****

**《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 |
| **专业（班级）** | **：** | 18软件工程2班 |
| **时间** | **：** | 2019年 10 月 22 日 |
| **组员** | **：** | 张楚明 18342125 |

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

### 一、 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。

### 二、 实验内容

设计一个单周期CPU，该CPU至少能实现以下指令功能操作。指令与格式如下：

**==> 算术运算指令**

（1）add rd , rs, rt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 000000 | 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”。**

（2）sub rd , rs , rt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 000001 | rs(5位) | rt(5位) | rd(5位) | reserved |

功能：rd←rs - rt。

（3）addiu rt , rs ,**immediate**

|  |  |  |  |
| --- | --- | --- | --- |
| 000010 | 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。

**==> 逻辑运算指令**

（4）andi rt , rs ,**immediate**

|  |  |  |  |
| --- | --- | --- | --- |
| 010000 | rs(5位) | rt(5位) | **immediate**(16位) |

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

（5）and rd , rs , rt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 010001 | rs(5位) | rt(5位) | rd(5位) | reserved |

功能：rd←rs & rt；逻辑与运算。

（6）ori rt , rs ,**immediate**

|  |  |  |  |
| --- | --- | --- | --- |
| 010010 | 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。

（7）or rd , rs , rt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 010011 | rs(5位) | rt(5位) | rd(5位) | reserved |

功能：rd←rs **|** rt；逻辑或运算。

**==>移位指令**

（8）sll rd, rt,sa

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 011000 | 未用 | rt(5位) | rd(5位) | sa(5位) | reserved |

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

**==>比较指令**

（9） slti rt, rs,**immediate** 带符号数

|  |  |  |  |
| --- | --- | --- | --- |
| 011100 | 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。

**==> 存储器读/写指令**

（10）sw rt ,**immediate(**rs) 写存储器

|  |  |  |  |
| --- | --- | --- | --- |
| 100110 | 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。

（11) lw rt , **immediate**(rs) 读存储器

|  |  |  |  |
| --- | --- | --- | --- |
| 100111 | 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。

**==> 分支指令**

（12）beq rs,rt,**immediate**

|  |  |  |  |
| --- | --- | --- | --- |
| 110000 | 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”。

（13）bne rs,rt,**immediate**

|  |  |  |  |
| --- | --- | --- | --- |
| 110001 | 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：与beq不同点是，不等时转移，相等时顺序执行。

（14）bltz rs,**immediate**

|  |  |  |  |
| --- | --- | --- | --- |
| 110010 | rs(5位) | 00000 | **immediate**(16位) |

功能：if(rs<$zero) pc←pc + 4 + (sign-extend)**immediate** <<2 else pc ←pc + 4。

**==>跳转指令**

（15）j addr

|  |  |
| --- | --- |
| 111000 | addr[27:2] |

功能：pc <－{(pc+4)[31:28],addr[27:2],2'b00}，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，剩下最高4位由pc+4最高4位拼接上。

**==> 停机指令**

（16）halt

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

功能：停机；不改变PC的值，PC保持不变。

### 实验原理

单周期CPU指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。时钟周期一般也称振荡周期（**如果晶振的输出没有经过分频就直接作为CPU的工作时钟，则时钟周期就等于振荡周期。若振荡周期经二分频后形成时钟脉冲信号作为CPU的工作时钟，这样，时钟周期就是振荡周期的两倍。**）

CPU在处理指令时，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根据指令字长度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，当然得到的“地址”需要做些变换才送入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。

单周期CPU，是在一个时钟周期内完成这五个阶段的处理。

**图1 单周期CPU指令处理过程**

**取指令**

**IF**

**指令译码**

**ID**

**指令执行**

**EXE**

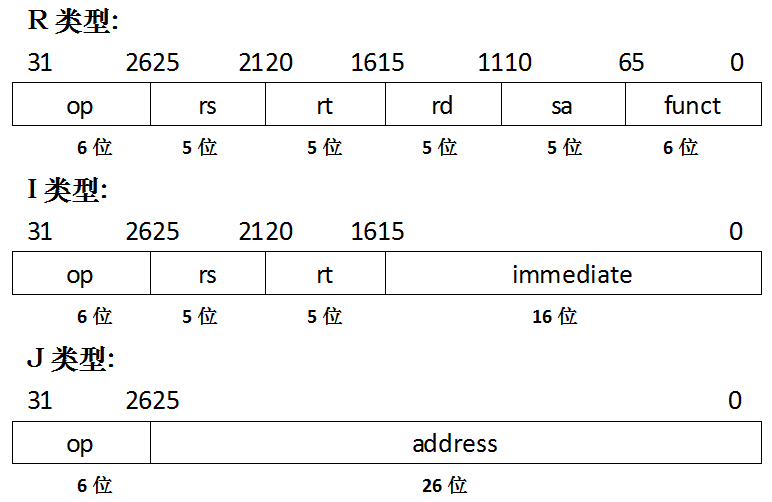
**存储器访问**

**MEM**

**结果写回**

**WB**

MIPS指令的三种格式：



其中，

**op：**为操作码；

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

**rt：**可读可写。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）；

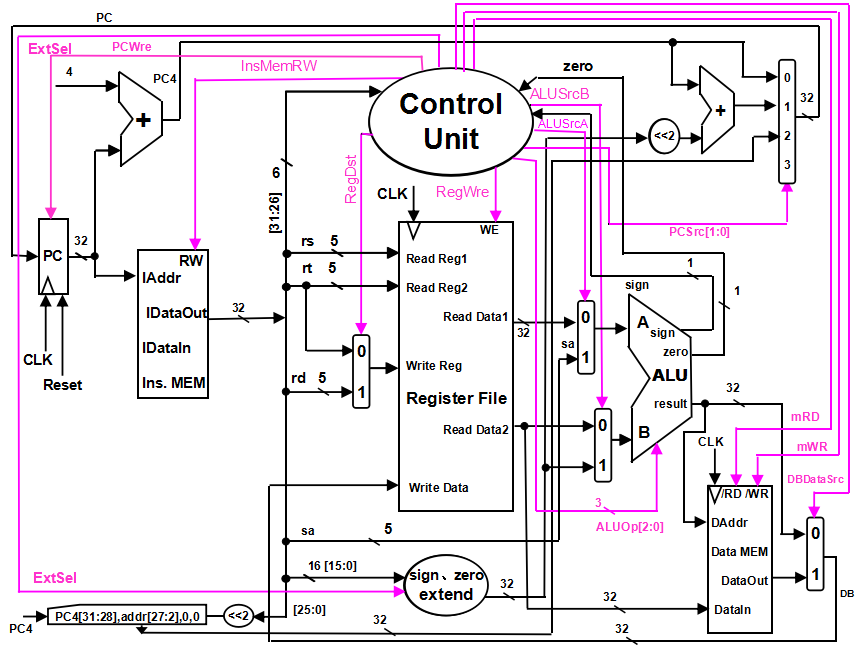
**rd：**只写。为目的操作数寄存器，寄存器地址（同上）；

**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与操作码配合使用；

**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（Laod）/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。



**图2 单周期CPU数据通路和控制线路图**

图2是一个简单的基本上能够在单周期CPU上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，先给出寄存器地址，读操作时不需要时钟信号，输出端就直接输出相应数据；而在写操作时，在 WE使能信号为1时，在时钟边沿触发将数据写入寄存器。图中控制信号作用如表1所示，表2是ALU运算功能表。

