中山大學
SUN YAT-SEN UNIVERSITY

《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）	：	17 软件工程 2 班
学 生 姓 名	：	郑佳豪
学　　号	：	16305204
时　　间	：	2018　年　11　月　26　日

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。

三. 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。

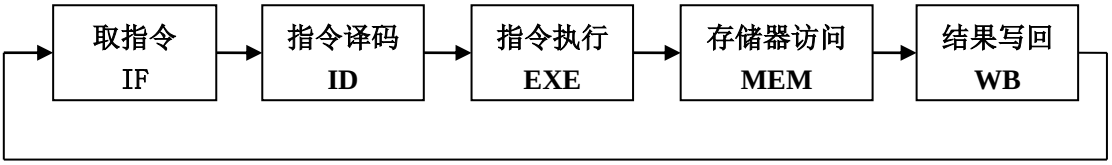


图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：

R 类型：

31	2625	2120	1615	1110	65	0
op	rs	rt	rd	sa	funct	
6 位	5 位	5 位	5 位	5 位	6 位	

I 类型：

31	2625	2120	1615	0
op	rs	rt	immediate	
6 位	5 位	5 位	16 位	

J 类型：

31	2625	0
op	address	
6 位	26 位	

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。

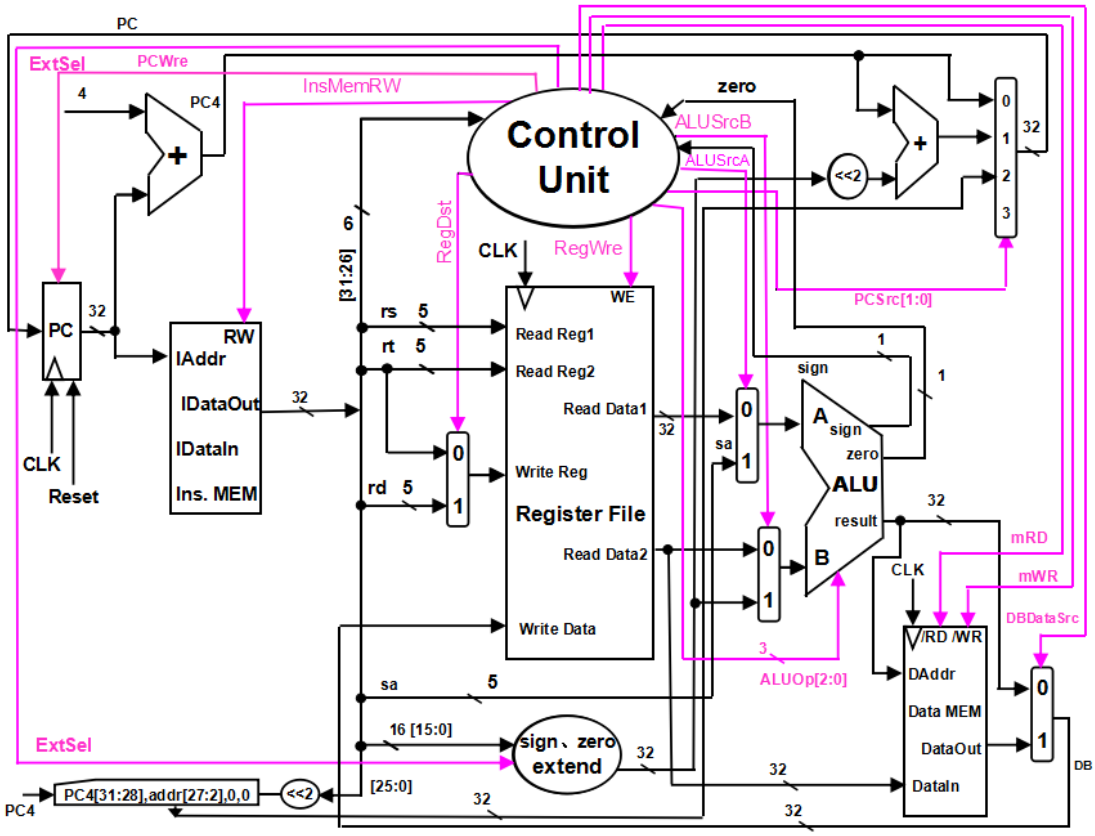


图 2　单周期 CPU 数据通路和控制线路图

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

相关部件及引脚说明：

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	

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

需要说明的是以上数据通路图是根据要实现的指令功能的要求画出来的，同时，还必须确定 ALU 的运算功能(当然，以上指令没有完全用到提供的 ALU 所有功能，但至少必须能实现以上指令功能操作)。从数据通路图上可以看出控制单元部分需要产生各种控制信号，当然，也有些信号必须要传送给控制单元。从指令功能要求和数据通路图的关系得出以上表 1，这样，从

表 1 可以看出各控制信号与相应指令之间的相互关系，根据这种关系就可以得出控制信号与指令之间的关系表（留给学生完成），再根据关系表可以写出各控制信号的逻辑表达式，这样控制单元部分就可实现了。

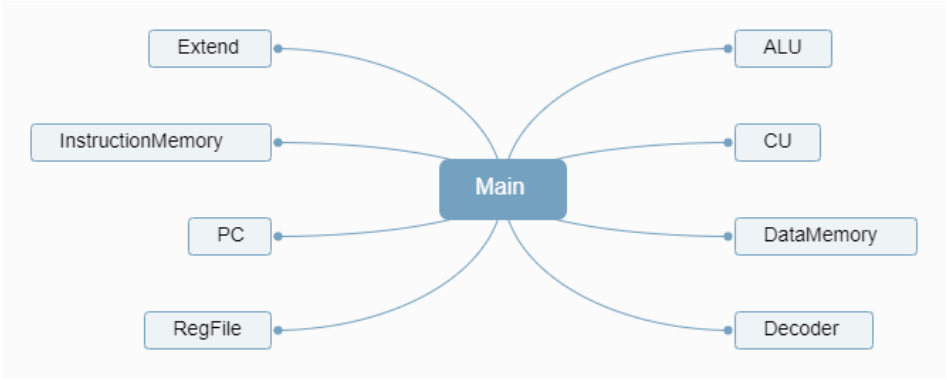
指令执行的结果总是在时钟下降沿保存到寄存器和存储器中，PC 的改变是在时钟上升沿进行的，这样稳定性较好。另外，值得注意的问题，设计时，用模块化的思想方法设计，关于 ALU 设计、存储器设计、寄存器组设计等等，也是必须认真考虑的问题。

四. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五. 实验过程与结果

1. 设计思想



参照上面提及到的数据通路，我将项目拆分为上图所示的几个部分。Main为顶层模块，用来将其余八大模块联系。

- CU：控制数据路线
- ALU：负责算术逻辑运算
- DataMemory：数据存储器
- Decoder：负责按照MIPS指令要求解析指令
- Extend：对立即数进行零扩展和符号扩展
- InstructionMemory：指令存储器
- PC：负责程序要处理指令的提供
- RegFile：寄存器堆

2. 模块设计

1) PC

该模块负责为处理器提供要处理的下一指令的地址。模块PCHelper根据正在执行的指令，计算下一条指令的地址，而模块PC则将PCHelper计算的地址在时钟上升沿或重置下降沿时作为下一周期执行的指令地址。

输入	**说明**
CLK	时钟信号
Reset	重置信号
PCWre	控制信号为 1，更新 PC
输出	**说明**
NewPC [31:0]	在下一周期执行的指令地址

```
`timescale 1ns / 1ps
`include "Constants.v"

module PC(
  input CLK,
  input Reset,
  input PCWre,
  input [31:0] NewPC,
  output reg [31:0] pc
);

  always@(posedge CLK or negedge Reset) begin
    if (Reset == 0)
      pc <= 0;
    else if (PCWre)
      pc <= NewPC;
  end

endmodule
```

```
`timescale 1ns / 1ps
`include "Constants.v"

module PCHelper(
  input [31:0] pc,
  input [15:0] Immediate,
  input [25:0] Address,
  input [1:0] PCSrc,
  output reg [31:0] NewPC
);

  initial begin
    NewPC = 0;
  end

  // Extend 16-bit immediate value to 32-bit immediate value
  wire [31:0] ExtImmediate = {{16{Immediate[15]}}, Immediate};

  always@(*) begin
    case(PCSrc)
      `PC_NEXT_INS: NewPC <= pc + 4;
      `PC_REL_JMP: NewPC <= pc + 4 + (ExtImmediate << 2);
      `PC_ABS_JMP: NewPC <= {pc[31:28], Address, 2'b00};
      `PC_HALT: NewPC <= pc;
    endcase
  end

endmodule
```

2) InstructionMemory

该模块根据当前的PC地址，得到对应地址的指令。

输入	说明
Iaddr [31:0]	执行读取的指令地址
输出	**说明**
IDataOut [31:0]	输出读取出来的指令

```
`timescale 1ns/1ps

module InstructionMemory(
  input [31:0] Iaddr,
  input [31:0] IDataIn,
  input RW,
  output reg [31:0] IDataOut
);

  reg [7:0] ROM [0:99];

  // Here, you should replace my file path as your own Instruction machine code's
  // location. And you must use the absolute path. Notice the slash characters
  // in the path.
  initial begin
    $readmemh ("E:/Users/Code/Single-Cycle-CPU/test/ROM.txt", ROM);
  end

  // MIPS Architecture uses the Big-Endian order.
  always@(*) begin
    IDataOut[31:24] = ROM[Iaddr];
    IDataOut[23:16] = ROM[Iaddr + 1];
    IDataOut[15:8] = ROM[Iaddr + 2];
    IDataOut[7:0] = ROM[Iaddr + 3];
  end

endmodule
```

