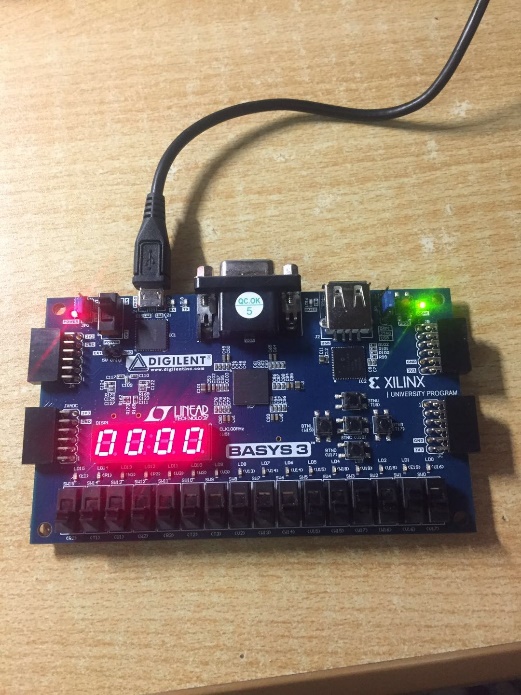
1. Basys实现：

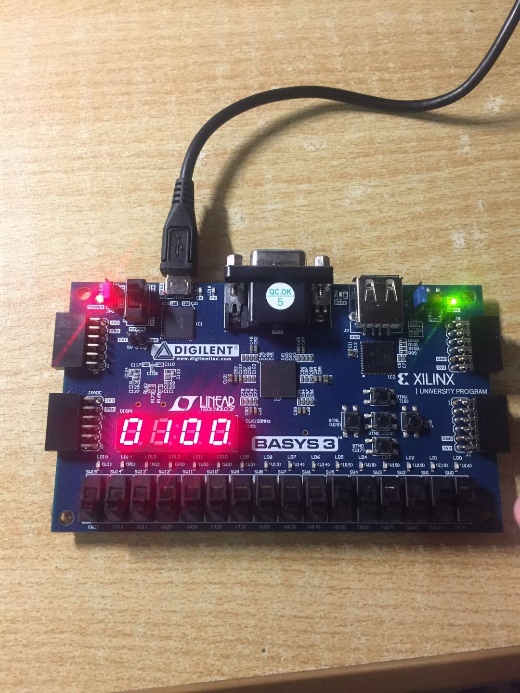
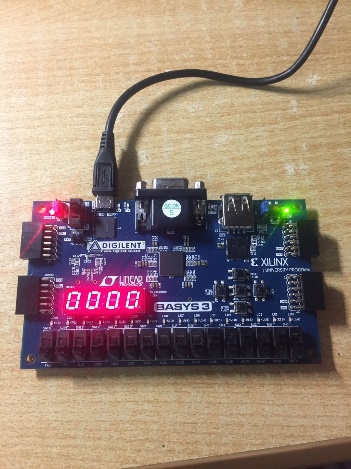
与单周期实现相同，显示模块也是分为消抖模块与数码管显示模块，直接从单周期CPU中拷贝过来即可。这里不多加解释。

下列显示前三条指令的结果截图：

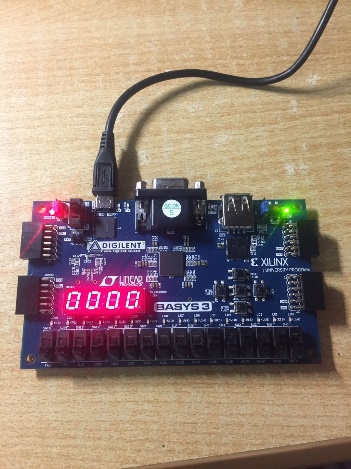
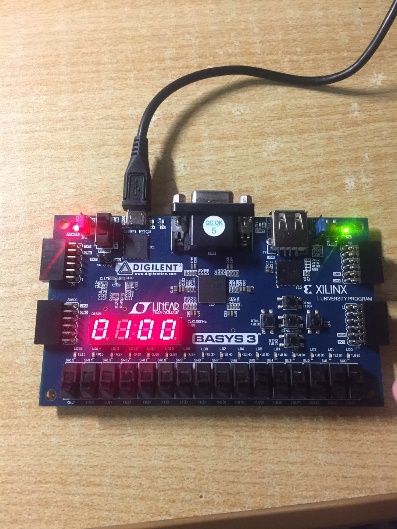
Add $1,$0,8