**表1 控制信号的作用**

|  |  |  |
| --- | --- | --- |
| 控制信号名 | 状态“0” | 状态“1” |
| **Reset** | 初始化PC为0 | PC接收新地址 |
| **PCWre** | PC不更改，相关指令：halt | PC更改，相关指令：除指令halt外 |
| **ALUSrcA** | 来自寄存器堆data1输出，相关指令：add、sub、addiu、or、and、andi、ori、slti、beq、bne、bltz、sw、lw | 来自移位数sa，同时，进行(zero-extend)sa，即 {{27{1'b0}},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、bltz | 来自sign或zero扩展的立即数，相关指令：addi、andi、ori、slti、sw、lw |
| **DBDataSrc** | 来自ALU运算结果的输出，相关指令：add、addiu、sub、ori、or、and、andi、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、bltz、sw、halt | 寄存器组写使能，相关指令：add、addiu、sub、ori、or、and、andi、slti、sll、lw |
| **InsMemRW** | 写指令存储器 | 读指令存储器(Ins. Data) |
| **mRD** | 输出高阻态 | **读数据存储器，相关指令：lw** |
| **mWR** | 无操作 | **写数据存储器，相关指令：sw** |
| **RegDst** | 写寄存器组寄存器的地址，来自rt字段，相关指令：addiu、andi、ori、slti、lw | 写寄存器组寄存器的地址，来自rd字段，相关指令：add、sub、and、or、sll |
| **ExtSel** | (zero-extend)**immediate（**0扩展**），**相关指令：andi、ori | (sign-extend)**immediate（**符号扩展**）**  ，相关指令：addiu、slti、sw、lw、beq、bne、bltz |
| **PCSrc[1..0]** | 00：pc<－pc+4，相关指令：add、addiu、sub、or、ori、and、andi、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0)；  01：pc<－pc+4+(sign-extend)**immediate<<2**，相关指令：beq(zero=1)、  bne(zero=0)、bltz(sign=1)；  10：pc<－{(pc+4)[31:28],addr[27:2],2'b00}，相关指令：j；  11：未用 | |
| **ALUOp[2..0]** | ALU 8种运算功能选择(000-111)，看功能表 | |

**相关部件及引脚说明：**

**Instruction Memory**：**指令存储器**，

Iaddr，指令存储器地址输入端口

IDataIn，指令存储器数据输入端口（指令代码输入端口）

IDataOut，指令存储器数据输出端口（指令代码输出端口）

RW，指令存储器读写控制信号，为0写，为1读

**Data Memory**：**数据存储器**，

Daddr，数据存储器地址输入端口

DataIn，数据存储器数据输入端口

DataOut，数据存储器数据输出端口

/RD，数据存储器读控制信号，为0读

/WR，数据存储器写控制信号，为0写

**Register File**：**寄存器组**

Read Reg1，rs寄存器地址输入端口

Read Reg2，rt寄存器地址输入端口

Write Reg，将数据写入的寄存器端口，其地址来源rt或rd字段

Write Data，写入寄存器的数据输入端口

Read Data1，rs寄存器数据输出端口

Read Data2，rt寄存器数据输出端口

WE，写使能信号，为1时，在时钟边沿触发写入

**ALU**： 算术**逻辑单元**

result，ALU运算结果

zero，运算结果标志，结果为0，则zero=1；否则zero=0

sign，运算结果标志，结果最高位为0，则sign=0，正数；否则，sign=1，负数

**表2 ALU运算功能表**

|  |  |  |
| --- | --- | --- |
| **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<B**  **不带符号** |
| **110** | **Y=(((A<B)&&(A[31] == B[31])) ||((A[31]==1&& B[31] == 0))) ? 1:0** | **比较A<B**  **带符号** |
| **111** | **Y = A ⊕ B** | **异或** |

**对应真值表如下:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Op | PCWre | ALUSrcA | ALUSrcB | DBDataSrc | InsMemRW | RegWre |
| 000000(add) | 1 | 0 | 0 | 0 | 1 | 1 |
| 000001(sub) | 1 | 0 | 0 | 0 | 1 | 1 |
| 000010(addiu) | 1 | 0 | 1 | 0 | 1 | 1 |
| 010000(addi) | 1 | 0 | 1 | 0 | 1 | 1 |
| 010001(and) | 1 | 0 | 0 | 0 | 1 | 1 |
| 010010(ori) | 1 | 0 | 1 | 0 | 1 | 1 |
| 010011(or) | 1 | 0 | 0 | 0 | 1 | 1 |
| 011000(sll) | 1 | 1 | 0 | 0 | 1 | 1 |
| 011100(slti) | 1 | 0 | 1 | 0 | 1 | 1 |
| 100110(sw) | 1 | 0 | 1 | 0 | 1 | 0 |
| 100111(lw) | 1 | 0 | 1 | 1 | 1 | 1 |
| 110000(beq) | 1 | 0 | 0 | 0 | 1 | 0 |
| 110001(bne) | 1 | 0 | 0 | 0 | 1 | 0 |
| 110010(bltz) | 1 | 0 | 0 | 0 | 1 | 0 |
| 111000(j) | 1 | 0 | 0 | 0 | 1 | 0 |
| 111111(halt) | 0 | 0 | 0 | 0 | 1 | 0 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Op | mRD | mWR | RegDst | ExtSel | PCSrc |  | ALUOP |
| 000000(add) | 0 | 0 | 1 | 0 | 00 |  | 000 |
| 000001(sub) | 0 | 0 | 1 | 0 | 00 |  | 001 |
| 000010(addiu) | 0 | 0 | 0 | 1 | 00 |  | 000 |
| 010000(addi) | 0 | 0 | 0 | 0 | 00 |  | 100 |
| 010001(and) | 0 | 0 | 1 | 0 | 00 |  | 100 |
| 010010(ori) | 0 | 0 | 0 | 0 | 00 |  | 011 |
| 010011(or) | 0 | 0 | 1 | 0 | 00 |  | 011 |
| 011000(sll) | 0 | 0 | 1 | 0 | 00 |  | 010 |
| 011100(slti) | 0 | 0 | 0 | 1 | 00 |  | 110 |
| 100110(sw) | 0 | 1 | 0 | 1 | 00 |  | 000 |
| 100111(lw) | 1 | 0 | 0 | 1 | 00 |  | 000 |
| 110000(beq) | 0 | 0 | 0 | 1 | 01(zero)/00 |  | 001 |
| 110001(bne) | 0 | 0 | 0 | 1 | 01(!zero)/00 |  | 001 |
| 110010(bltz) | 0 | 0 | 0 | 1 | 01(sign)/00 |  | 000 |
| 111000(j) | 0 | 0 | 0 | 0 | 10 | | 000 |
| 111111(halt) | 0 | 0 | 0 | 0 | 00 | | 000 |

（注：三条分支指令用异或来实现）

（在读写指令、跳转指令和停机指令时设置ALU的控制输入为000）

### 实验设备

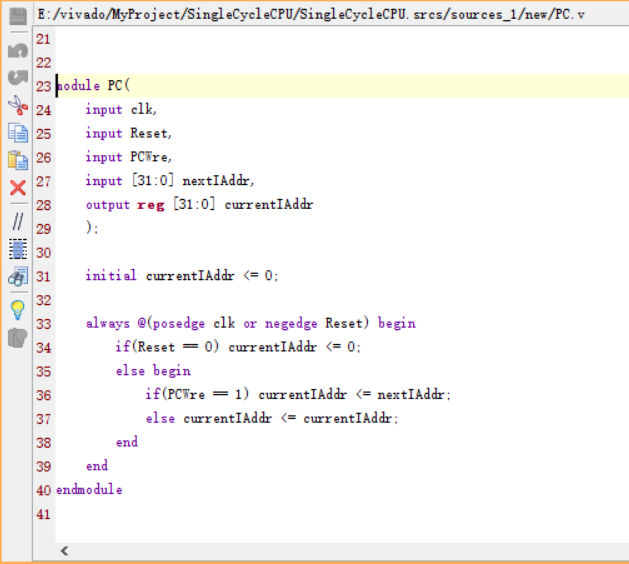
PC机一台，BASYS 3 实验板一块，Xilinx Vivado 开发软件一套。