3) RegFile

该模块为其余模块提供读取和写入寄存器的功能。

输入	说明
CLK	时钟信号
Reset	复位信号
WE	使能信号，为 0 时不写入，为 1 时写入数据
ReadReg1	读数据地址输入端
ReadReg2	读数据地址输入端
WriteReg	写数据地址输入端
WriteData	写入数据
输出	**说明**
ReadData1 [31:0]	读数据输出端
ReadData2 [31:0]	读数据输出端

```
`timescale 1ns / 1ps

module RegFile(
  input CLK,
  input Reset,
  input [4:0] ReadReg1,
  input [4:0] ReadReg2,
  input [4:0] WriteReg,
  input [31:0] WriteData,
  output [31:0] ReadData1,
  output [31:0] ReadData2,
  input WE
);

  integer index;
  reg [31:0] regFile[1:31];

  assign ReadData1 = ReadReg1 == 0 ? 0 : regFile[ReadReg1];
  assign ReadData2 = ReadReg2 == 0 ? 0 : regFile[ReadReg2];

  always@(negedge CLK or negedge Reset) begin
    if (Reset == 0) begin
      // The for statement using here would increase unneccessary units'usage.
      // However, to make the code more developer-friendly, I used this kind
      // of statement.
      for (index = 1; index < 32; index = index + 1) begin
        regFile[index] <= 0;
      end
    end
    else if (WE == 1 && WriteReg != 0) begin
      regFile[WriteReg] <= WriteData;
    end
  end

endmodule
```

4) DataMemory

该模块能提供读取、写入数据的功能。在时钟下降沿且RD为1时，表现为读取数据到DataOut；在时钟下降沿且WR为1时，表现为将DataIn写入到RAM。

输入	**说明**
CLK	时钟信号
RD	控制信号为 1，进行读操作
WR	控制信号为 1，进行写操作
Daddr [31:0]	要操作的内存地址
DataIn [31:0]	来自寄存器要写入的数据
输出	**说明**
DataOut [31:0]	输出读操作的结果

```
`timescale 1ns / 1ps

module DataMemory(
  input CLK,
  input [31:0] Daddr,
  input [31:0] DataIn,
  output [31:0] DataOut,
  input RD,
  input WR
);

  reg [7:0] RAM [0:60];

  assign DataOut[7:0] = RD ? RAM[Daddr + 3] : 8'bz;
  assign DataOut[15:8] = RD ? RAM[Daddr + 2] : 8'bz;
  assign DataOut[23:16] = RD ? RAM[Daddr + 1] : 8'bz;
  assign DataOut[31:24] = RD ? RAM[Daddr] : 8'bz;

  always@(negedge CLK) begin
    if (WR) begin
      RAM[Daddr] <= DataIn[31:24];
      RAM[Daddr + 1] <= DataIn[23:16];
      RAM[Daddr + 2] <= DataIn[15:8];
      RAM[Daddr + 3] <= DataIn[7:0];
    end
  end

endmodule
```

5) Extend

该模块提供将16位立即数扩展到32位的功能。

输入	**说明**
Immediate [15:0]	输入的 16 位立即数
ExtSel	控制信号为 1，进行符号扩展；为 0，进行零扩展
输出	**说明**
Immediate [32:0]	扩展后的 32 位立即数

```
`timescale 1ns / 1ps

module Extend(
  input [15:0] Immediate,
  input ExtSel,
  output [31:0] ExtImmediate
);

  assign ExtImmediate[15:0] = Immediate[15:0];
  assign ExtImmediate[31:16] = ExtSel == 1 && Immediate[15] == 1 ? 16'hFFFF : 16'h0000;

endmodule
```

6) ALU

该模块接收寄存器的数据和控制信号的输入，将运算后的结果输出。

输入	**说明**
ALUOp [2:0]	控制信号，用于运算功能的选择
ReadData1 [31:0]	来自寄存器 rs 的数据
ReadData2 [31:0]	来自寄存器 rt 的数据
输出	**说明**
Zero	当运算结果为 0，输出为 1，否则输出为 0
Sign	运算结果最高位，0 为正数，1 为负数
Result [31:0]	运算结果

```
`timescale 1ns / 1ps
`include "Constants.v"

module ALU(
  input [2:0] ALUOp,
  input [31:0] ReadData1,
  input [31:0] ReadData2,
  output reg [31:0] Result,
  output Zero,
  output Sign
);

  initial begin
    Result = 0;
  end

  assign Zero = Result == 0 ? 1 : 0;
  assign Sign = Result[31];

  always@(*) begin
    case(ALUOp)
      `ALU_ADD: Result = ReadData1 + ReadData2;
      `ALU_SUB: Result = ReadData1 - ReadData2;
      `ALU_SLL: Result = ReadData2 << ReadData1;
      `ALU_OR: Result = ReadData1 | ReadData2;
      `ALU_AND: Result = ReadData1 & ReadData2;
      `ALU_CMPU: Result = ReadData1 < ReadData2 ? 1 : 0;
      `ALU_CMPS: Result = ((ReadData1 < ReadData2) && (ReadData1[31] == ReadData2[31]))
                          || (ReadData1[31] == 1 && ReadData2[31] == 0)
                          ? 1 : 0;
      `ALU_XOR: Result = ReadData1 ^ ReadData2;
      default: Result = 8'h00000000;
    endcase
  end

endmodule
```

7) CU

该模块控制数据通路的路线，即控制其余各模块的数据通路。由于代码量较大和篇幅的限制，此处不贴出具体代码，具体可参照附件。

输入：Opcode [6:0]、Zero、Sign

输出：各控制信号

3. 模块测试

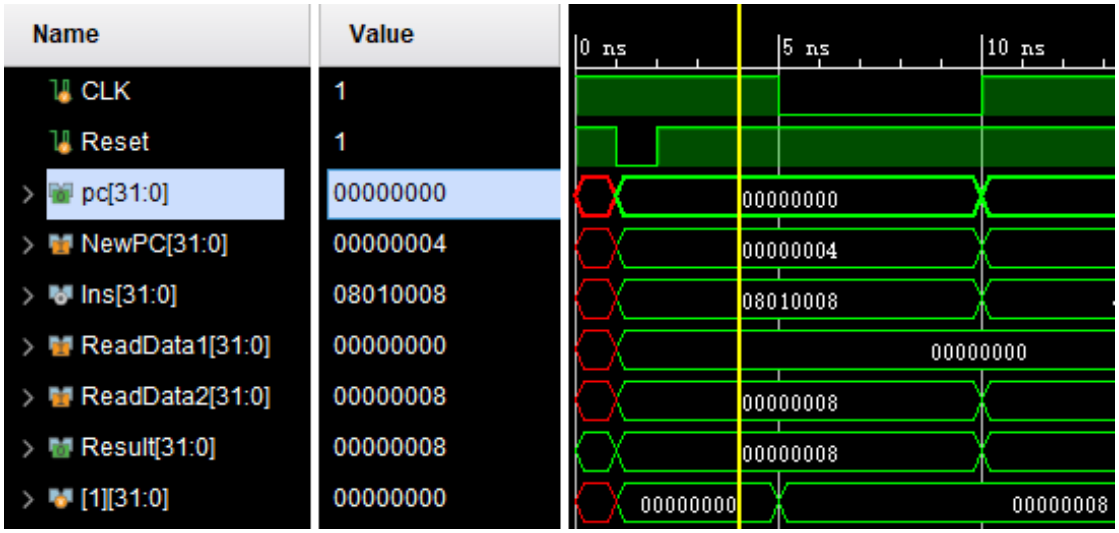
1) 测试代码

地址	汇编程序	答案	指令代码				
			op(6)	rs(5)	rt(5)	rd(5)/immediate	16 进制
0x00000000	addiu $1,$0,8	$1=8	000010	00000	00001	0000 0000 0000 1000	08010008
0x00000004	ori $2,$0,2	$2=2	010010	00000	00010	0000 0000 0000 0010	48020002
0x00000008	add $3,$2,$1	$3=10	000000	00010	00001	00011	00411800
0x0000000C	sub $5,$3,$2	$5=8	000001	00011	00010	00101	04622800
0x00000010	and $4,$5,$2	$4=0	010001	00101	00010	00100	44a22000
0x00000014	or $8,$4,$2	$8=2	010011	00100	00010	01000	4c824000
0x00000018	sll $8,$8,1	$8=4,8	011000	00000	01000	01000	60084040
0x0000001C	bne $8,$1,-2(不等转 18)	4,8	110001	01000	00001	1111 1111 1111 1110	c501fffe
0x00000020	slti $6,$2,4	$6=1	011100	00010	00110	0000 0000 0000 0100	70460004
0x00000024	slti $7,$6,0	$7=0	011100	00110	00111	0000 0000 0000 0000	70c70000
0x00000028	addiu $7,$7,8	$7=8,16	000010	00111	00111	0000 0000 0000 1000	08e70008
0x0000002C	beq $7,$1,-2(等则转 28)	8,16	110000	00111	00001	1111 1111 1111 1110	c0e1fffe
0x00000030	sw $2,4($1)	12 <- 2	100110	00001	00010	0000 0000 0000 0100	98220004
0x00000034	lw $9,4($1)	$9=2	100111	00001	01001	0000 0000 0000 0100	9c290004
0x00000038	addiu $10,$0,-2	$10=-2	000010	00000	01010	1111 1111 1111 1110	080afffe
0x0000003C	addiu $10,$10,1	$10=-1,0	000010	01010	01010	0000 0000 0000 0001	094a0001
0x00000040	bltz $10,-2(<0,转 3C)		110010	01010	00000	0000 0000 0000 0010	c940fffe
0x00000044	andi $11,$2,2	$11=2	010000	00010	01011	0000 0000 0000 0001	404b0002
0x00000048	J 0x0000050		111000	0000 0000 0000 0000 0000 0101 00			e000014
0x0000004C	Or $8,$4,$2	不执行	010011	00100	00010	01000	4c824000
0x00000050	halt		111111	00000	00000	0000 0000 0000 0000	fc000000

2) 功能验证

i.

