数字逻辑电路实验报告

第十三次实验：简易计算机系统

张铭方
161220169
16 级计算机系 5 班
161220169@smail.nju.edu.cn

许致明
161180162
16 级计算机系 5 班
161180162@smail.nju.edu.cn

2017 年 12 月 29 日

第一节　实验目的

本实验的目标是在 Nexys 4 开发板上实现一个简易的计算机系统，能够运行简单的指令，包括循环、整数计算、函数调用、递归等。这些指令使用 RISC 方式编写，存储在开发板的存储器（ROM）中。开发板的另一部分存储器（RAM）用来保存程序运行中所需的数据。此外，在完成后，开发板还具有一定的输出功能，程序输入通过向 ROM 中初始化相应的机器码实现。

第二节　实验原理

一　计算机系统简介

计算机系统主要由 CPU 和外部设备组成。CPU 是系统中最重要的部分，它负责控制系统运行和信息处理。外部设备负责和外界进行交互，使得计算机的可以接受输入，产生相应的输出。本实验需要实现一个简化的计算机系统。下面对两种基本的系统结构做简要介绍。

第一种是冯・诺伊曼结构，这种结构被现代的大多数 CPU 所使用。在这种结构下，处理器使用同一个存储器，经过同一个总线传输，具有以下特点：

1. 结构上由运算器、控制器、存储器和输入/输出设备组成；
2. 存储器是按地址访问的，每个地址是唯一的；
3. 指令和数据都是以二进制形式存储的；
4. 指令按顺序执行，即一般指令按照存储顺序执行，程序的分支、循环由转移指令实现；
5. 以运算器为中心，在输入输出设备与控制器之间的数据传送都途径运算器。运算器、存储器、输入输出设备的操作以及它们之间的联系都由控制器集中控制。

第二种是哈佛结构，它使用两个独立的存储模块，分别存储指令和数据，并具有一条独立的地址总线和一条独立的数据总线，具有以下特点：

1. 每个存储块都不允许指令和数据并存，以便实现并行处理；
2. 利用公共地址总线访问两个存储模块（程序存储模块 ROM 和输出存储模块 RAM），公用数据总线则被用来完成程序存储模块或数据存储模块与 CPU 之间的数据传输；
3. 地址总线和数据总线由程序存储器和数据存储器分时共用。

数字信号处理一般需要较大的运算量和较高的运算速度，为了提高数据吞吐量，在数字信号处理中大多采用哈佛结构。本实验所构建的计算机系统就采用了哈佛结构。

二 RISC 架构简介

2.2.1 RISC 架构的基本特征

RISC，即 Reduced Instruction Set Computer，是精简指令集计算机的简称，与它相对的是 CISC（Complex Instruction Set Computer）。RISC 架构主要具有以下特点：

1. 只包含一些使用频率较高的指令，并用这些指令的组合来实现较为复杂指令的功能；
2. 指令长度固定，指令格式、寻址方式比 CICS 少；
3. 只有加载、存储两条指令需要访问内存，其他指令都是在寄存器和寄存器或寄存器和立即数之间进行操作；
4. CPU 中包含多个通用寄存器，执行指令过程中的数据均暂存在寄存器中，提高指令的执行速度；
5. 常常采用流水线技术，这样大部分指令可以在一个时钟周期内完成。还可以采用超标量和超流水线技术，使指令平均执行时间小于一个时钟周期；
6. 控制器采用组合逻辑的控制方式，不使用微程序控制的方式。

CPU 是计算机中的核心部件。RISC 架构中的 CPU 进行信息处理时，主要进行如下两个步骤：

1. 将数据和指令（二进制串）读入到计算机的存储器中；
2. 从第一条开始，按顺序执行程序，直至停机，结束运行。

这一过程还可以用伪代码表示如下：

```
    CPU-EXECUTE
1   while TRUE
2       FETCH instruction Instr[PC]
3       DECODE Instr[PC]
4       EXECUTE Instr[PC]
5       PC = PC + 1
```

为了实现这些操作，CPU 至少需要具有以下功能：

1. 取指令：当程序已经在存储器（ROM）时，首先根据程序入口地址取出一条指令。需要 CPU 能发出正确的地址信息和产生控制读取存储器的信号；
2. 指令译码：这一操作即需要分析出此二进制串的意义，获得它指示的操作内容，并且产生相应的操作控制命令；