### 实验过程与结果

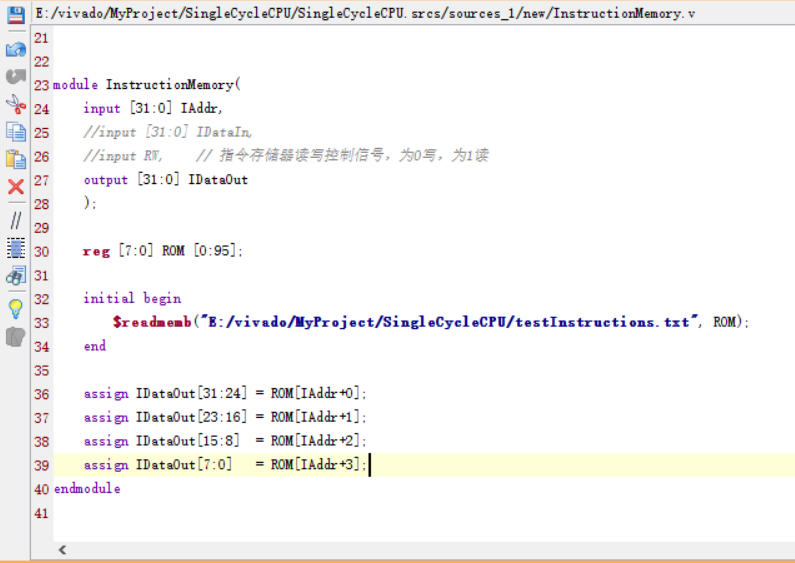
1. **CPU设计(引脚接口说明见第三点实验原理)**

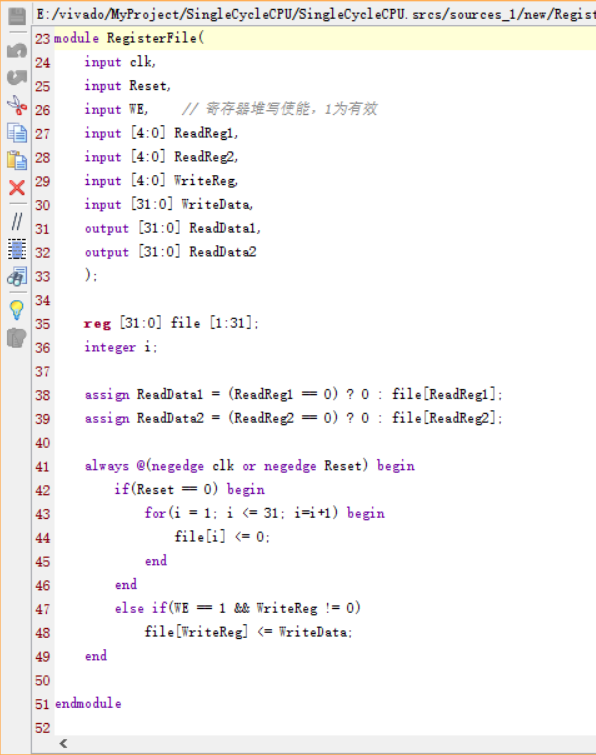
五个底层模块:

1. IF阶段对应的PC

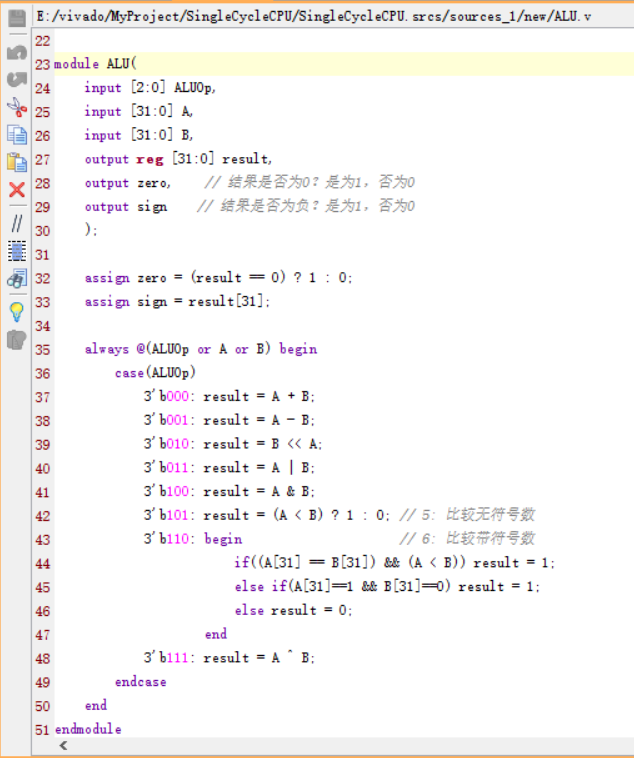


1. ID阶段对应的InstructionMemory(从外部文件”testInstructions.txt”中读取二进制指令,测试指令见第五点最后)和RegisterFile

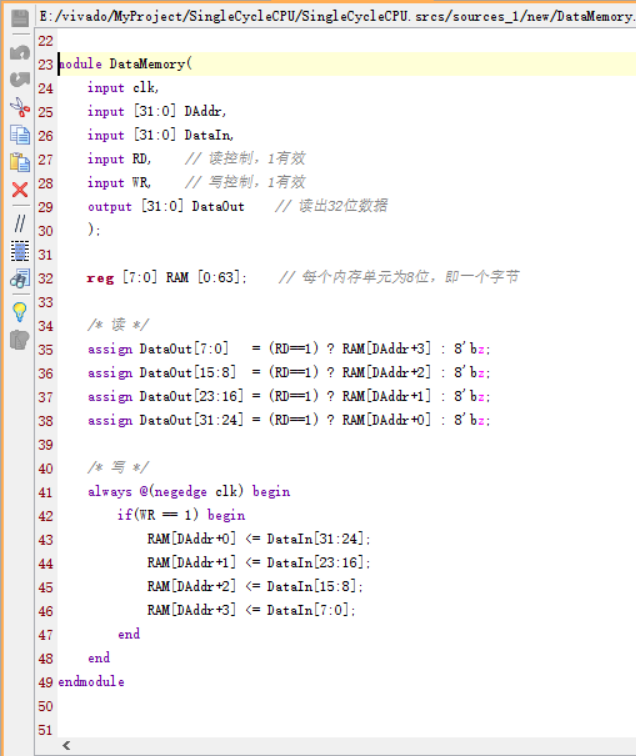




1. EX阶段对应的ALU



1. MEM阶段对应的DataMemory



1. WB阶段对应的RegisterFile(见ID阶段)

除此之外，还需要生成控制信号的控制单元（ControlUnit），还要有对立即数进行扩展的

ImmediateExtend。最后，还需要使用各种不同的数据选择器（Mux）构成完整的数据通路等等,由于代码块比较多在这里不一一展示。

1. **验证CPU的正确性**

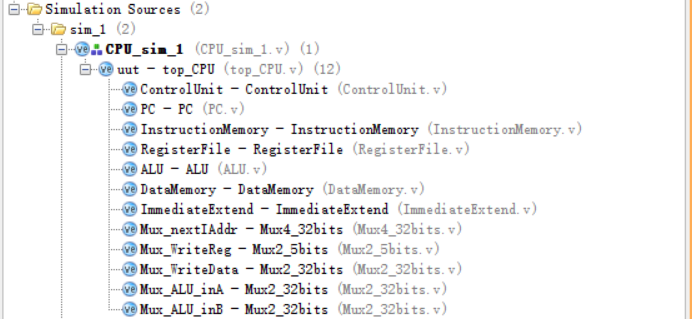
(1)测试指令集如下:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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** | addiu $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** | add $3,$2,$1 | **000000** | **00010** | **00001** | **00011 00000000000** | **=** | 00411800 |
| **0x0000000C** | sub $5,$3,$2 | **000001** | **00011** | **00010** | **00101 00000000000** | **=** | 04622800 |
| **0x00000010** | and $4,$5,$2 | **010001** | **00101** | **00010** | **00100 00000000000** | **=** | 44A22000 |
| **0x00000014** | or $8,$4,$2 | **010011** | **00100** | **00010** | **01000 00000000000** | **=** | 4C824000 |
| **0x00000018** | sll $8,$8,1 | **011000** | **01000** | **01000** | **0000 0000 0000 0001** | **=** | 61080001 |
| **0x0000001C** | **bne $8,$1,-2 (≠,转18)** | **110001** | **01000** | **00001** | **1111 1111 1111 1110** | **=** | C428FFFE |
| **0x00000020** | slti $6,$2,4 | **011100** | **00110** | **00010** | **0000 0000 0000 0100** | **=** | 70460004 |
| **0x00000024** | slti $7,$6,0 | **011100** | **00111** | **00110** | **0000 0000 0000 0000** | **=** | 70C70000 |
| **0x00000028** | addiu $7,$7,8 | **000010** | **00111** | **00111** | **0000 0000 0000 1000** | **=** | 08D70008 |
| **0x0000002C** | **beq $7,$1,-2 (=,转28)** | **110000** | **00111** | **00001** | **1111 1111 1111 1110** | **=** | C027FFFE |
| **0x00000030** | sw $2,4($1) | **100110** | **00001** | **00010** | **0000 0000 0000 0100** | **=** | 98220004 |
| **0x00000034** | lw $9,4($1) | **100111** | **00001** | **01001** | **0000 0000 0000 0100** | **=** | 9C290004 |
| **0x00000038** | addiu $10,$0,-2 | **000010** | **01010** | **00000** | **1111 1111 1111 1110** | **=** | 080AFFFE |
| **0x0000003C** | addiu $10,$10,1 | **000010** | **01010** | **01010** | **0000 0000 0000 0001** | **=** | 084A0001 |
| **0x00000040** | **bltz $10,-2(<0,转3C)** | **110010** | **01010** | **00000** | **1111 1111 1111 1110** | **=** | C80AFFFE |
| **0x00000044** | **andi $11,$2,2** | **010000** | **01011** | **00010** | **0000 0000 0000 0010** | **=** | 404B0002 |
| **0x00000048** | **j 0x00000050** | **111000** | **00000000000000000000010100** | | | **=** | E0000014 |
| **0x0000004C** | or $8,$4,$2 | **010011** | **00100** | **00010** | **01000 0000000000** | **=** | 8C881000 |
| **0x00000050** | **halt** | **111111** | **00000** | **00000** | **0000000000000000** | **=** | FC000000 |

(2) 可以推导得到指令的实际执行过程、寄存器的变化、指令跳转情况以及ALU的运算结果等信息，如下表:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 指令地址 | 汇编程序 | 寄存器变化  （十进制） | 跳转情况 | ALU 结果低 8 位  （十六进制） |
| 0x00000000 | addiu $1,$0,8 | $1 = 8 |  | 08 |
| 0x00000004 | ori $2,$0,2 | $2 = 2 |  | 02 |
| 0x00000008 | add $3,$2,$1 | $3 = 10 |  | 0A |
| 0x0000000C | sub $5,$3,$2 | $5 = 8 |  | 08 |
| 0x00000010 | and $4,$5,$2 | $4 = 0 |  | 00 |
| 0x00000014 | or $8,$4,$2 | $8 = 2 |  | 02 |
| 0x00000018 | sll $8,$8,1 | $8 = 4 |  | 04 |
| 0x0000001C | bne $8,$1,-2 |  | ≠，转18 | FC |
| 0x00000018 | sll $8,$8,1 | $8 = 8 |  | 08 |
| 0x0000001C | bne $8,$1,-2 |  | =，继续 | 00 |
| 0x00000020 | slti $6,$2,4 | $6 = 1 |  | 01 |
| 0x00000024 | slti $7,$6,0 | $7 = 0 |  | 00 |
| 0x00000028 | addiu $7,$7,8 | $7 = 8 |  | 08 |
| 0x0000002C | beq $7,$1,-2 |  | =， 转28 | 00 |
| 0x00000028 | addiu $7,$7,8 | $7 = 16 |  | 16 |
| 0x0000002C | beq $7,$1,-2 |  | ≠，继续 | 08 |
| 0x00000030 | sw $2,4($1) |  |  | 0C |
| 0x00000034 | lw $9,4($1) | $9 = 2 |  | 0C |
| 0x00000038 | addiu $10,$0,-2 | $10 = -2 |  | FE |
| 0x0000003C | addiu $10,$10,1 | $10 = -1 |  | FF |
| 0x00000040 | bltz $10,-2 |  | <0，转3C | FF |
| 0x0000003C | addiu $10,$10,1 | $10 = 0 |  | 00 |
| 0x00000040 | bltz $10,-2 |  | ≮0，继续 | 00 |
| 0x00000044 | andi $11,$2,2 | $11 = 2 |  | 02 |
| 0x00000048 | j 0x00000050 |  | 转50 | 00 |
| 0x0000004C | or $8,$4,$2 |  |  | 00 |
| 0x00000050 | halt |  |  | 00 |

(3)仿真模块结构如下图: (设置仿真时间单位与仿真时钟周期相同，为100ns)



(4)下面将对16条测试指令进行注意验证: (由于指令数较多而仿真波形时间跨度较小，所以令指令1~5共一图；指令6~8共一图；指令9、10共一图；指令11~13共一图；指令14~16共一图)

1. addiu $1, $0, 8 （0~100ns）

当前PC地址（即currentIAddr）为0x00000000。PCSrc 信号为00，顺序执行； rs和rt分别为0号和1号，执行指令后值分别为0和8（注意寄存器堆的写入发生在时钟下降沿）；

ALUOp为000，做加法运算，运算结果为8。ALUSrcA信号为0，ALU的输入A为 rs寄存器的值；ALUSrcB信号为1，ALU的输入B为符号扩展（ExtSel信号1）后的立即数；

RegDst信号为0，因此目的寄存器是rt；DBDataSrc信号为0，写回寄存器的值来自ALU的输出。该指令不涉及数据存储器，因此mRD和mWR均为0。

1. ori $2, $0, 2 （100~200ns）

当前PC为0x00000004。PCSrc为00，顺序执行； rs和rt分别为0号和2号，执行后值分别为0和2；

ALUOp为003，做逻辑或运算，结果为2。ALUSrcA信号为0，输入rs寄存器的值；

ALUSrcB信号为1，输入零扩展（ExtSel信号为0）后的立即数； RegDst信号为0，因此目的寄存器是rt；DBDataSrc信号为0，写回寄存器的值来自ALU的输出。该指令不涉及数据存储器，因此mRD和mWR均为0。