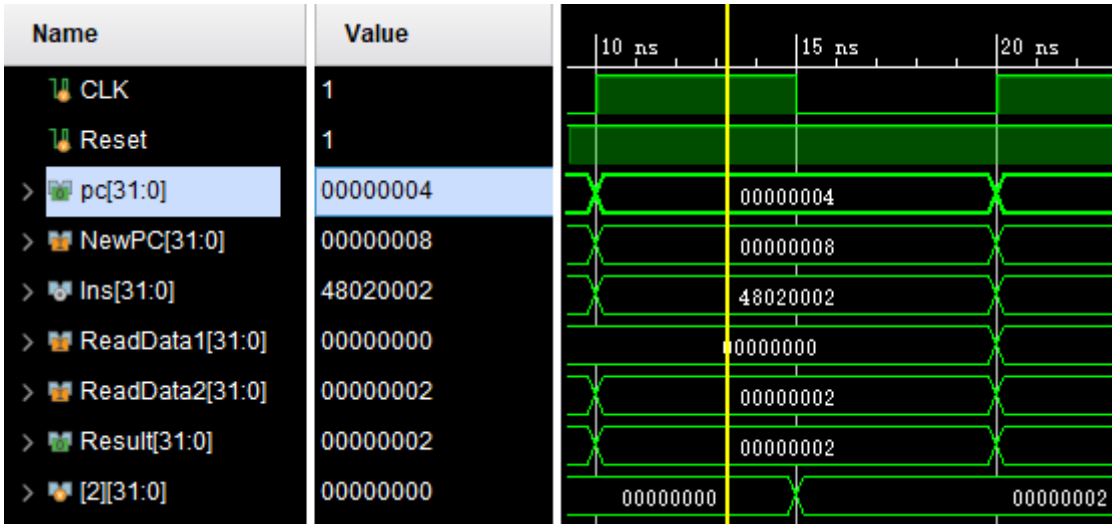
0x00000000	addiu $1,$0,8	$1=8	000010	00000	00001	0000 0000 0000 1000	08010008



PC为0x00000000，该指令表示为与无符号立即数的加法运算，ReadData1为寄存器$0的内容（0），ReadData2为立即数8，ALU运算结果为8。在时钟下降沿到达后，ALU运算结果会被写入寄存器$1中，其内容为8。

ii.

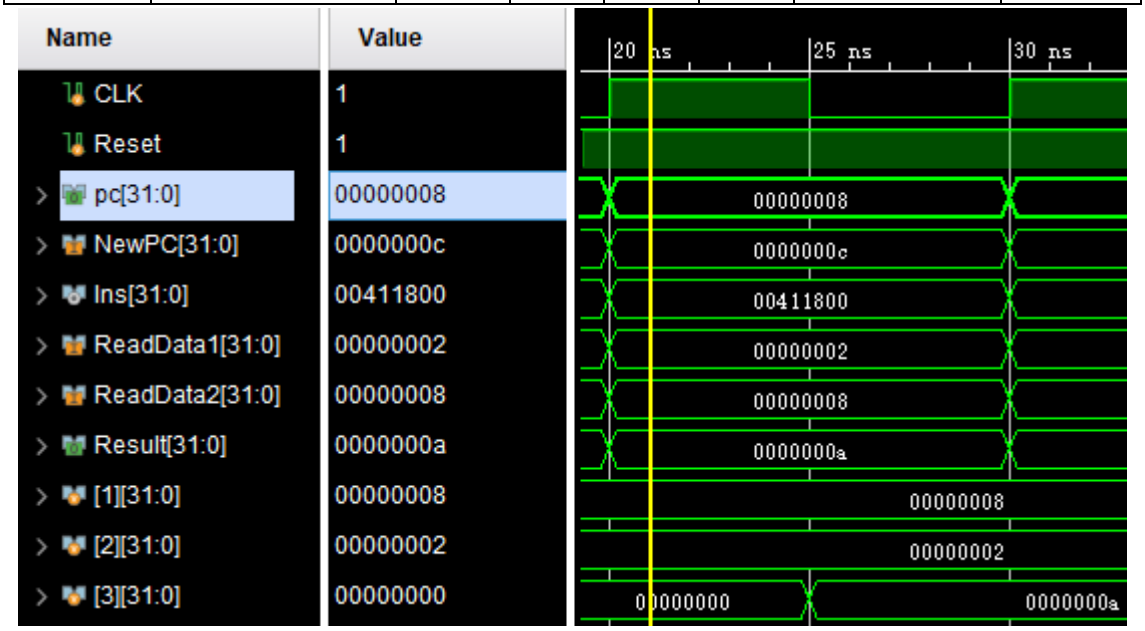
0x00000004	ori $2,$0,2	$2=2	010010	00000	00010	0000 0000 0000 0010	48020002



时钟上升沿到达后，PC为0x00000004，该指令表示与立即数的或运算，ReadData1为寄存器$0的内容（0），ReadData2为立即数2，ALU运算结果为2。在时钟下降沿到达后，ALU运算结果会被写入寄存器$2中，其内容为2。

iii.

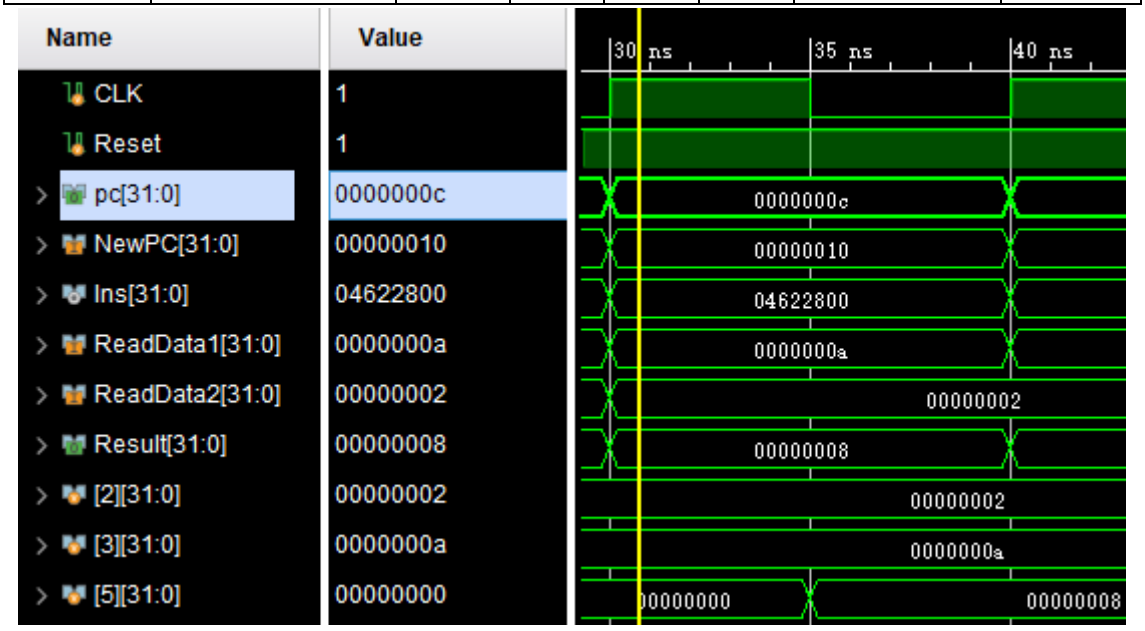
0x00000008	add $3,$2,$1	$3=10	000000	00010	00001	00011	00411800



时钟上升沿到达后，PC为0x00000008，该指令表示加法运算，ReadData1为寄存器$2的内容（2），ReadData2为寄存器$1的内容（8），ALU运算结果为10。在时钟下降沿到达后，ALU运算结果会被写入寄存器$3中，其内容为10。

iv.

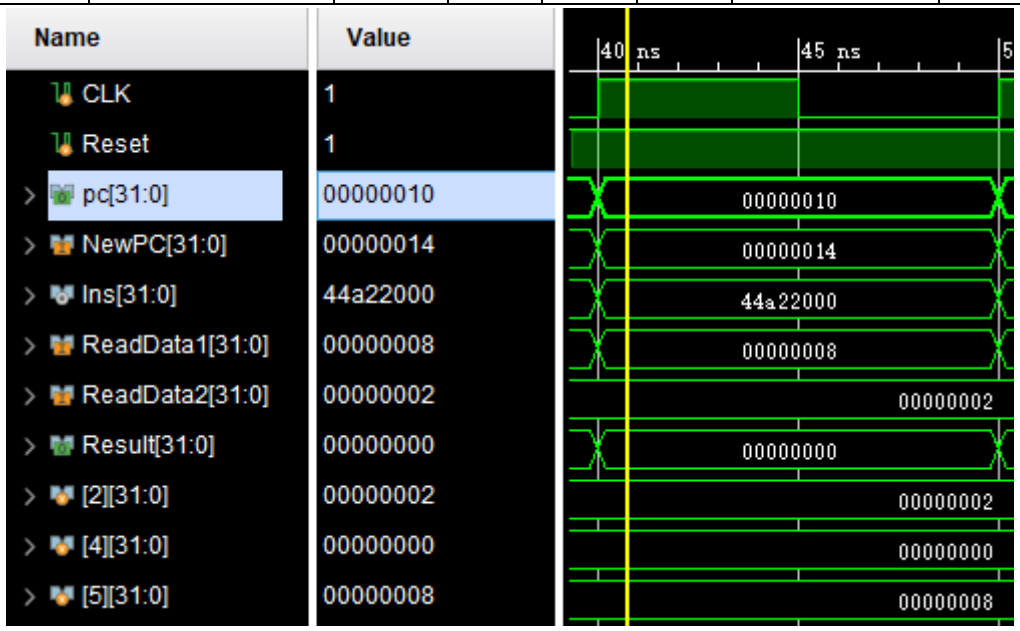
0x0000000C	sub $5,$3,$2	$5=8	000001	00011	00010	00101	04622800



时钟上升沿到达后，PC为0x0000000c，该指令表示减法运算，ReadData1为寄存器$3的内容（10），ReadData2为寄存器$2的内容（2），ALU运算结果为8。在时钟下降沿到达后，ALU运算结果会被写入寄存器$5中，其内容为8。

v.