3. 执行指令：根据指令译码得到的结果，产生相应的操作控制信号序列，控制运算器、存储器、输入输出设备的动作，完成这条指令的功能。其中包含对运算结果的处理（如设置标志位）以及下一条指令的地址的形成（如根据跳转指令更改 PC 的值）。

总而言之，CPU 做为计算机系统的核心，主要的任务就是取出指令，解释指令，然后根据得到的结果执行相应的指令。在这些过程中，可能涉及到存储器（例如内存、内部寄存器）的读取、写入，PC 内容更改（自增 1 或按条件跳转），以及和外部设备的数据交换（接收外设传来的输入信号，将输出信号发送给外设）。同时，CPU 也需要产生相应的控制信号，使得其他部件能正确的工作。

2.2.2 RISC 架构下 CPU 的基本构成

RISC CPU 主要包括三方面的功能：数据存储、数据运算、时序控制。与此对应的就是三方面的硬件设备：寄存器和内存、运算器 ALU、控制器。其基本结构如下面的图 1 所示：

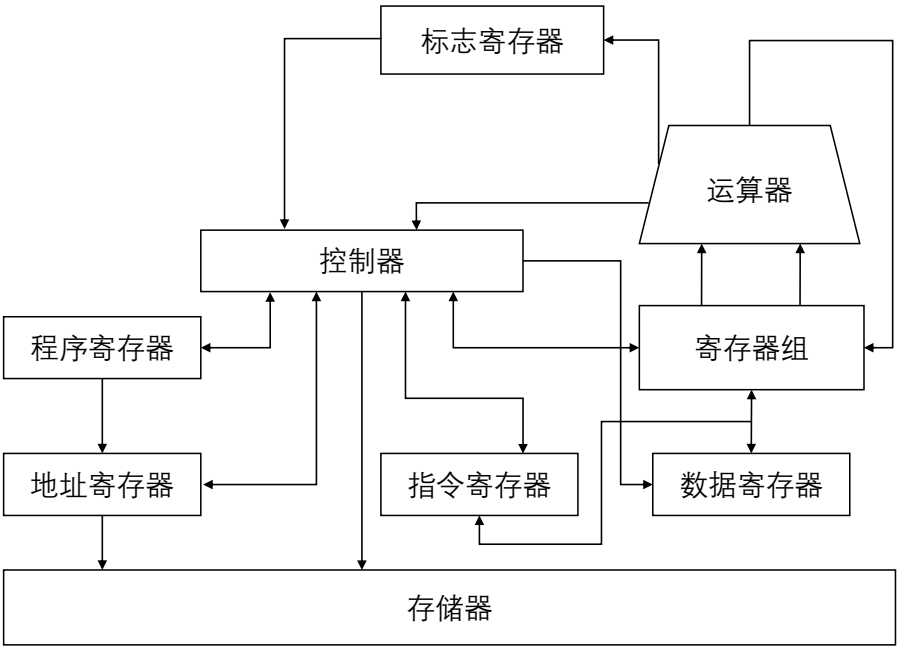


图 1: RISC 架构 CPU 的基本结构

寄存器用于存放指令和数据，在此 CPU 设计中，包含了较多寄存器，这符合 RISC 架构的主要思想，即将指令执行过程中的各种数据存入到不同的寄存器中，减少访问速度较慢的内存的次数，从

而提高处理器的运行速度。图中的箭头指明了数据在这条通路上的传输方向。通过多条通路，指令和数据能够独立传输、互不干扰，这样也能提高 CPU 的运行速度。

三　CPU 的指令系统设计

在此次实验中，我们完成了一个 16 位的 RISC 架构指令集，共包含 32 条 5 个二进制位的定长指令，具有算术运算、存储读写、输入输出和逻辑控制这些基本功能。具体如表 1 所示：