1. add $3, $2, $1 （200~300ns）

当前PC为0x00000008。PCSrc为00，顺序执行； rs和rt分别是2号和1号，执行前后值不变，分别为2和8；

ALUOp为000，做加法运算，结果为8+2=10（十六进制的0a）。ALUSrcA和 ALUSrcB信号均为0，因此ALU的输入来源都是寄存器堆；

RegDst信号为1，因此目的寄存器由rd指定而不是rt；DBDataSrc信号为0，写回寄存器的值来自ALU的输出。

该指令不涉及数据存储器，因此mRD和mWR均为0。

1. sub $5, $3, $2 （300~400ns）

当前PC为0x0000000c。PCSrc为00，顺序执行； rs和rt分别是3号和2号，值分别为10和2；

ALUOp为001，做减法运算，结果为10-2=8。ALUSrcA和ALUSrcB信号均为0，因此ALU的输入来源都是寄存器堆；

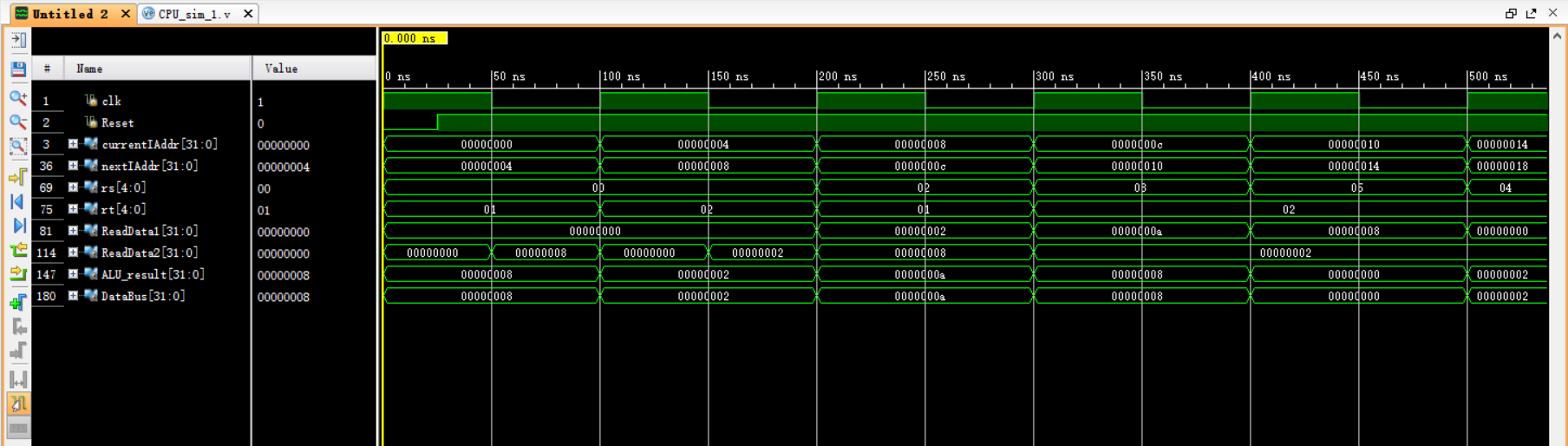
RegDst信号为1，因此目的寄存器由rd指定而不是rt；DBDataSrc信号为0，写回寄存器的值来自ALU的输出。该指令不涉及数据存储器，因此mRD和mWR均为0。

1. and $4, $5, $2 （400~500ns）

当前PC为0x00000010，PCSrc为00，顺序执行； rs和rt分别是5号和2号，值分别为8和2；

ALUOp为100，做逻辑与运算，结果为8&2=0。ALUSrcA和ALUSrcB信号均为0，因此ALU的输入来源都是寄存器堆；

RegDst信号为1，因此目的寄存器由rd指定而不是rt；DBDataSrc为0，写回寄存器的值来自ALU的输出。该指令不涉及数据存储器，因此mRD和mWR均为0。



（指令1~5图）

1. or $8, $4, $2 （500~600ns）

当前PC为0x00000014。PCSrc为00，顺序执行； rs和rt分别是4号和2号，值分别为0和2；

ALUOp为011，做逻辑或运算，结果为0|2=2；ALUSrcA和ALUSrcB信号均为0，因此ALU的输入来源都是寄存器堆；

RegDst信号为1，因此目的寄存器由rd指定而不是rt；DBDataSrc信号为0，写回寄存器的值来自ALU的输出。

该指令不涉及数据存储器，因此mRD和mWR均为0。

1. sll $8, $8, 1 （600~700ns）

当前PC为0x00000018，PCSrc为0，顺序执行； rt寄存器为8号，其值为2<<1=4；

ALUOp为010，做逻辑左移操作，结果为4；ALUSrcA信号为1，输入为经零扩展

（ExtSel为0）的sa立即数，ALUSrcB信号为0，输入rt寄存器的值；

RegDst信号为1，因此目的寄存器由rd指定；DBDataSrc信号为0，写回寄存器的值来自ALU的输出。

该指令不涉及数据存储器，因此mRD和mWR均为0。

1. bne $8, $1, -2 （700~800ns）

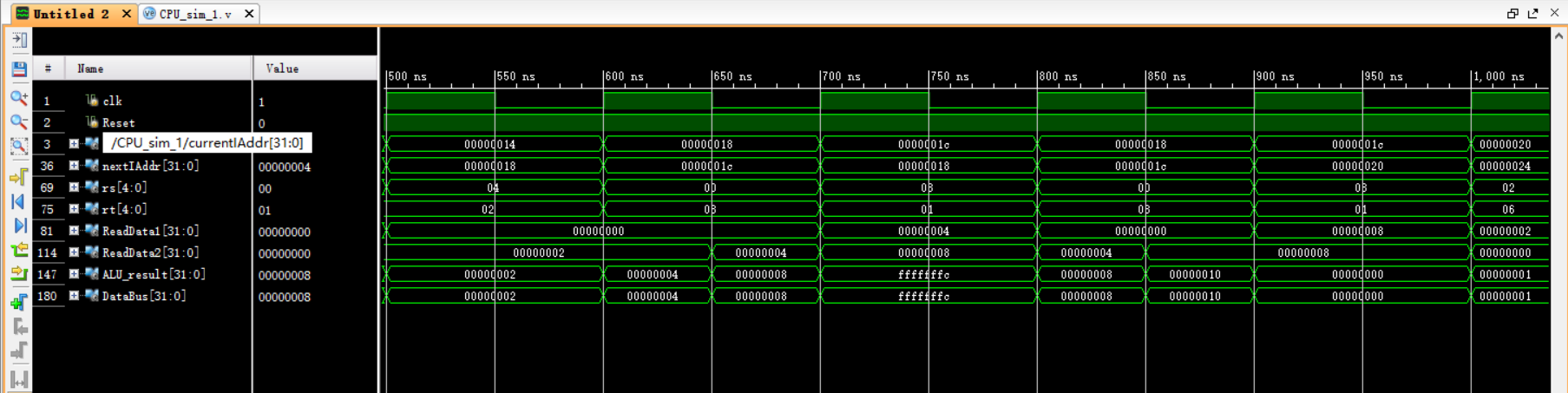
当前PC为0x0000001c。这是跳转指令，PCSrc待定，一会再分析。

rs和rt分别是8号和1号，值分别为4和8；

ALUOp为001，做减法运算，结果为4-8=-4，补码为0xfffffffc。此结果不是0，故ALU的zero输出为0，这表示比较结果不相等，bne条件为真。该结果（zero=0）再被送入控制单元，控制单元根据规则发出控制信号PCSrc为01，表示nextIAddr 为currentIAddr+4-2×4，下一步将跳转到分支去。由波形图可以看出，当前指令地址为0x0000001c，下一条指令的地址为00000018，这证明跳转确实发生了。该指令不涉及写回寄存器堆，也不涉及数据存储器，因此RegWre、mRD、mWR 信号均为0。

另外，900~1000ns也为0x0000001c处的bne指令，只不过这时ALU的zero输出为1，表示结果相等，bne条件为假。PCSrc信号00，不再分支，而是顺序执行到

0x00000020。



（指令6~8图）

1. slti $6, $2, 4 （1000~1100ns）

当前PC为0x00000020。PCSrc为00，顺序执行； rs和rt分别为2号和6号，执行后值分别为2和1；

ALUOp为110，做带符号比较操作，结果为2<1=1。ALUSrcA信号为0，输入rs 寄存器的值；ALUSrcB信号为1，输入符号扩展（ExtSel信号为1）后的立即数； RegDst信号为1，因此目的寄存器是rt；DBDataSrc信号为0，写回寄存器的值来自ALU的输出。