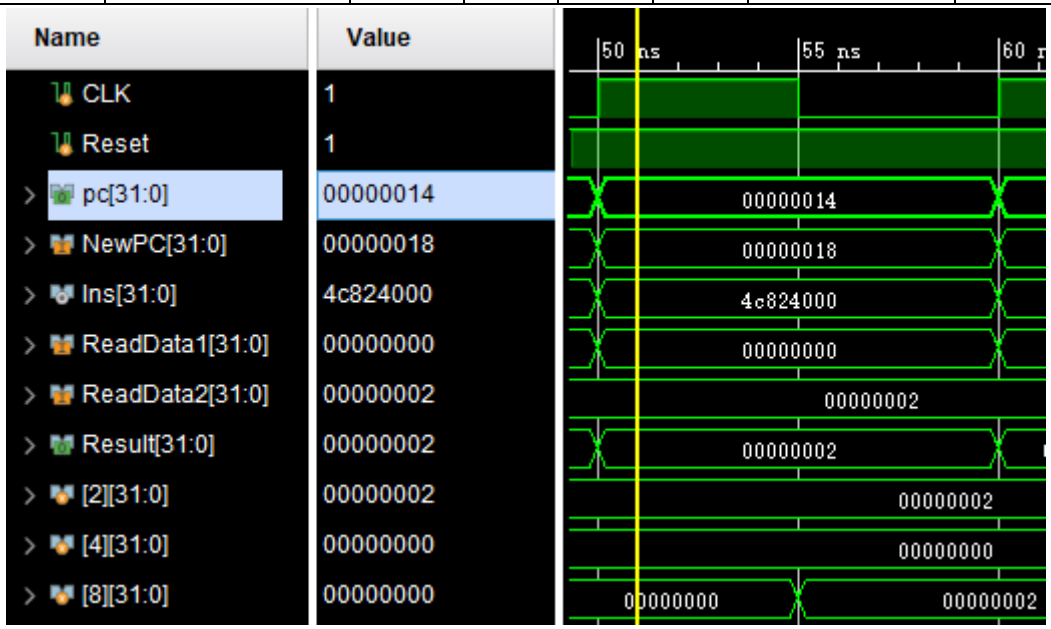
0x00000010	and $4,$5,$2	$4=0	010001	00101	00010	00100	44a22000



时钟上升沿到达后，PC为0x00000010，该指令表示与运算，ReadData1为寄存器$5的内容（8），ReadData2为寄存器$2的内容（2），ALU运算结果为0。在时钟下降沿到达后，ALU运算结果会被写入寄存器$4中，其内容为0。

vi.

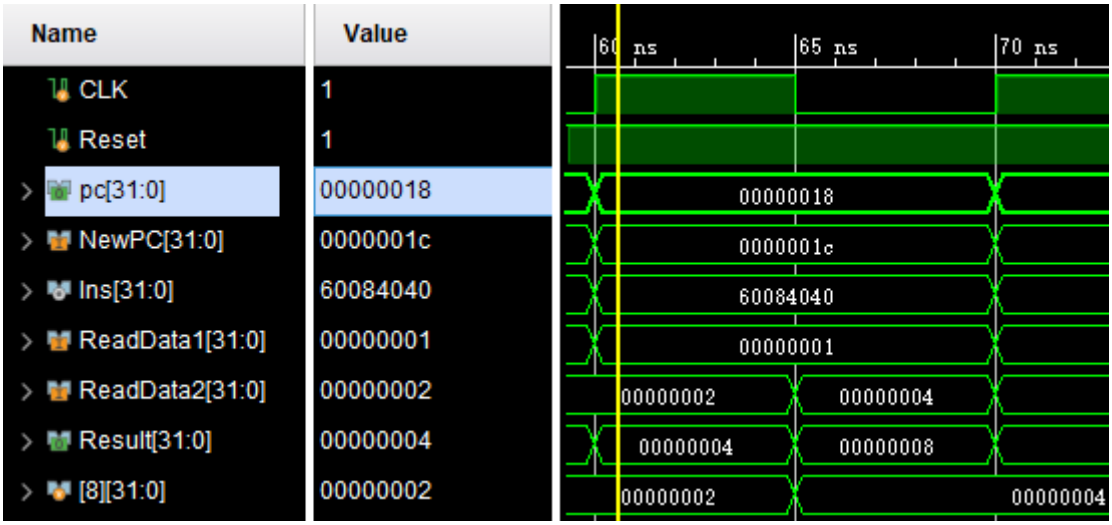
0x00000014	or $8,$4,$2	$8=2	010011	00100	00010	01000	4c824000



时钟上升沿到达后，PC为0x00000014，该指令表示或运算，ReadData1为寄存器$4的内容（0），ReadData2为寄存器$2的内容（2），ALU运算结果为2。在时钟下降沿到达后，ALU运算结果会被写入寄存器$8中，其内容为2。

vii.

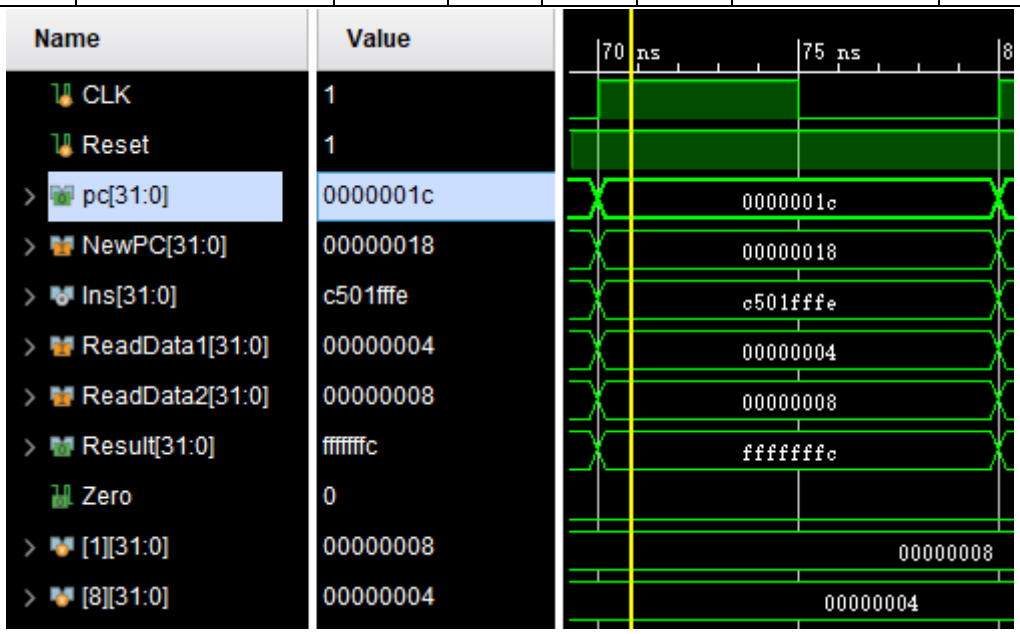
0x00000018	sll $8,$8,1	$8=4	011000	00000	01000	01000 | shamt=00001	60084040



时钟上升沿到达后，PC为0x00000018，该指令表示左移位运算，ReadData1为立即数1，ReadData2为寄存器$8的内容（2），ALU运算结果为4。在时钟下降沿到达后，ALU运算结果会被写入寄存器$8中，其内容为4。

viii.

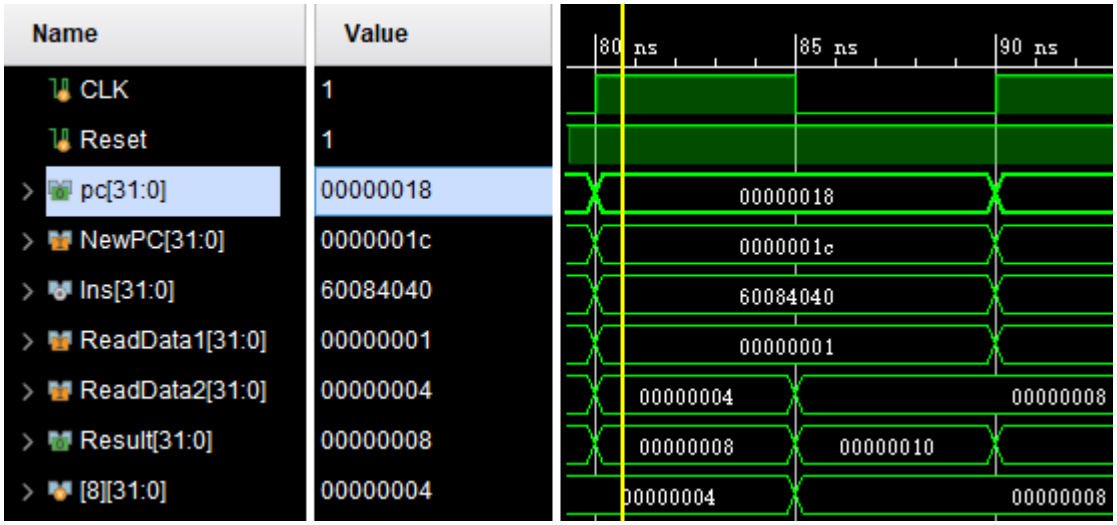
0x0000001C	bne $8,$1,-2(不等则转 18)	跳转至 18	110001	01000	00001	1111 1111 1111 1110	c501fffe



时钟上升沿到达后，PC为0x0000001c，该指令表示条件跳转，ReadData1为寄存器$8的内容（4），ReadData2为寄存器$8的内容（8），ALU运算结果Zero为0，表示$8和$1内容不相等，故下一条指令地址为nextPC=currentPC+immediate<<2，即跳转至0x00000018。

ix.

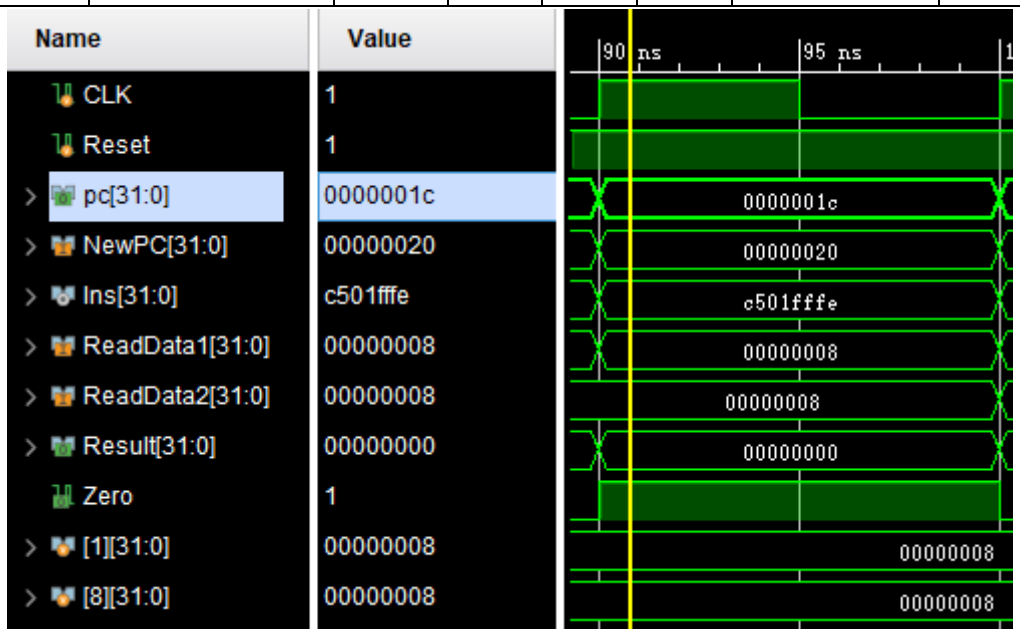
0x00000018	sll $8,$8,1	$8=8	011000	00000	01000	01000 | shamt=00001	60084040



时钟上升沿到达后，PC为0x00000018，该指令表示左移位运算，ReadData1为立即数1，ReadData2为寄存器$8的内容（4），ALU运算结果为8。在时钟下降沿到达后，ALU运算结果会被写入寄存器$8中，其内容为8。

x.