表 1: 指令系统概要

汇编指令格式	操作数	功能描述	类型
ADC %SR, %DR	00000	R[DR]+R[SR]+CF→R[DR]	算数逻辑
SBB %SR, %DR	00001	R[DR]-R[SR]-CF→R[DR]	
MUL %SR, %DR	00010	R[DR]*R[SR]→R[DR]	
DIV %SR, %DR	00011	R[DR]/D[SR]→D[DR]（无符号）	
ADDI $IMM, %DR	00100	R[DR]+$IMM→R[DR]（寄存器与立即数相加）	
CMP %SR, %DR	00101	R[DR]-R[SR]（只改变标志寄存器，若相等，则 ZF=1）	
AND %SR, %DR	00110	R[DR]&R[SR]→R[DR]（按位与）	
OR %SR, %DR	00111	R[DR]|R[SR]→R[DR]（按位或）	
NOT %DR	01000	~R[DR]→R[DR]（按位取反）	
XOR %SR, %DR	01001	R[DR]^R[SR]→R[DR]（按位异或）	
TEST %SR, %DR	01010	R[DR]^R[SR]（根据结果改变标志寄存器 ZF）	
SHL %DR	01011	R[DR]<<1→R[DR]（逻辑或算数左移，最高位入 CF）	
SHR %DR	01100	R[DR]>>1→R[DR]（逻辑右移，高位补 0，最低位入 CF）	
SAR %DR	01101	R[DR]>>1→R[DR]（算术右移，高位补符）	
IN PORT, %DR	01110	[PORT]→R[DR]	I/O
OUT %SR, PORT	01111	R[SR]→PORT	
MOV %SR, %DR	01110	R[SR]→R[DR]	数据传送
MOVIL $IMM, %DR	01111	IMM→R[DR][0..7]（8 位立即数移入寄存器高 8 位）	
MOVIH $IMM, %DR	10000	IMM→R[DR][8..15]（8 位立即数移入寄存器低 8 位）	
LOAD %SR, %DR	10001	M[R[SR]]→R[DR]	访存
STORE %SR, %DR	10010	R[SR]→M[R[DR]]	
PUSH %SR	10101	R[SR]→M[R[SP]], R[SP]-1→R[SP]（SR 入栈）	栈操作
POP %DR	10110	M[R[SP]]→R[DR], R[SP]+1→R[SP]（DR 出栈）	
JMP OFFSET	10111	PC+OFFSET→PC，无条件跳转指令，OFFSET 有符号	控制转移
JMPC	11000	当 CF=1 时进行跳转	
JNC	11001	当 CF≠1 时进行跳转	
JMPZ	11010	当 ZF=1 时进行跳转	
JNZ	11011	当 ZF≠1 时进行跳转	
CALL	11100		
RET	11101		
NOP	11110	空操作，PC+1→PC	处理器
HALT	11111	停机	

我们设计出的这个指令规定数据字长也为 16 位。指令格式固定，操作数寻址方式共有 3 种。大多数均为寄存器寻址和立即数寻址，只有 LOAD/STORE 两条指令涉及到内存寻址。指令的格式有图 2 所示的三种，对其中的符号做如下说明：

1. *Opcode* 为操作数，即指令表中对应的第二栏；
2. $\%SR$ 是源操作数所在的寄存器，$\%DR$ 是目的操作数所在的寄存器；
3. $\$IMM$ 是立即数，受指令长度限制，所有的立即数均为 8 位二进制数；
4. *Offset* 是偏移量，为有符号数，符号由 *Offset* 域的最高位决定，0 为负，1 为正，由此给定后 8 位立即数的符号。

指令类型一：

$Opcode$	$\%SR$	$\%DR$	$Undefined$

指令类型二:

$Opcode$	$\%DR$	$\$IMM$

指令类型三:

$Opcode$	$Offset$

图 2: 指令格式类型

第三节　实验器材和环境

一　硬件

Nexys 4 开发板

二　环境

运行在 Windows 10 下的 Vivado 2016.4

第四节　实验设计思路

一　RISC CPU 的数据通路图

我们设计的 CPU 数据通路如下图所示：

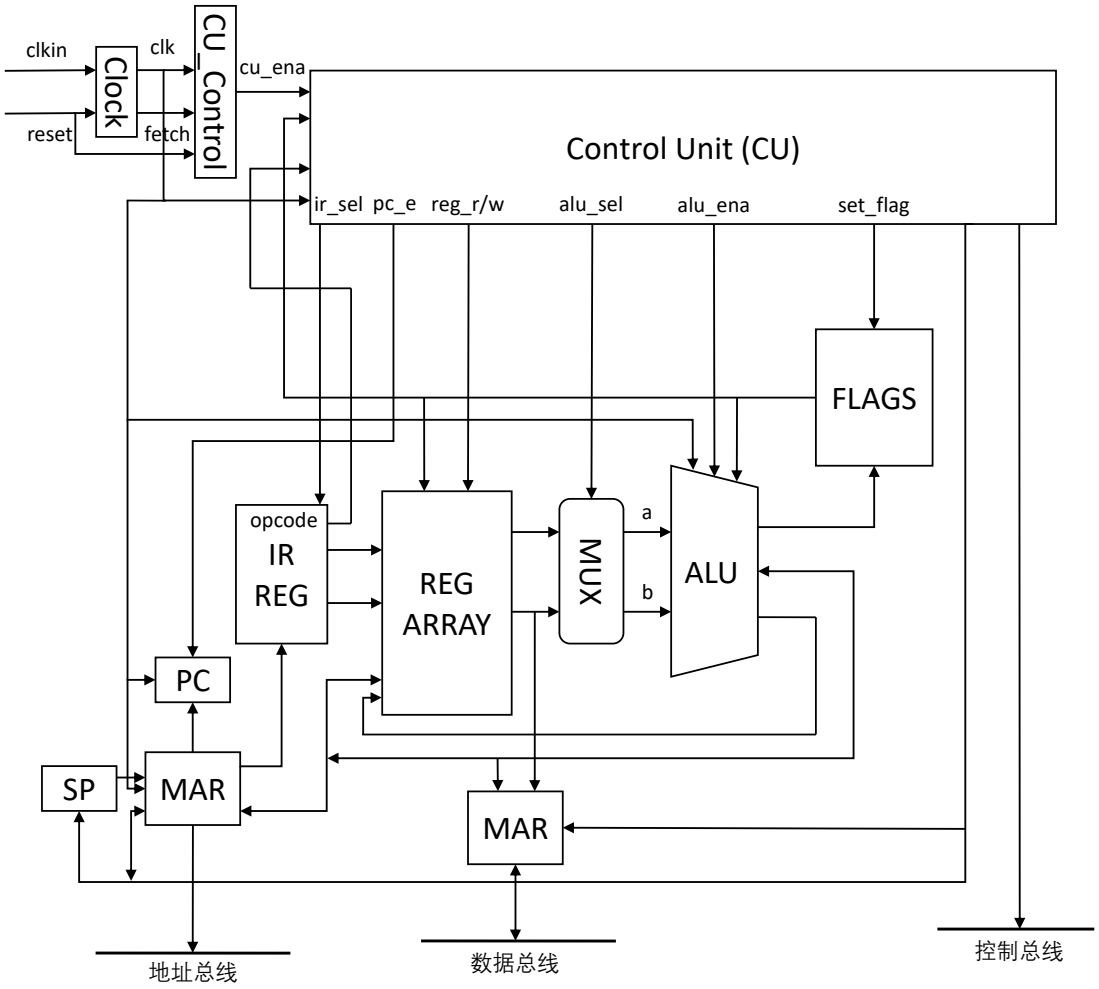


图 3: CPU 的数据通路

我们在这个 CPU 中设计了 8 个 16 位的按照 16 位可读写的通用寄存器组（图中 REG_BANK）。除此之外，还包含指令寄存器（图中 IR REG），地址寄存器（图中 MAR），数据寄存器（图中 MDR），栈指针寄存器（图中 SP, Stack Pointer）以及标志寄存器（图中 FLAG）。运算器为算术逻辑单元（图中 ALU），负责进行数学运算和逻辑运算。程序计数器在图中为 PC。控制部分为图中的 Control Unit (CU)，时钟信号由图中 Clock 发生。

二　模块划分

按照功能，我们将 CPU 划分为如下 11 个功能模块，下面逐个介绍这些模块的功能：

1. 时钟发生器（Clock）：时钟发生器的功能是根据外部时钟信号，产生一系列的特定时钟信号，送往其他部件，控制它们进行操作；

2. 控制器（Control Unit, CU）：控制器是 CPU 最核心的部件。CPU 通过它来产生控制时序和控制信号，用来控制 CPU 中其他模块（例如 ALU、寄存器）按照一定的时序关系工作。这一部分由两部分组成：状态机（CU）和状态机的控制器（CU_Control）；

3. 指令寄存器（IR REG）：指令寄存器用来存放当前正在执行的（PC 指向的）指令；

4. 通用寄存器组（REG ARRAY）：这个寄存器组全部用于存储数据，它由 8 个 16 位寄存器组组成。此部件支持双端口读出、双端口写入操作。这些寄存器用来保存程序执行过程中需要的数据、中间变量和最后的结果，在访问时，必须一次性读出全部 16 位数据；

5. 程序计数器（PC, Program Counter）：程序计数器保存下一条指令的地址，CPU 使用它在相应的 ROM 中取出指令；

6. 算术逻辑运算器（ALU, Arithmetic and Logic Unit）：算数逻辑运算器是一个 16 位定点运算器，支持基本的加减乘除、与或非等 14 算数、逻辑运算；