该指令不涉及数据存储器，因此mRD和mWR均为0。

1. beq$7, $1, -2 （1300~1400ns）

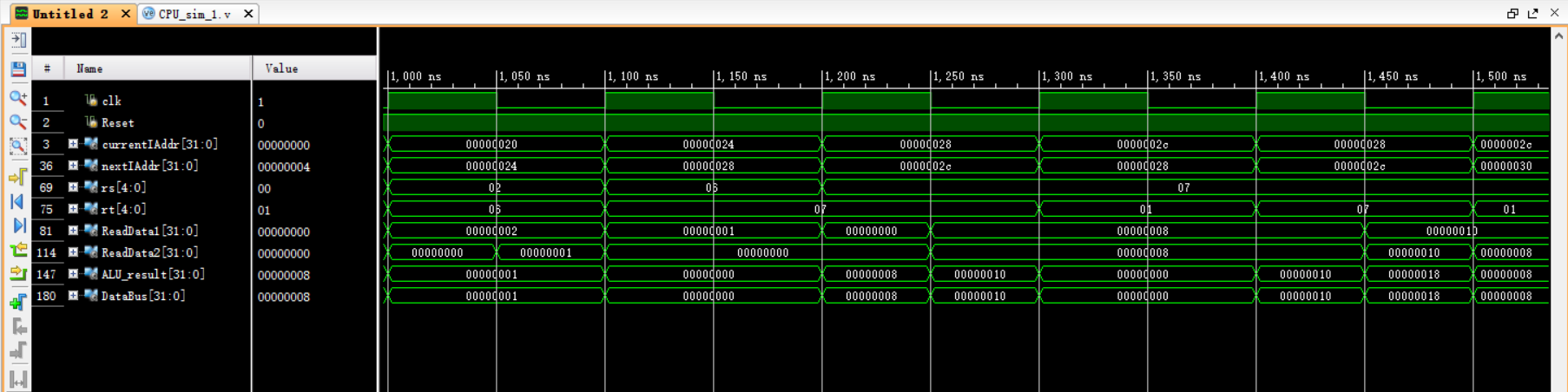
当前PC为0x0000002c，这是跳转指令。 rs和rt分别为7号和1号，值分别为8和8；

ALUOp为001，做减法运算，结果为8-8=0，ALU的zero输出为1，表示运算数相等，beq条件为真。因此PCSrc为10，，表示nextIAddr为currentIAddr+4-2×4，

下一步将跳转到分支去。由波形图可以看出，当前指令地址为0x0000002c,下一条指令地址为0x00000028，因此确实发生了跳转。

该指令不涉及写回寄存器堆，也不涉及数据存储器，因此RegWre、mRD、mWR 信号均为0。

另外，在1500ns~1600ns也是bne指令，这里的rs和rt的值不再相等，ALU的zero 输出为0，beq条件为假。PCSrc信号变为00，顺序执行。



（指令9、10图）

1. sw $2, 4($1) （1600ns~1700ns）

当前PC为0x00000030，PCSrc为00，顺序执行； rs和rt分别为1号和2号，值分别为8和2；

ALUOp为000，做加法运算；ALUSrcA为0，运算数A为寄存器堆的rs中的值，

ALUSrcB为1，表示运算数B为符号扩展（ExtSel为1）后的立即数；最后ALU运算出来的结果为8+4=12，表示数据存储器地址。

此时mWR信号变为1，数据存储器写使能有效，根据数据通路图便可得知，在时钟下降沿到来时，rt寄存器的值（即8）被写入数据存储器，写入地址即是ALU计算结果，即12。

RegWre信号为0，结果不需写回寄存器堆。

1. lw $9, 4($1) （1700~1800ns）

当前PC为0x00000034，PCSrc为00，顺序执行； rs为1号，其值为8；rt为9号，在执行前值为0；

ALUOp为000，做加法运算；ALUSrcA为0，运算数A为寄存器堆的rs中的值，

ALUSrcB为1，表示运算数B为符号扩展（ExtSel为1）后的立即数；最后ALU运算出来的结果为8+4=12，表示数据存储器地址。

此时mWR信号变为0，不可写；mRD信号变为1，数据存储器读使能有效；

DBDataSrc信号为1，数据总线（DataBus）上的数据来自数据存储器的输出

（DataOut），值为2。

RegWre信号为1，数据总线上的数据被写入寄存器堆。

1. bltz $10, -2 （1800~1900ns）

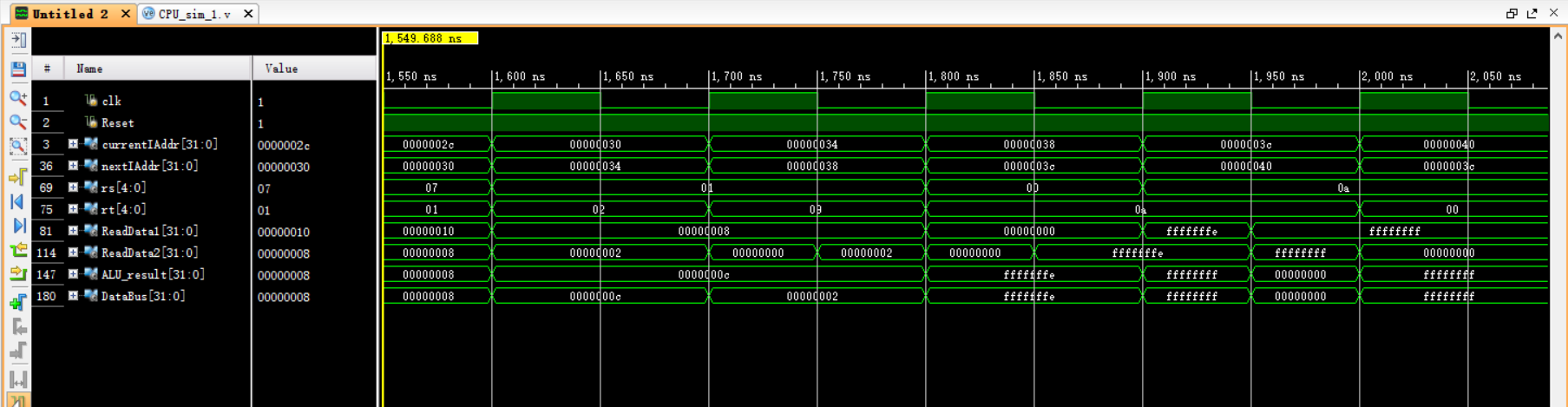
当前PC为0x00000040，PCSrc为00，顺序执行；

rs和rt分别为10号（十六进制的0a）和0号，值分别为-1（补码为0xffffffff）和0；

ALUOp为000，做加法运算，这是因为将运算数与0相加结果仍为本身，这样就可以利用ALU的sign输出。ALUSrcA和ALUSrcB信号都为0，ALU的操作数都来自寄存器堆。ALU将算出-1+0=-1，因此sign输出为1，bltz条件为真。因此PCSrc 为10，，表示nextIAddr为currentIAddr+4-2×4，下一步将跳转到分支去。由波形图可以看出，当前指令地址为0x00000040,下一条指令地址为0x0000003c，因此确实发生了跳转。