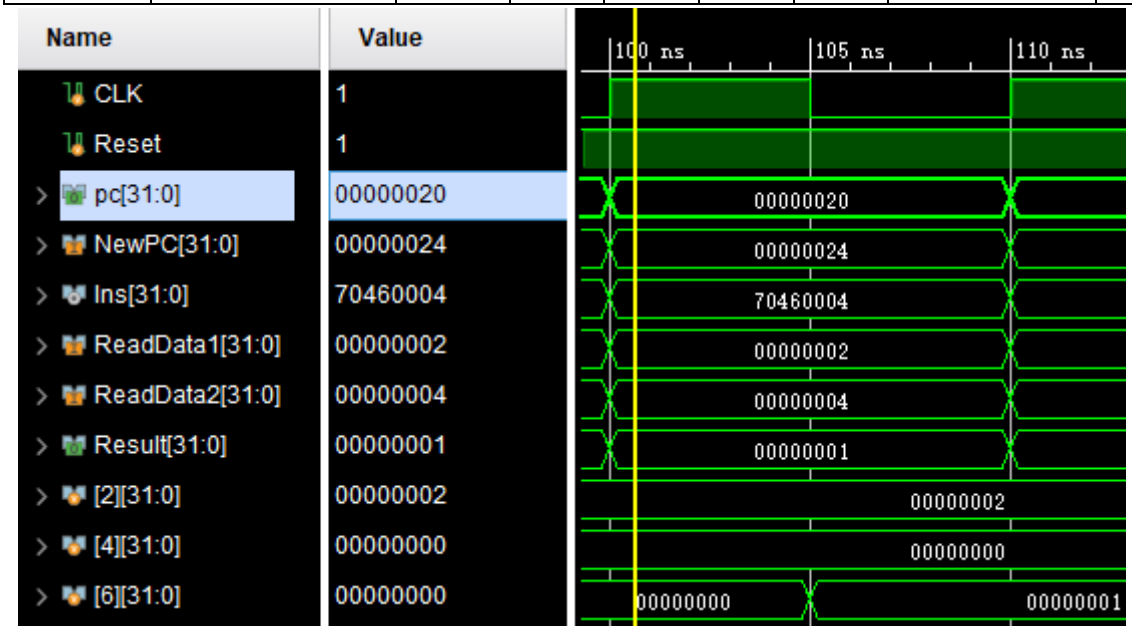
0x0000001C	bne $8,$1,-2	跳转至 20	110001	01000	00001	1111 1111 1111 1110	c501fffe



时钟上升沿到达后，PC为0x0000001c，该指令表示条件跳转，ReadData1为寄存器$8的内容（8），ReadData2为寄存器$8的内容（8），ALU运算结果Zero为1，表示$8和$1内容相等，故下一条指令地址为0x00000020。

xi.

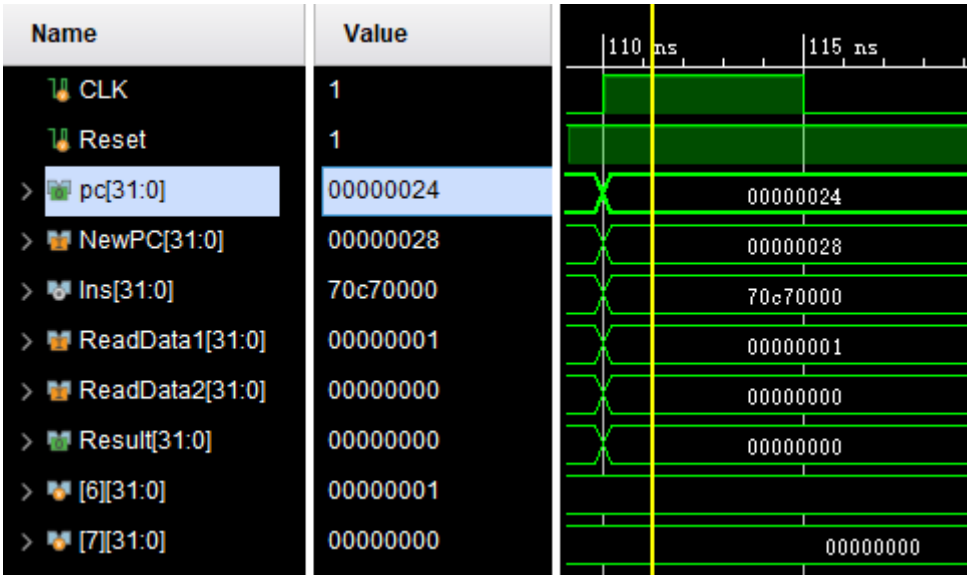
0x00000020	slti $6,$2,4	$6=1		011100	00010	00110	0000 0000 0000 0100	70460004



时钟上升沿到达后，PC为0x00000020，该指令表示与立即数的比较运算，ReadData1为寄存器$2的内容（2），ReadData2为立即数4，ALU运算结果为1，表示$2内容小于4。在时钟下降沿到达后，ALU运算结果会被写入寄存器$6中，其内容为1。

xii.

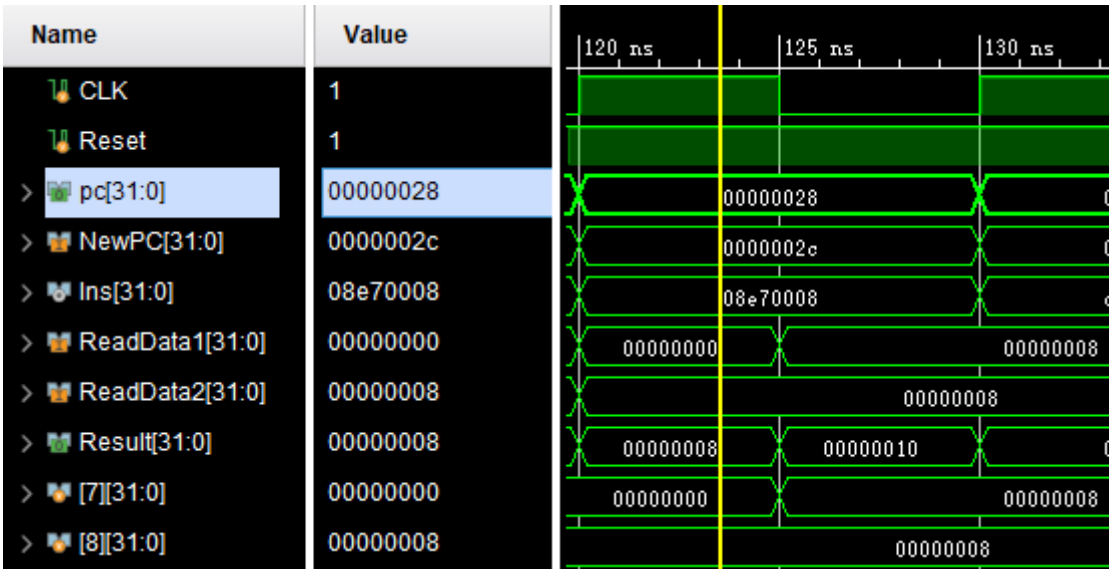
0x00000024	slti $7,$6,0	$7=0	011100	00110	00111	0000 0000 0000 0000	70c70000



时钟上升沿到达后，PC为0x00000024，该指令表示与立即数的比较运算，ReadData1为寄存器$6的内容（1），ReadData2为立即数0，ALU运算结果为0，表示$2内容大于或等于4。在时钟下降沿到达后，ALU运算结果会被写入寄存器$7中，其内容为0。

xiii.

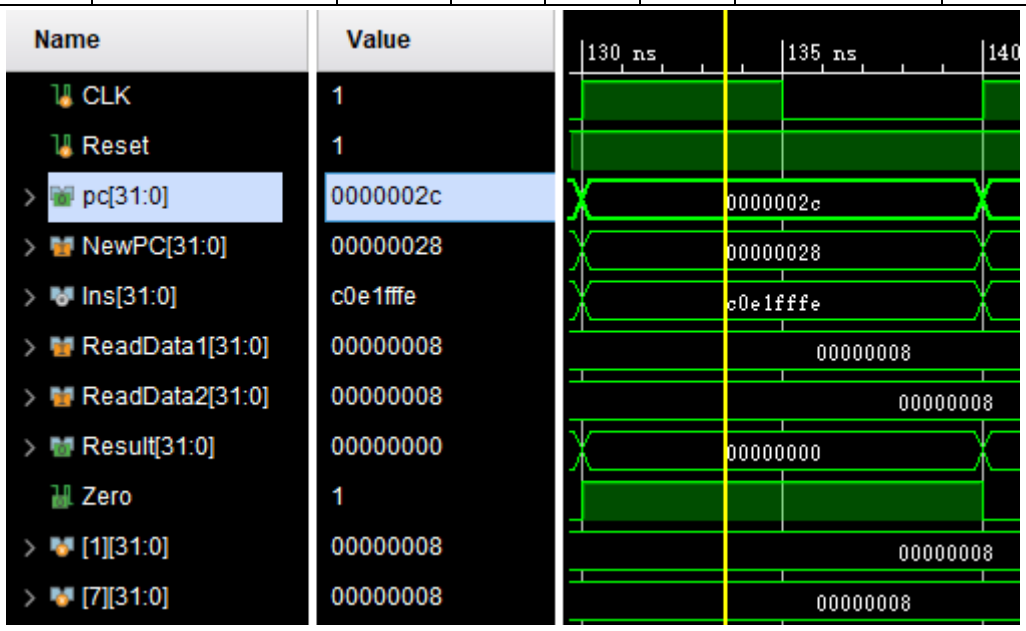
0x00000028	addiu $7,$7,8	$7=8	000010	00111	00111	0000 0000 0000 1000	08e70008



时钟上升沿到达后，PC为0x00000028，该指令表示与无符号立即数的加法运算，ReadData1为寄存器$7的内容（0），ReadData2为立即数8，ALU运算结果为8。在时钟下降沿到达后，ALU运算结果会被写入寄存器$7中，其内容为8。

xiv.