7. 运算输入控制部件：此部件用来控制运算器的输入数据。送入 ALU 的数据只有两种：来自数据寄存器的 16 位数据和来自立即数的 8 位数据；

8. 标志寄存器（FLAG）：标志寄存器共有 8 位，但是只使用了 4 位。高 4 位未定义，低 4 位存储运算过程中产生的标志位。低 4 位由高到低依次为：进位（CF）、零（ZF）、溢出（OF）和符号（SF）；

9. 地址寄存器（MAR, Main Address Register）：地址寄存器用来存储 CPU 访存时给出的地址，访存时的地址来源于此部件；

10. 数据寄存器（MDR, Main Data Register）：数据寄存器用来存储需要向地址总线输出的数据，或存储从总线上读取到的数据。它是双向输入输出的；

11. 栈指针寄存器（SP, Stack Pointer）：栈指针寄存器用于存储当前 1 的 16 位栈顶地址。每次出栈或入栈操作后，都要更新它的值。

    这些模块的具体实现将在第五节介绍。

三　指令执行流程

为了更清楚的描述此 CPU 的运行过程，下面介绍指令的执行流程。总体上，所有指令的执行均包含 3 个阶段：取出指令、指令译码和取出操作数、执行和回写。这其中每个阶段又分为 3 个小阶段，所以执行中共经历了 9 个阶段。负责此模块实现的控制单元（CU）中的状态机也就在 9 个状态间进行转移。

1. 算数逻辑运算指令：先把数据从寄存器中取出，然后经过运算器处理后再将结果写回寄存器，或设置相应的标志位。下面以带进位的加法指令 ADC 为例，介绍执行过程的各个阶段：[1]

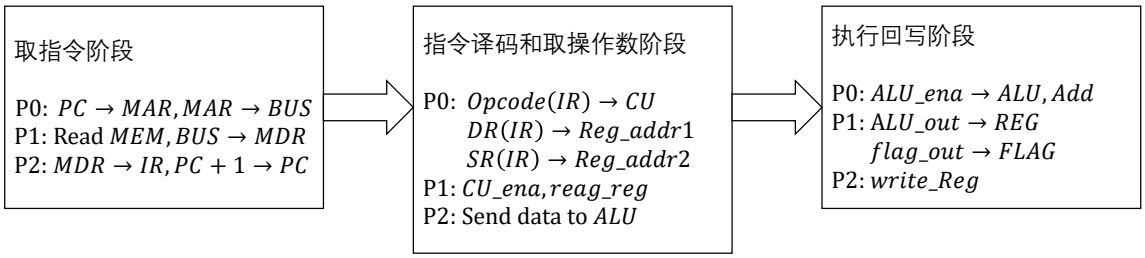


图 4: 算数和逻辑运算指令执行过程，以 ADC 为例

2. 数据传送类指令：此类指令包含在寄存器之间传输数据的 MOV，寄存器的高 8 位或低 8 位的加载指令（MOVIH/L $IMM, %DR）。下面以寄存器间的传送指令 MOV 为例，介绍执行过程的各个阶段：

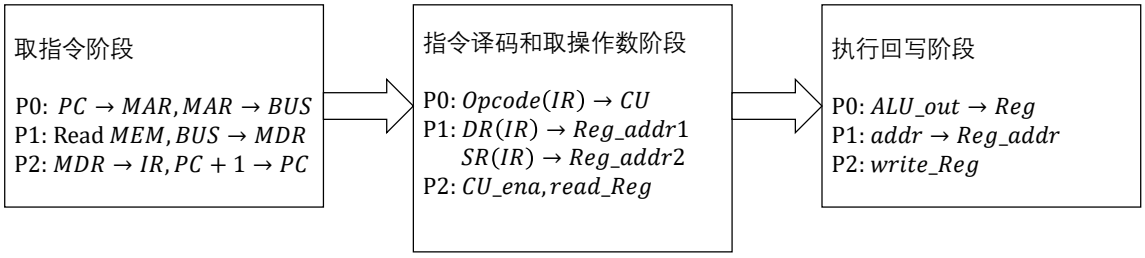


图 5: 数据传送类指令执行过程，以 MOV 为例

3. I/O 类指令：此类指令包含写 I/O 端口指令（OUT %SR, PORT）和读 I/O 端口指令（IN PORT, %DR）。下面以 OUT 指令为例，介绍执行过程的各个阶段：

[1]与 ADC 执行过程相似的指令还有：SBB, DIV, MUL, AND, NOT, OR, XOR, SHL, SHR, SAR, CMP, TEST 和 ADDI 等。