该指令不涉及写回寄存器堆，也不涉及数据存储器，因此RegWre、mRD、mWR 信号均为0。

另外2200~2300ns也是bltz指令，只不过这时rs的值为0，ALU的sign输出为0，因此PCSrc信号为00，将顺序执行，不做跳转。



（指令11~13图）

1. andi $11, $2, 2（ 2300~2400ns）

当前PC为0x00000044。PCSrc为00，顺序执行； rs和rt分别为2号和1号（十六进制的0b），执行后值分别为2和2；

ALUOp为100，做逻辑与运算，结果为2。ALUSrcA信号为0，输入rs寄存器的值；

ALUSrcB信号为1，输入零扩展（ExtSel信号为0）后的立即数；

RegDst信号为0，因此目的寄存器是rt；DBDataSrc信号为0，写回寄存器的值来自ALU的输出。

该指令不涉及数据存储器，因此mRD和mWR均为0。

1. j 0x00000050 （2400~2500ns）

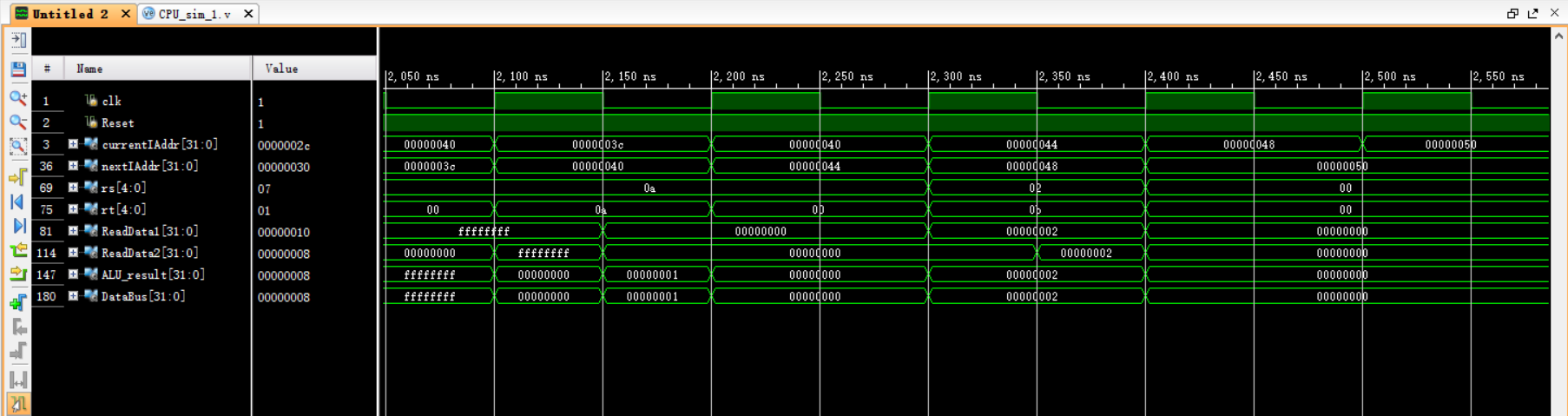
当前PC为0x00000048。PCSrc为10，为无条件跳转。跳转目标地址是32位的，其高4位和当前PC的高4位相同，低28位是指令的低26位左移两位的结果，这个地址就是0x00000050。从波形图也可以看到， nextIAddr为0x00000050。

J型指令不涉及寄存器堆、数据存储器、ALU，无关控制信号不做讨论。

1. halt （>=2500ns）

当前PC为0x00000050，PCSrc为11，因此 nextIAddr于currentIAddr相同（见实验内容 - 表1）。halt指令使得PC保持不变，从波形图可以看出，2500ns以后，所有信号、数值都保持不变了。

该指令不涉及其他操作，无关信号不做讨论。



（指令14~16图）

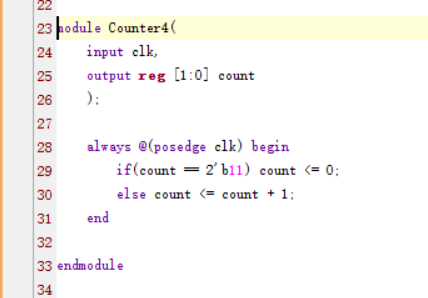
1. **在Basy3板上实现所设计的CPU**
2. **思想方法：**

根据实验要求，要在板上的四个七段数码管上显示当前PC、下一PC、rs 地址、rs数据、rt地址、rt数据、ALU结果以及DB总线的数据。而要显示的数据由CPU提供，因此需要先写出显示七段数码管的模块。