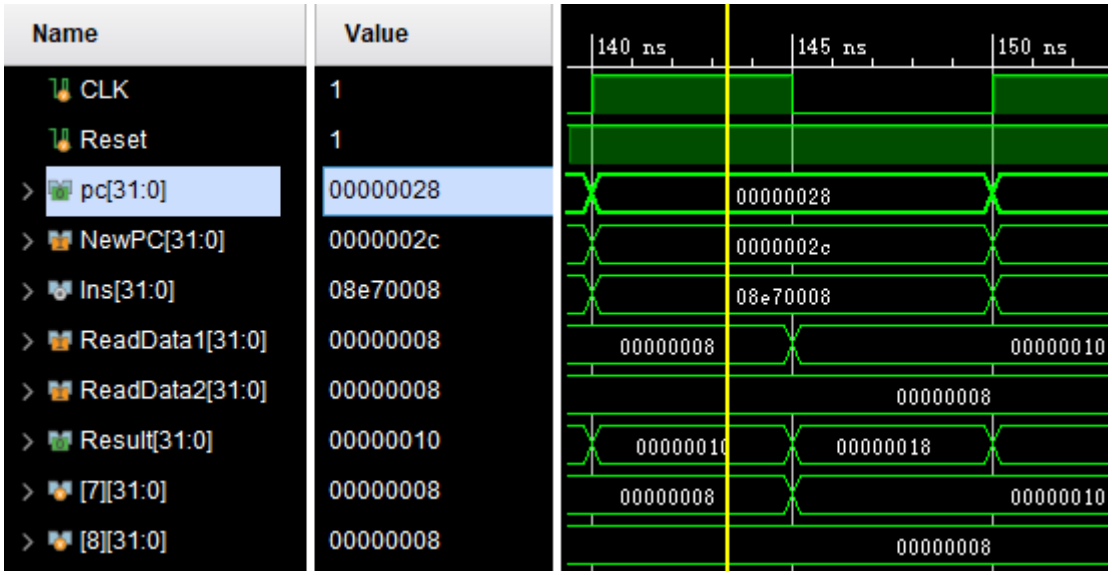
0x0000002C	beq $7,$1,-2(等则转 28)	跳转至 28	110000	00111	00001	1111 1111 1111 1110	c0e1fffe



时钟上升沿到达后，PC为0x0000002c，该指令表示条件跳转，ReadData1为寄存器$7的内容（8），ReadData2为寄存器$1的内容（8），ALU运算结果Zero为1，表示$7和$1内容相等，故下一条指令地址nextPC=currentPC+4，为0x00000028。

xv.

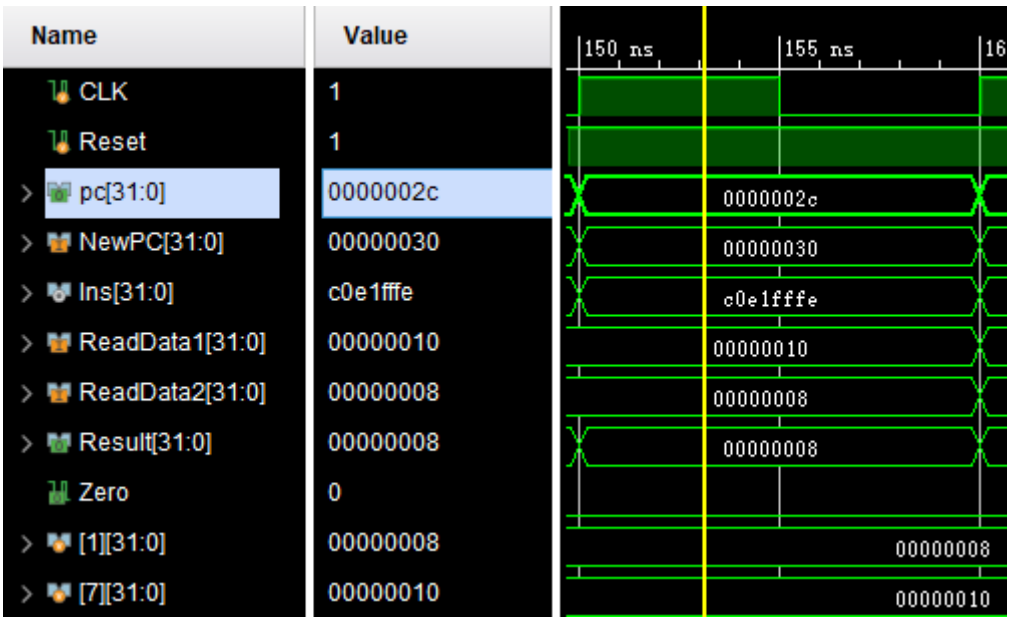
0x00000028	addiu $7,$7,8	$7=16	000010	00111	00111	0000 0000 0000 1000	08e70008



时钟上升沿到达后，PC为0x00000028，该指令表示与无符号立即数的加法运算，ReadData1为寄存器$7的内容（8），ReadData2为立即数8，ALU运算结果为8。在时钟下降沿到达后，ALU运算结果会被写入寄存器$7中，其内容为16。

xvi.

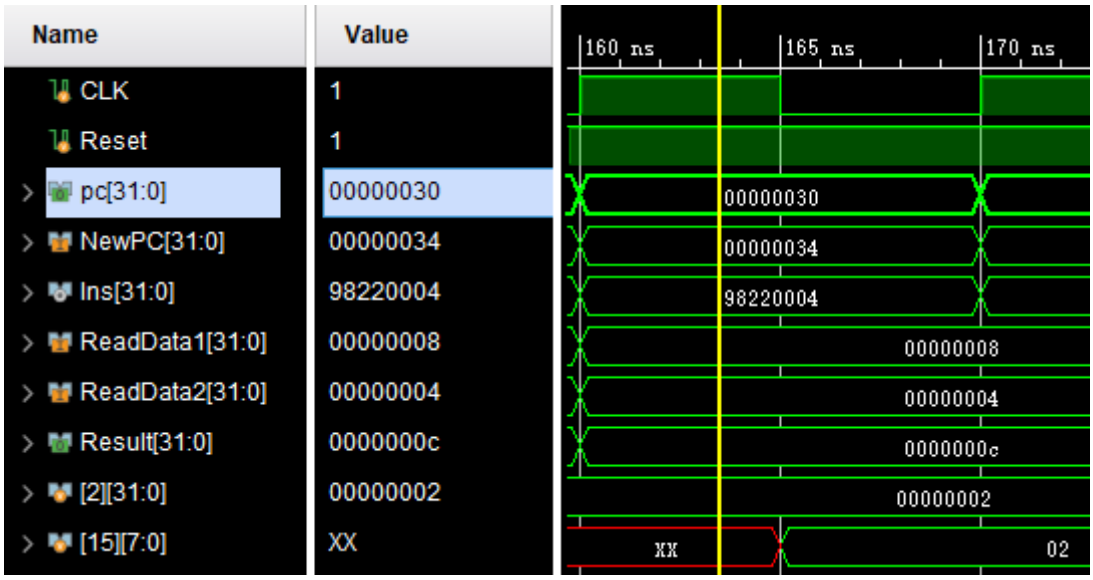
0x0000002C	beq $7,$1,-2	跳转至 30	110000	00111	00001	1111 1111 1111 1110	c0e1fffe



时钟上升沿到达后，PC为0x0000002c，该指令表示条件跳转，ReadData1为寄存器$7的内容（16），ReadData2为寄存器$1的内容（8），ALU运算结果Zero为0，表示$7和$1内容不相等，故下一条指令地址为nextPC=currentPC+4，为0x00000030。

xvii.

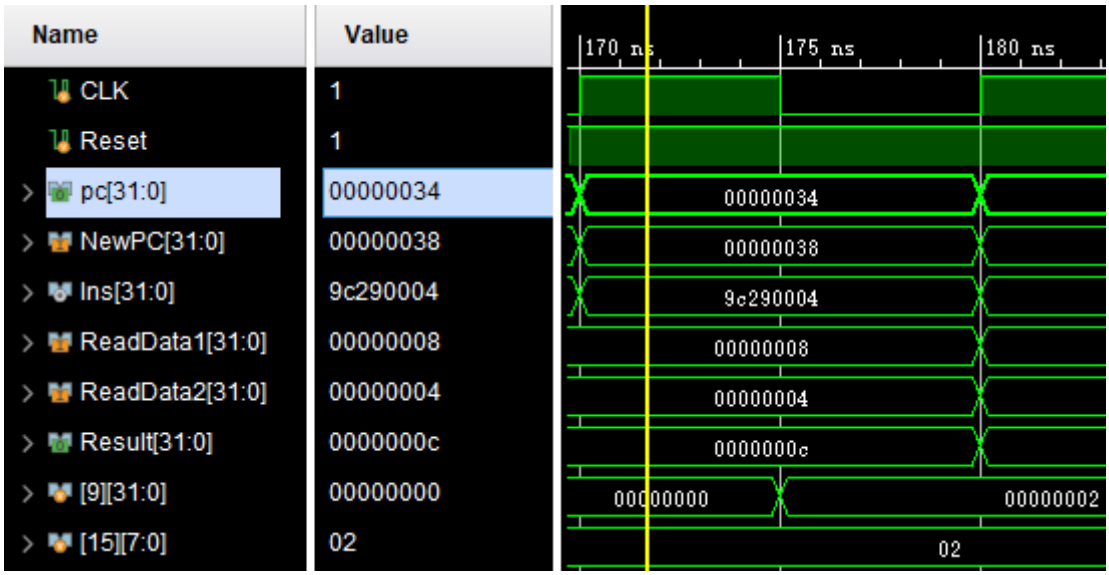
0x00000030	sw $2,4($1)	12 <- 2	100110	00001	00010	0000 0000 0000 0100	98229994



时钟上升沿到达后，PC为0x00000030，该指令表示写入寄存器，将寄存器$2的内容（2）写入到地址为4+($1)的内存单元，即12号内存单元。在时钟下降沿到达后，$2的内容会被写入到[15][7:0]，其内容为2。

xviii.