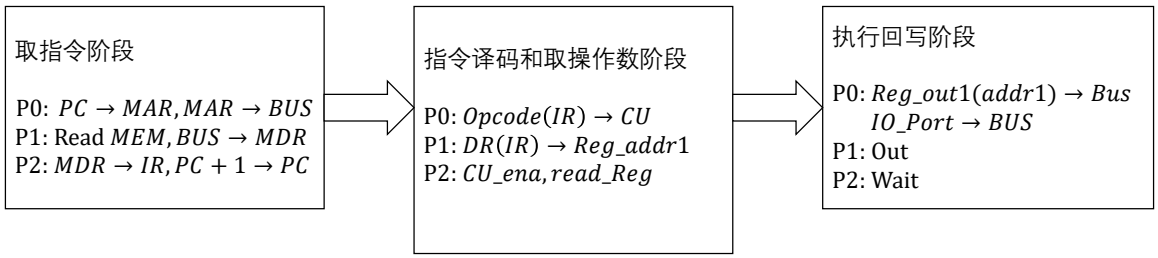


图 6: I/O 指令执行过程，以 OUT 为例

4. 控制转移类指令：可以分为有条件转移指令[2]和无条件转移指令[3]两大类。下面以 JMPC 为例，介绍执行过程的各个阶段：

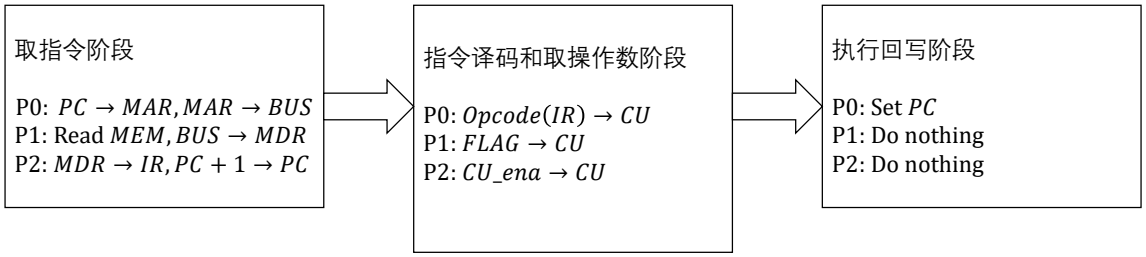


图 7: 条件转移类指令执行过程，以 JMPC 为例

5. 栈操作指令：包含入栈（PUSH %SR）和出栈（POP %DR）指令。下面以出栈指令 POP 为例，介绍执行过程的各个阶段：

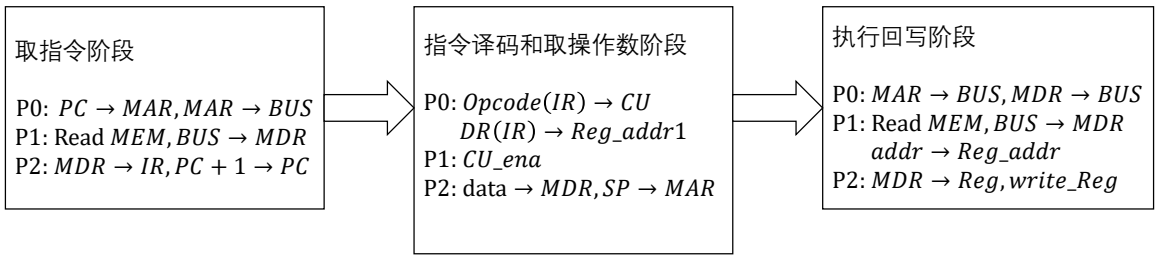


图 8: 栈操作指令执行过程，以 POP 为例

6. 访问内存类指令：包含取操作数指令（LOAD %SR, %DR）和存操作数指令（STORE %SR, %DR）。

[2]此类指令包括 JMPC, JNC, JMPZ 和 JNZ。

[3]此类指令为 JMP

LOAD 指令的作用是将源操作数寄存器中的数据当作地址，将这个内存地址中的数据装入目的寄存器%DR；STORE 指令的作用是将目的操作数寄存器中的数据当作地址，将源操作数中的数据存入此地址指向的内存。LOAD 和 STORE 的执行过程分别如下所示：