最后，将封装好的CPU顶层模块和新写好的显示模块再次封装，就可以综合、实现、烧板了。

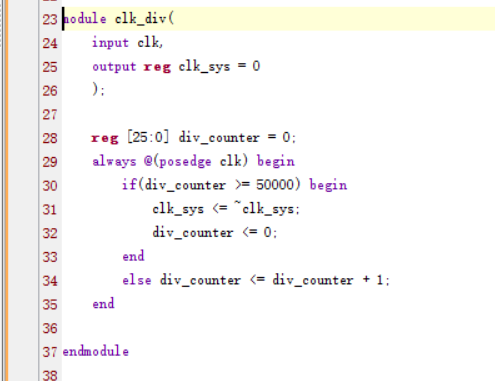
1. **操作方法：（步骤d~f在数电中已学习）**
2. 设置四位计数器

需要一个四位计数器来扫描四个七段数码管。

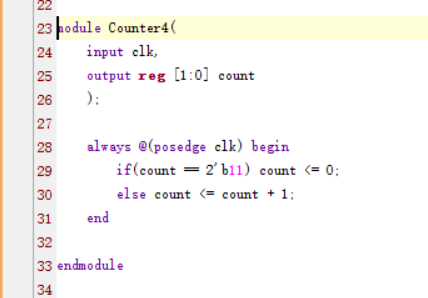


1. 时钟分频器

Basys3板载时钟频率为100MHz，可分频为1000Hz



1. 扫描显示时用到的四位计数器

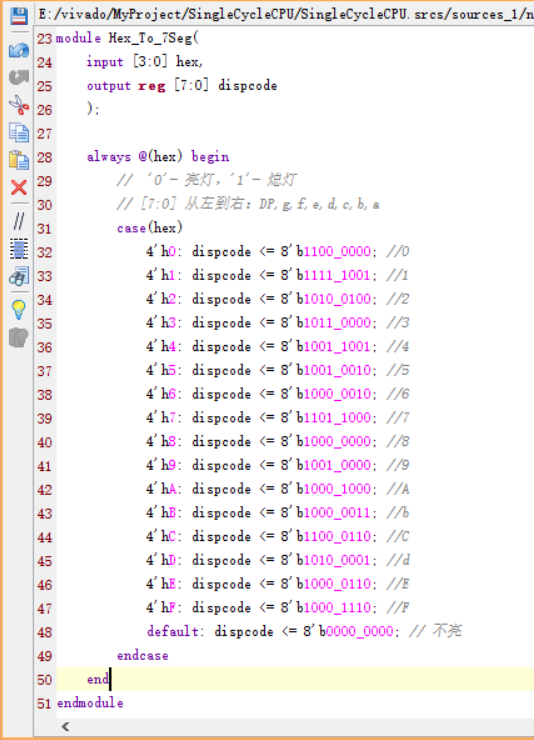


1. 十六进制到七段数码管的译码器

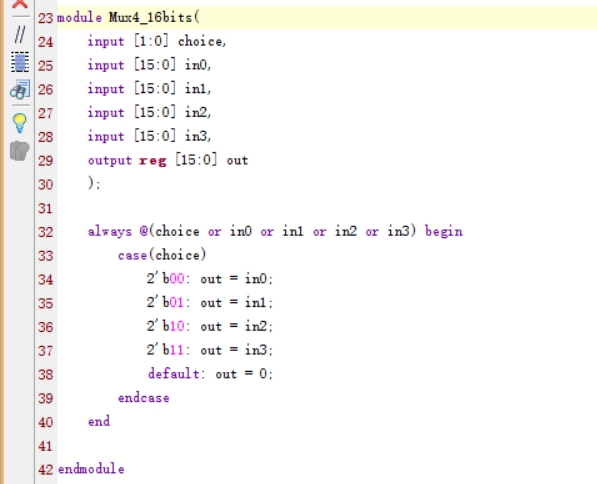
引脚说明：

hex，输入一个4位数字

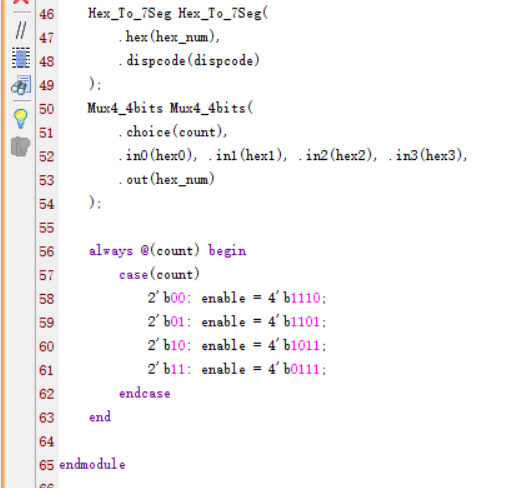
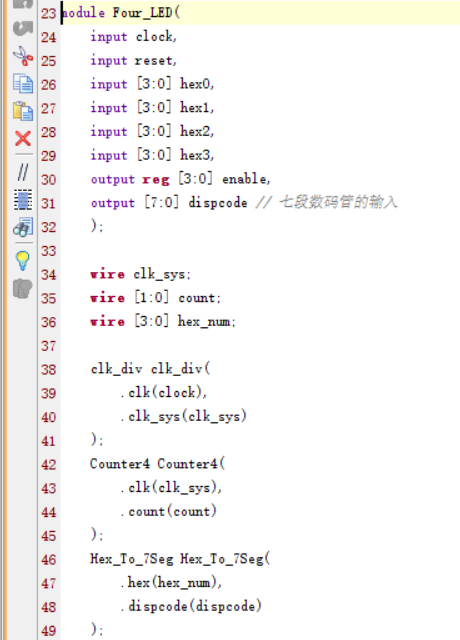
dispcode，输出七段数码管显示信号



1. 四位四选一数据选择器

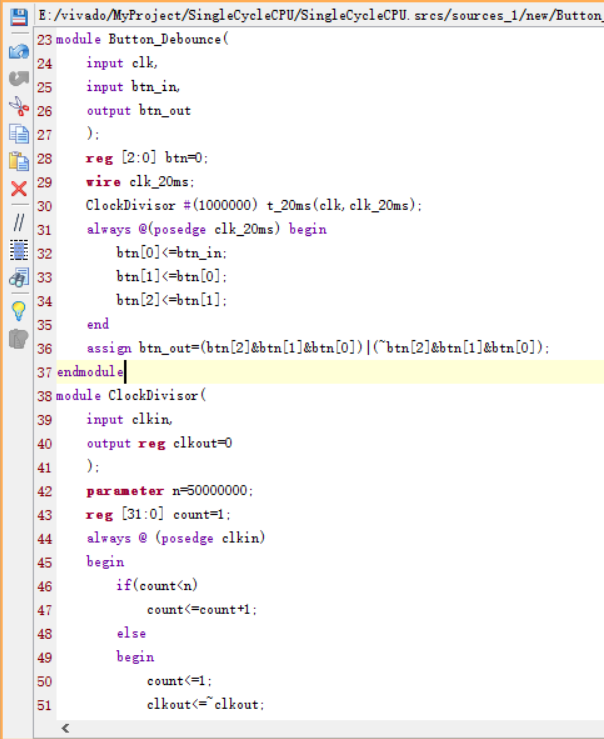


1. 扫描显示四个数字模块



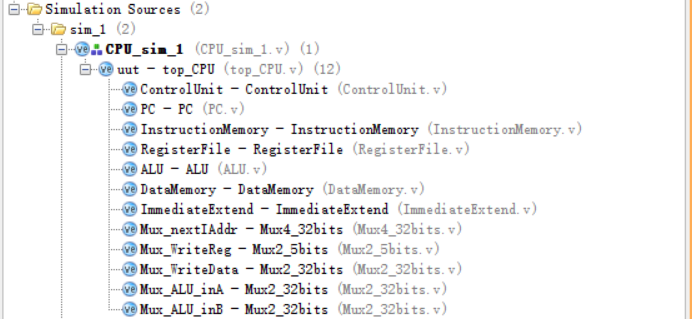
1. 按键消抖（重点难点——实验与现实需要的结合）

由于硬件原因，按板上的按钮时提供的电平变化可能不是单一的、稳定的上升沿或下降沿，因此需要对按钮进行消抖，其原理是按下按钮后延迟20ms才接受下一次按下。



1. 顶层连接模块

该模块封装top\_CPU（就是仿真用的那个顶层模块）和Four\_LED（用于显示四个数字的模块）。 具体对应引脚分配见“CPU烧板时，Basys3板的使用说明”。



1. **图片展示（前5条指令）**

addiu $1,$0,8

|  |  |  |  |
| --- | --- | --- | --- |
| 当前PC:下一PC | rs地址:rs数据 | rt地址:rt数据 | ALU结果:DB总线 |
|  |  |  |  |

ori $2,$0,2

|  |  |  |  |
| --- | --- | --- | --- |
| 当前PC:下一PC | rs地址:rs数据 | rt地址:rt数据 | ALU结果:DB总线 |
|  |  |  |  |

add $3,$2,$1

|  |  |  |  |
| --- | --- | --- | --- |
| 当前PC:下一PC | rs地址:rs数据 | rt地址:rt数据 | ALU结果:DB总线 |
|  |  |  |  |

sub $5,$3,$2

|  |  |  |  |
| --- | --- | --- | --- |
| 当前PC:下一PC | rs地址:rs数据 | rt地址:rt数据 | ALU结果:DB总线 |
|  |  |  |  |

and $4,$5,$2

|  |  |  |  |
| --- | --- | --- | --- |
| 当前PC:下一PC | rs地址:rs数据 | rt地址:rt数据 | ALU结果:DB总线 |
|  |  |  |  |

### 实验心得

1. 全局观念。单周期CPU设计这一项实验作业是一项比较大的工程项目，因此，在开始实验之前，必须要有全局观，要在脑海中构建出该实验的大致框架：熟悉CPU内部结构、熟悉单周期CPU的数据通路图、理解PC、指令存储器、寄存器堆、ALU、数据存储器的工作原理，还要清楚地了解实验内容中要求的每一条指令的组成，才能有效率地用Verilog 代码实现单周期CPU。

2. 对Verilog 语言的熟练度大大提高，对阻塞赋值与非阻塞赋值、过程赋值语句与持续赋值语句、 wire类型与reg类型等概念有了更清晰的认识，也学会了在合适的地方使用合适的表达式和语句（比如always中不能用assign赋值，模块需要记得endmodule，begin之后要有end）

3. 更加深入地理解了Verilog语言的“模块化”这一特性。本次设计的单周期CPU，其模块可以分为3个级别：最顶层的Basys3\_CPU，其中包含top\_CPU和Four\_LED两个模块，再往下则是各个底层模块。

4. 关于控制单元的触发条件。在设计控制单元（ControlUnit）时，其敏感信号表达式必须为：“opcode or zero or sign”，需要注意的是不能漏掉zero和sign。

5. =与<=的区别：<=是非阻塞式赋值，=是阻塞式赋值，当=这个赋值语句执行的时候是不允许有其他语句执行的，这就是阻塞的原因。而非阻塞赋值，例如a<=b，这个赋值语句执行的时候是不阻碍其他语句执行的。

6. 关于j指令。要清楚j指令的目标地址方式。测试程序段中的“j 0x00000050”，这个地址实际上就是目标地址了，但是要特别注意，这不是机器码中的26位地址。机器码中的26位地址是0x00000050右移2位的结果，即0x14，详见附录A表格的倒数第三行。

7. 烧板后，第一条指令没有写回的问题。首先明确“指令执行的结果总是在时钟下降沿保存到寄存器和存储器中，PC的改变是在时钟上升沿进行的，这样稳定性较好”。在Basys3板上，按钮松开为0，按下为1，因此按下按钮的动作对应一个时钟上升沿。