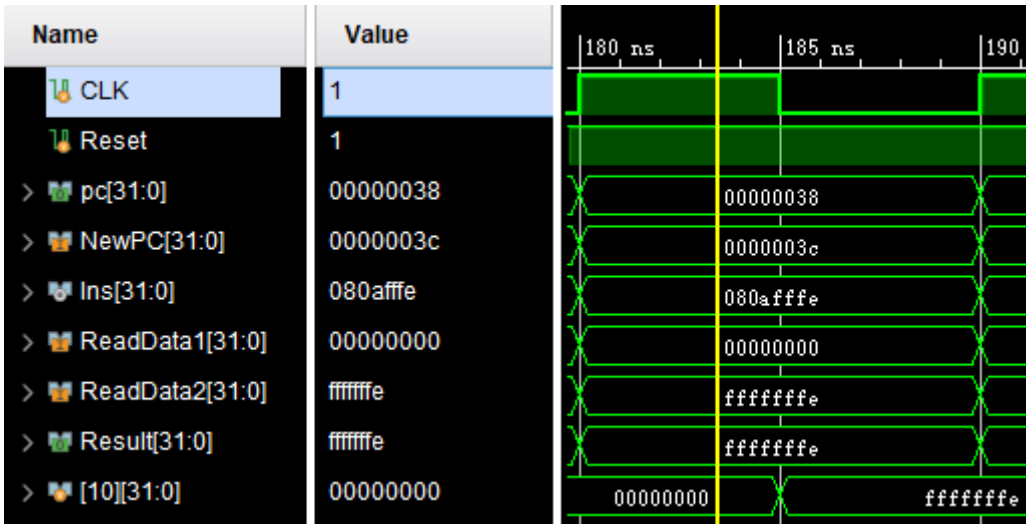
0x00000034	lw $9,4($1)	$9=2	100111	00001	01001	0000 0000 0000 0100	9c290004



时钟上升沿到达后，PC为0x00000034，该指令表示读取寄存器，读取地址为4+($1)的内存单元至寄存器$2。在时钟下降沿到达后，内存单元会被读取至寄存器$9，其内容为2。

xix.

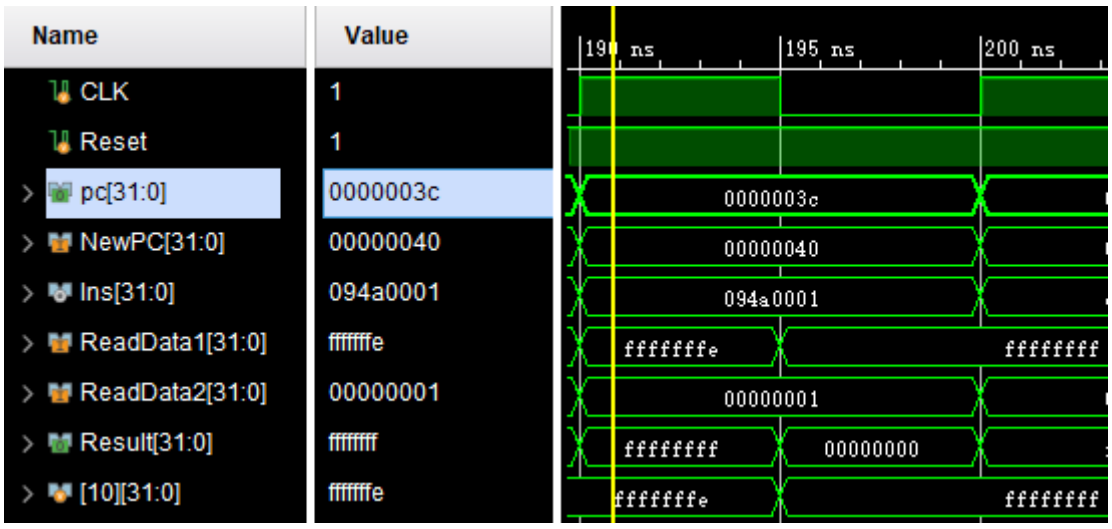
0x00000038	addiu $10,$0,-2	$10=-2	000010	00000	01010	1111 1111 1111 1110	080afffe



时钟上升沿到达后，PC为0x00000038，该指令表示与无符号立即数的加法运算，ReadData1为寄存器$0的内容（0），ReadData2为立即数-2，ALU运算结果为-2。在时钟下降沿到达后，ALU运算结果会被写入寄存器$10中，其内容为-2。

xx.

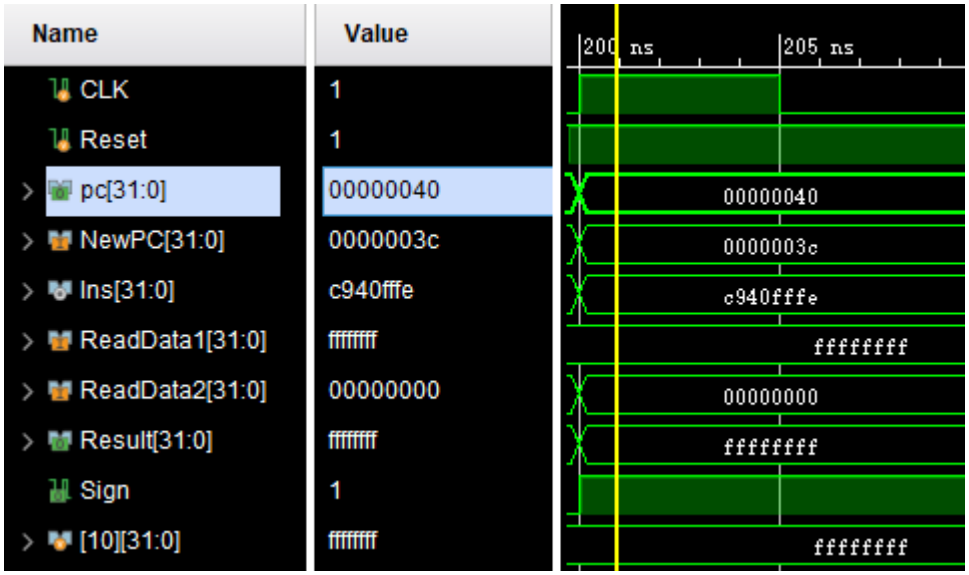
0x0000003C	addiu $10,$10,1	$10=-1	000010	01010	01010	0000 0000 0000 0001	094a0001



时钟上升沿到达后，PC为0x0000003c，该指令表示与无符号立即数的加法运算，ReadData1为寄存器$10的内容（-2），ReadData2为立即数1，ALU运算结果为-1。在时钟下降沿到达后，ALU运算结果会被写入寄存器$10中，其内容为-1。

xxi.

0x00000040	bltz $10,-2(<0,转 3C)	跳转至 3c	110010	01010	00000	0000 0000 0000 0010	c940fffe



时钟上升沿到达后，PC为0x00000040，该指令表示比较运算，ReadData1为寄存器$10的内容（-1），ReadData2为立即数0，Sign为1，表示$10内容小于0，故下一条指令地址为nextPC=currentPC+immediate<<2，即跳转至0x0000003c。

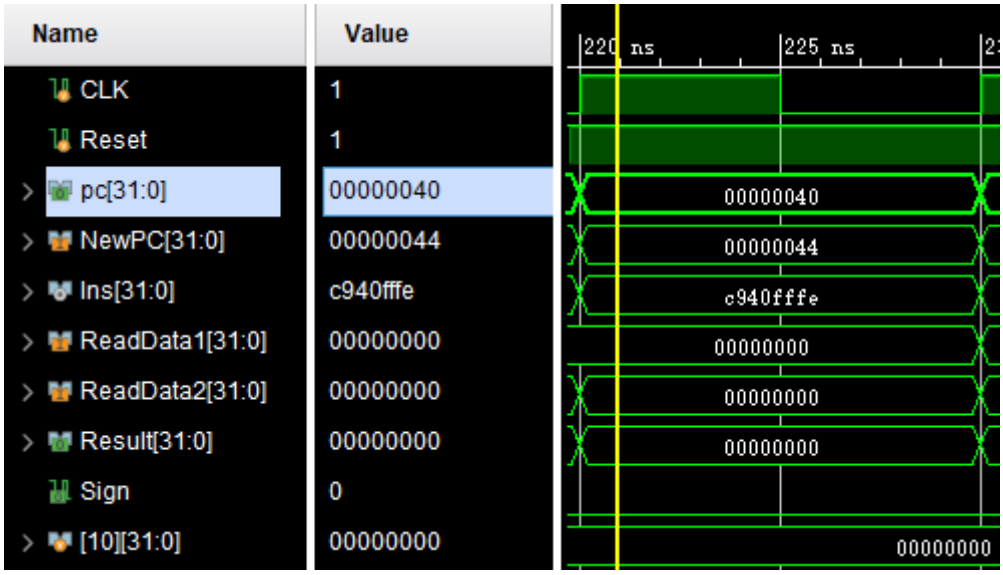
xxii.

0x0000003C	addiu $10,$10,1	$10=0	000010	01010	01010	0000 0000 0000 0001	094a0001

时钟上升沿到达后，PC为0x0000003c，该指令表示与无符号立即数的加法运算，ReadData1为寄存器$10的内容（-1），ReadData2为立即数1，ALU运算结果为0。在时钟下降沿到达后，ALU运算结果会被写入寄存器$10中，其内容为0。

xxiii.