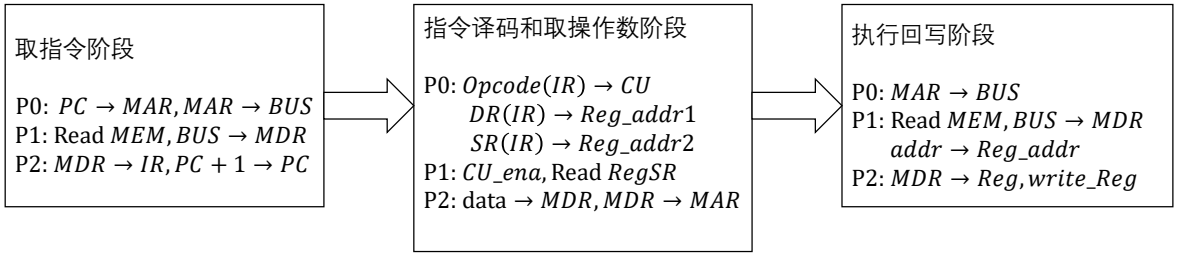


图 9: LOAD 指令执行过程

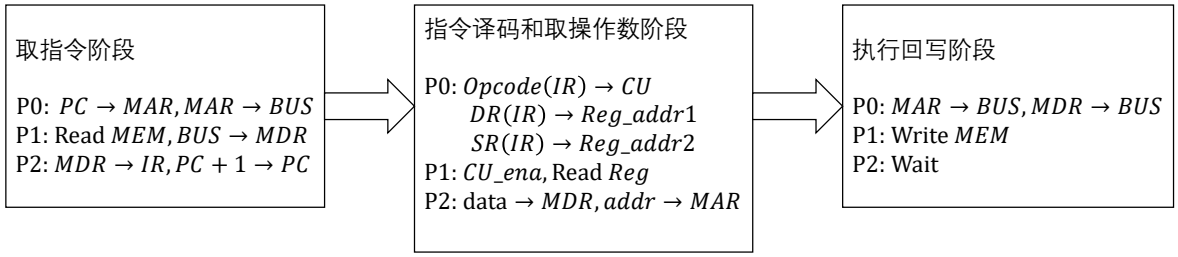


图 10: STORE 指令执行过程

7. 处理器控制类指令：包含空操作（NOP）和停机（HALT）两个指令。NOP 不执行任何操作，只将 PC 增加 1；HALT 指令负责停止 CPU 的运行。它们的执行过程分别如下图所示：

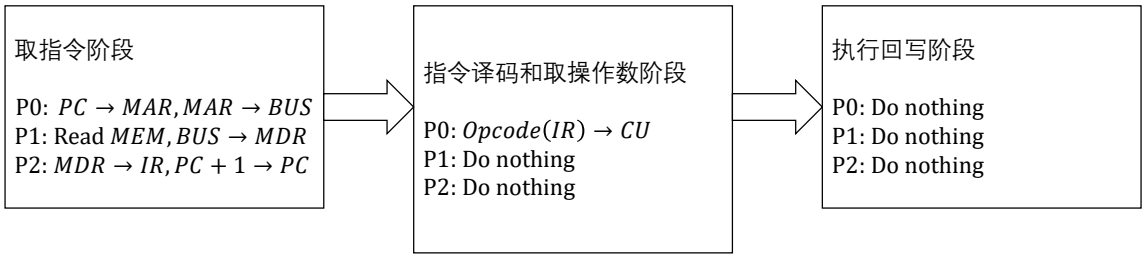


图 11: NOP 指令执行过程

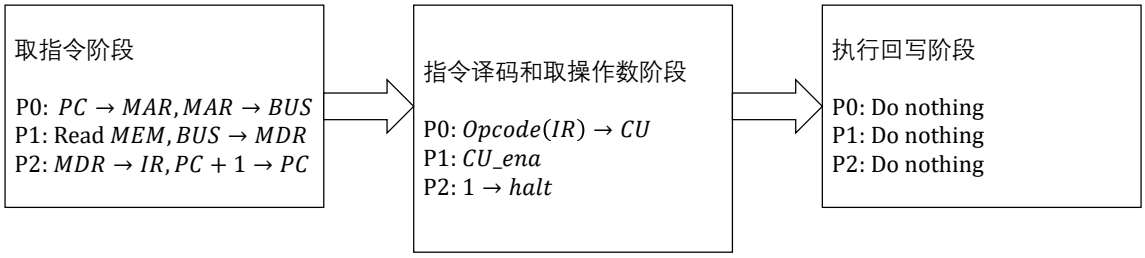


图 12: HALT 指令执行过程

第五节　实验过程

本节介绍图 3[4]中标出的各个模块的主要构成和 *Verilog HDL* 代码实现。

一　时钟发生器

时钟发生器（Clock）模块利用外来的时钟信号 *clkin* 来生成一系列的时钟信号 *clkout*, $idfetch$ 送往 CPU 的其他部件。实现代码如下所示，为了简化 CPU 频率控制功能的实现，分成了分频器和时钟信号发生器。下面先说明分频器的信号端口：