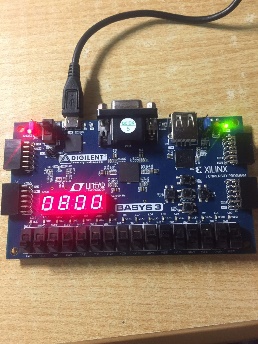
IF、ID、EXE:

PC: rs:

Rt: DB/ALU:

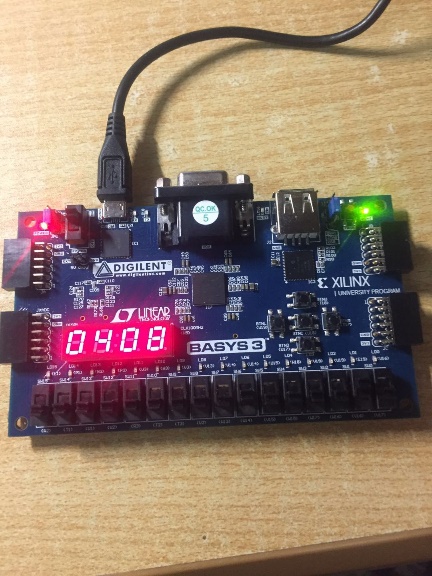
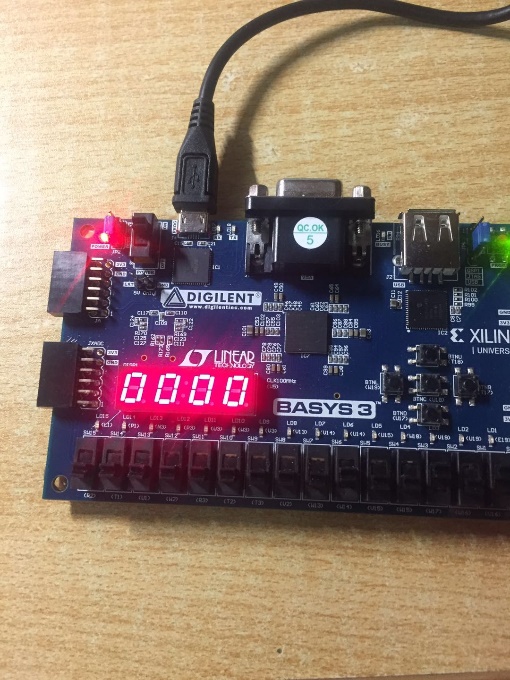
WB:

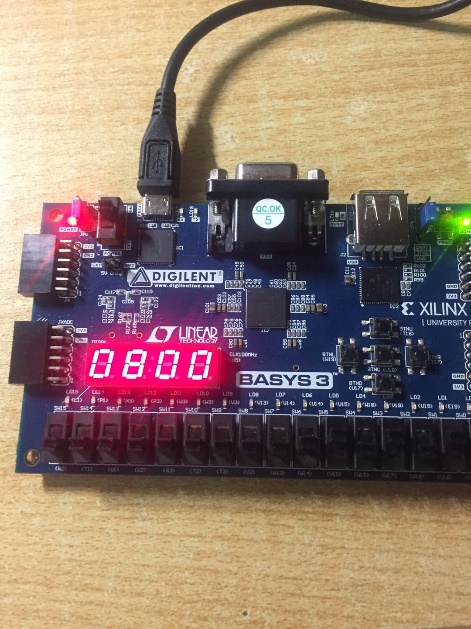
RS: rt:

ALU/DB:

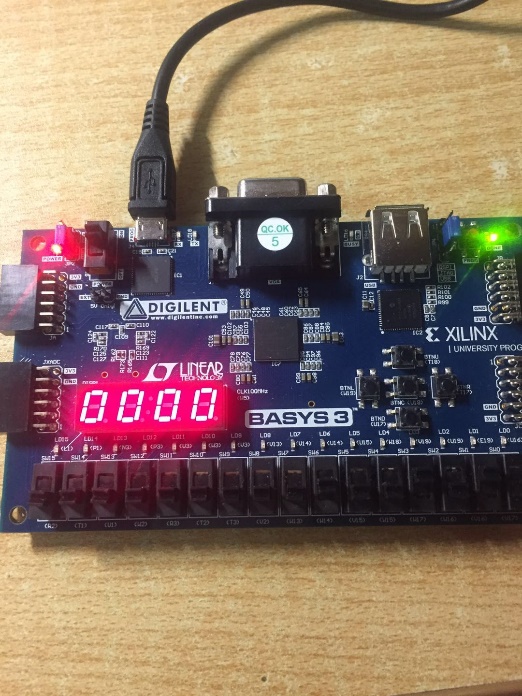
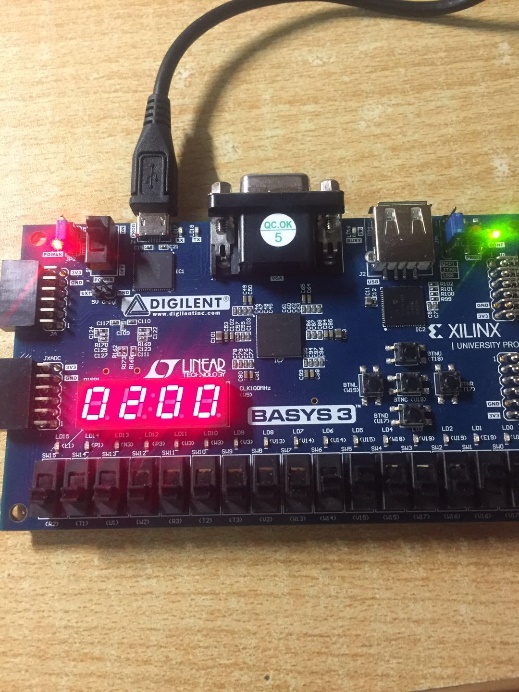
第二条指令：ori $2,$0,2

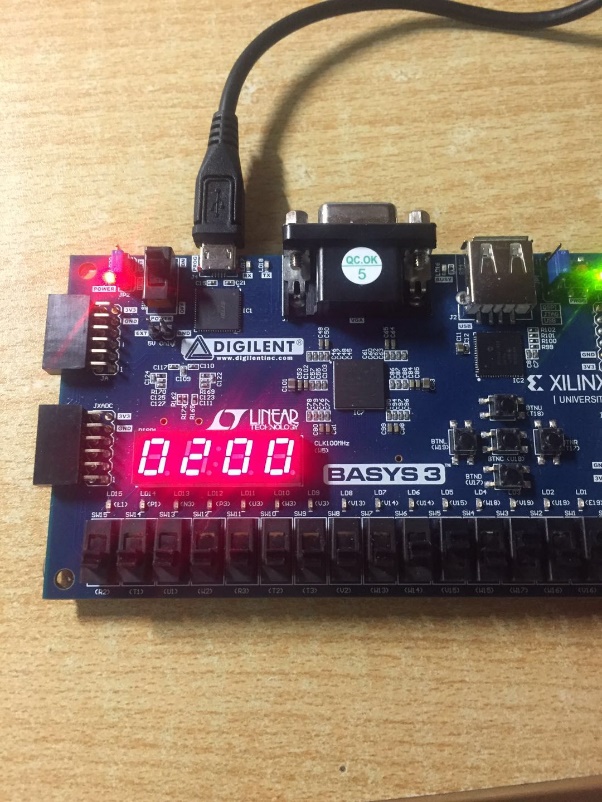
IF:

PC: rs:

Rt: alu/db:

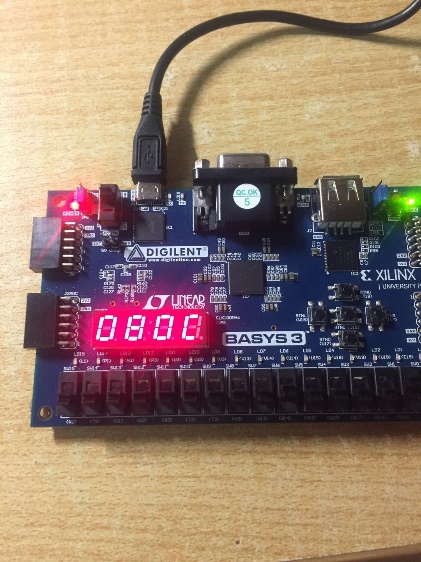
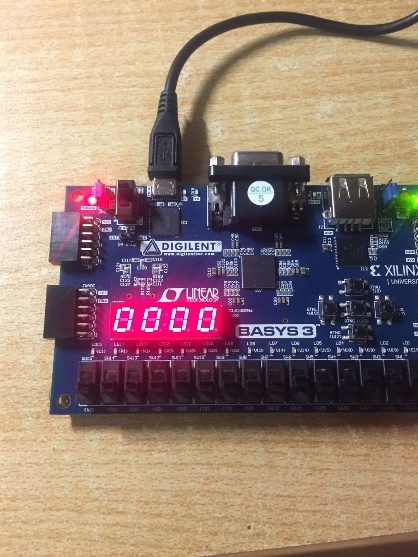
ID,EXE,WB:

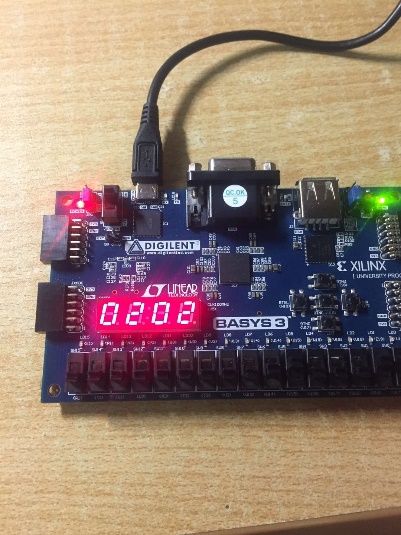
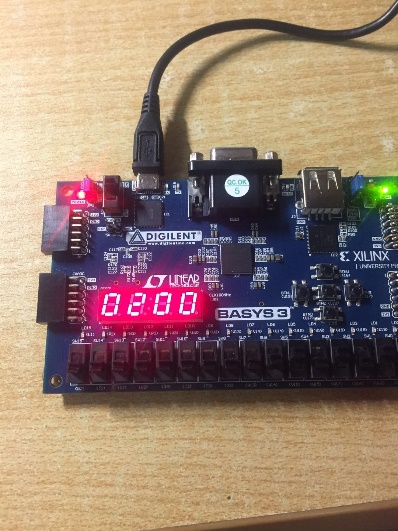
Rs:rt:

Alu/db:

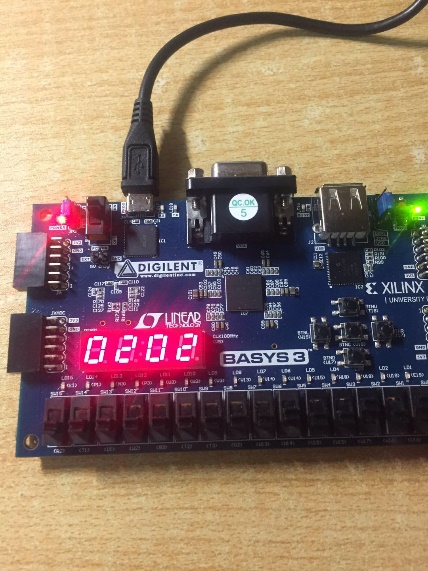
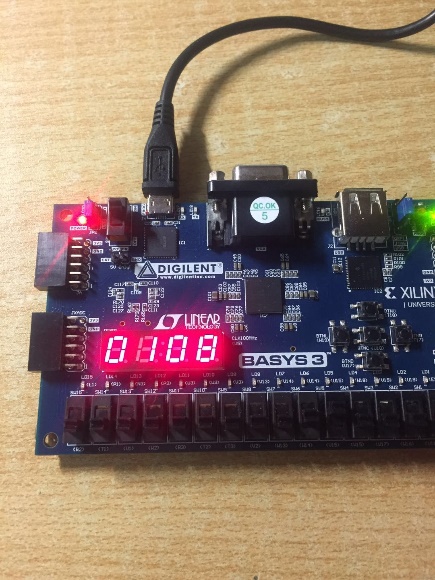
第三条指令：or $3,$2,$1

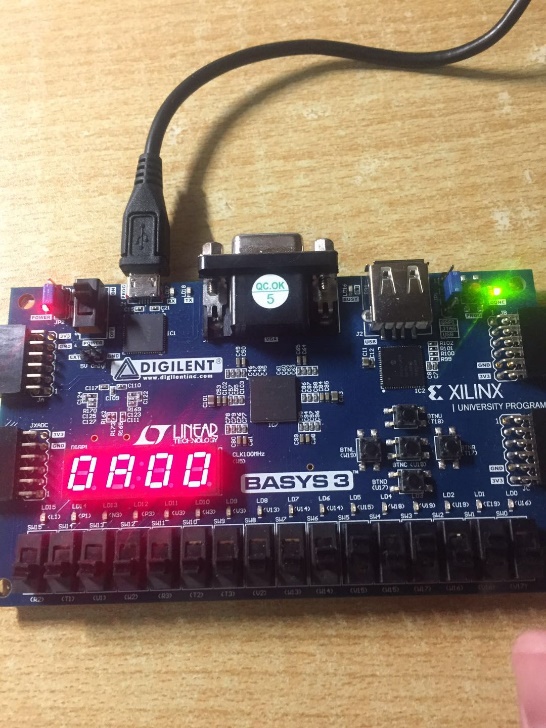
IF:

Pc:rs:

Rt:alu/db:

ID/EXE/WB:

Rs: rt:

alu/db:

说明：根据需要书写相关内容，如：CPU是怎么设计的？如何验证所设计的CPU是正确的？讲清楚这两个问题。

1、CPU设计的思想、方法：流程图，控制信号表的作用，如何应用？相关图（模块）的说明，及其相关实现代码说明（代码部分必须有侧重点，无需太泛！），强调关键部分，必须有文字加以说明。

2、如何验证你设计的CPU正确性？截波形图说明。波形图上必须包含与该条指令相关的、能说明该指令正确的信号（控制信号、结果数据和地址等），还有结果存放的寄存器及内容，或数据存储器地址及内容，或顺序执行的地址，或即将转移的地址。必须说明相关信号及其作用。对设计中要求实现的每条指令都必须这么做。

3、实现。如何在Basys3板上运行所设计的CPU？运行结果情况说明。

**切记！以上内容，在书写实验报告时，必须删除，不能保留在自己的实验报告中。**

* + - 1. **实验心得**

体会和建议。（**必须认真写，若过于简单，扣分！**）

**(**所写内容，也就是对本实验的总结。从新认识，温故而知新。如实验的整个过程中，所思考的问题、所碰到的问题等，以及最后如何获得解决，从中得到什么?等等，当然，还可能存在未解决的问题，或有所建议等。**整个来说，就是总结一下本实验的情况，温故而知新，这个道理显而易见。**)

**说明：实验报告电子文档单独存放，其余设计文档（Projiect），全部打包在一个文件中。**