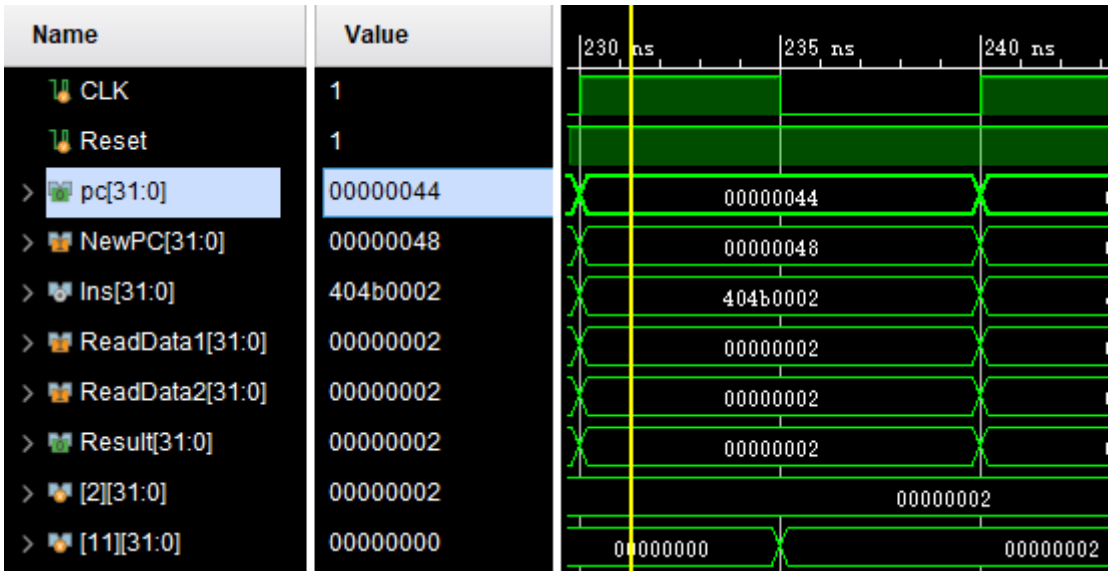
0x00000040	bltz $10,-2	跳转至 44	110010	01010	00000	0000 0000 0000 0010	c940fffe



时钟上升沿到达后，PC为0x00000040，该指令表示比较运算，ReadData1为寄存器$10的内容（0），ReadData2为立即数0，Sign为0，表示$10内容大于0，故下一条指令地址为nextPC=currentPC+4，即跳转至0x00000040。

xxiv.

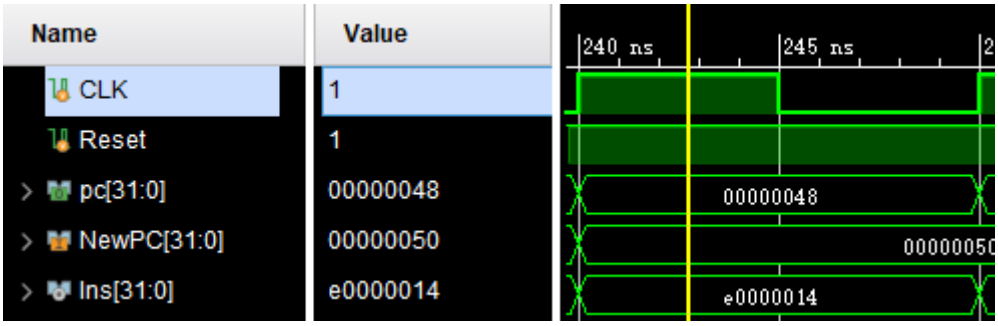
0x00000044	andi $11,$2,2	$11=2	010000	00010	01011	0000 0000 0000 0001	404b0002



时钟上升沿到达后，PC为0x00000044，该指令表示与立即数的与运算，ReadData1为寄存器$2的内容（2），ReadData2为立即数2，ALU运算结果为2。在时钟下降沿到达后，ALU运算结果会被写入寄存器$11中，其内容为2。

xxv.

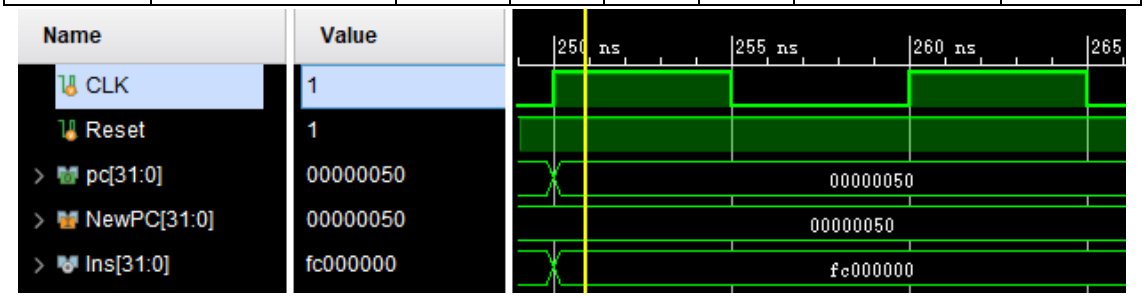
0x00000048	J 0x0000050		111000	0000 0000 0000 0000 0000 0101 00	e000014



时钟上升沿到达后，PC为0x00000048，该指令表示无条件跳转，故下一条指令地址为0x00000050。

xxvi.

0x00000050	halt		111111	00000	00000	0000 0000 0000 0000	fc000000



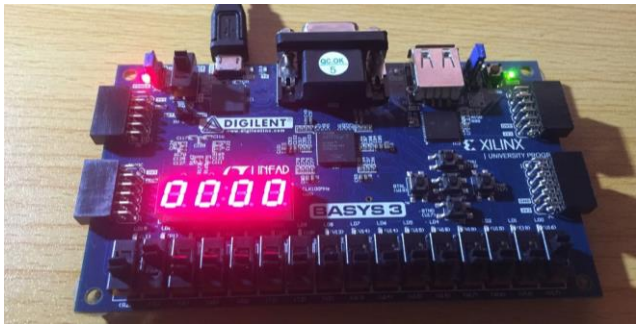
时钟上升沿到达后，PC为0x00000050，该指令表示停机，执行完PC保持原值。

六. Basys3 测试结果

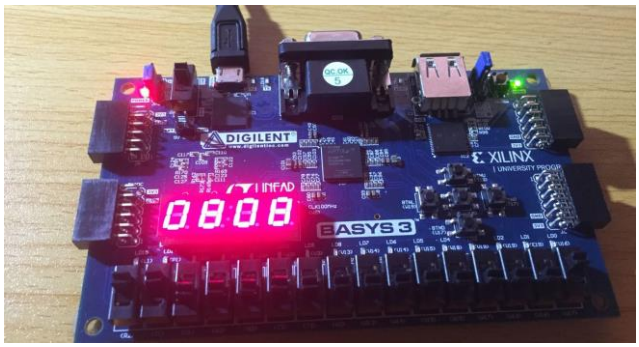
- 左边为当前PC地址00h，右边为下一周期执行的PC地址04h

- 左边为当前指令的rs寄存器编号01，右边为其内容00h

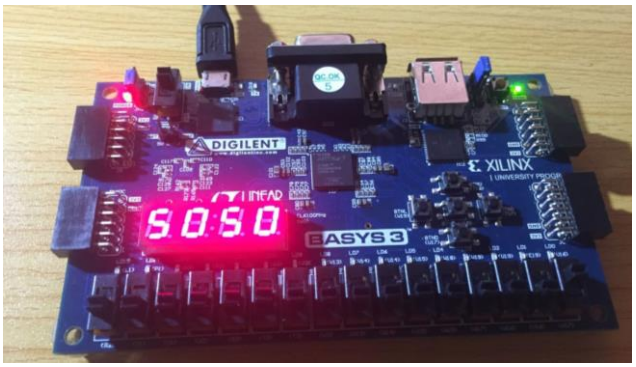
- 左边为当前指令的rt寄存器编号00，右边为其内容00h



- 左边为ALU运算结果08h，右边为DB总线数据08h



- 执行halt指令后，PC保持50h不变



七. 实验心得

总体来说，本次设计单周期CPU的实验给我带来的收获是巨大的。由于之前在数字电路设计课程中，曾独立完成基于MIPS指令子集的单周期CPU的设计与仿真，这给了我巨大的信心来成功完成本次实验。尽管之前有成功完成类似的实验，我依旧遇到了几个棘手的问题，并耗费较多的时间在这些问题的解答上。

- 指令文件路径需使用绝对路径，且路径分割符需使用“/”。一开始在导入指令文件时使用的是“\”路径分割符，然后软件一直提示静态文件错误，后来才发现如果使用“\”会造成转义，因此使用“/”。但仍未解决问题，后来在查询网上的资料后，才想起路径需要使用绝对路径，这个错误在之前的实验中也犯过，希望自己能够将正确的做法牢记于心。
- 烧录Basys3时需选择正确的型号。由于在一开始新建项目时没有注意型号的正确选择，出现了各项测试通过、但bin生成失败的情况。当时候第一反应是检查约束文件，但发现约束文件并无问题，与其他同学的进行比较也发现没有问题，后来我就将目光转移至项目设置，最后发现是型号选择错误，导致生成bin文件时的接口不对应问题。因为粗心，导致在这个并无太大技术含量的问题耽误了许久，是需要好好反思与调整自己，希望自己在未来的实验中能够不犯低级错误。
- 各模块的频率需正确设置。显示选择模块的频率是防抖按键输出的频率，扫描数码管的频率应该是分频后的频率。在烧录过程中，由于没有将时钟分频设置到1000Hz，没有利用到余晖效应，导致数码管无法正常显示。

通过这次单周期CPU设计实验，配合最近自学的操作系统知识，让我对计算机运作的底层原理有了更深入的理解。对于软件工程学生来说，虽然未来可能不做与硬件直接相关的工作，但如果能从硬件层次理解计算机运行的过程，这无疑是对软件开发有巨大帮助的。当然，由于本次实验是单周期CPU的设计，CPU设计存在许多完善的地方，如流水线机制，以及多周期CPU实现，希望自己能够在未来的时间里，能够继续学习。