输入信号：
clkin——时钟信号；
clkcon——频率控制信号；
reset——复位信号。
输出信号：
clkout——分频后的时钟信号。

和时钟发生器的信号端口：

输入信号：
clk——分频后的时钟信号；
reset——复位信号。
输出信号：
clk_r——*clk* 取反后的时钟信号；
fetch——*clk* 的八分频，取指令信号。

[4]第 6 页

```
module divider(
    input rst,
    input [2:0] clkcon,
    input clkin,        //100MHz
    output reg clkout=0   //Hz
    );
    integer cnt=0;
    integer num=30_00000;

    always @ (posedge clkin)
    begin
        if(!rst)
            casex(clkcon)
                3'b000:
                begin num=80_00000; cnt=0; end
                3'b001:
                begin num=20_00000; cnt=0; end
                3'b010:
                begin num=10_00000; cnt=0; end
                3'b011:
                begin num=3_50000; cnt=0; end
                3'b100:
                begin num=1_00000; cnt=0; end
                3'b101:
                begin num=4_0000; cnt=0; end
                3'b110:
                begin num=700; cnt=0; end
            endcase
        else
        begin
            if(cnt==num)
            begin
                clkout=~clkout;
                cnt=0;
            end
            else
                cnt=cnt+1;
        end
    end
endmodule
```

```
module clock(
    input clk,
    input reset,
    output reg fetch,
    output clk_r
    );
    integer cnt=0;

    assign clk_r=~clk;
    always @ (negedge clk)
    begin
        if(!reset)
        begin
            cnt=0;
            fetch=0;
        end
        else
        begin
            if(cnt==3)
            begin
                cnt=cnt+1;
                fetch=1;
            end
            else if(cnt==7)
            begin
                cnt=0;
                fetch=0;
            end
            else
            begin
                cnt=cnt+1;
            end
        end
    end
endmodule
```

代码 1: 时钟模块

下面是此时钟发生器的模块符号，以及仿真结果。

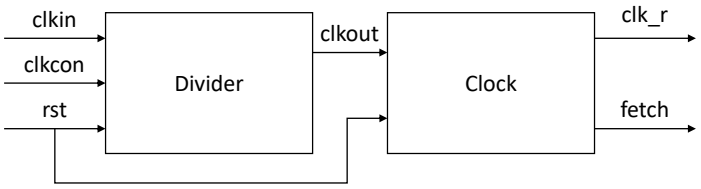


图 13: 时钟模块符号

二　程序计数器

程序计数器（PC, Program Counter）用于提供执行指令的地址。指令按顺序存放在存储器（ROM）中，在控制器的控制下按顺序读取并执行指令。CPU 的控制逻辑有两种生成指令地址的方法：

(1) 顺序执行时，$PC = PC + 1$（规定每条指令长度都为 1 个字，16 位）；

(2) 需要跳转时，根据跳转指令改变 PC 的值。

端口说明如下：

输入信号：
clk——时钟信号；
rst——复位信号；
offset——转移时的偏移量；
pc_inc——自增 1 控制信号（低电平时 PC 加 1）；
pc_ena——PC 更新使能信号；
sw——程序选择信号。
输出信号：
pc_value——PC 的值，即指令地址。

复位（rst 置高位后），$PC = sw \times 32 + 10$. 即每次 CPU 重新启动时，将根据程序选择信号 sw 读取相应起始地址中的指令，然后执行。在指令执行阶段，译码过后根据指令内容对程序计数器进行更新。如果执行的是跳转指令，则控制器（CU）根据标志寄存器（FLAGS）中的相关状态位来判断是否发生跳转，具体过程如下表所示。如果需要发生跳转，控制器将 pc_ena 和 pc_inc 信号输出为高电位，PC 的值在本模块中得到更新；如果跳转条件不满足或不是跳转指令，则 CU 将 pc_ena 置位高电位，pc_inc 置为低电位，此时 PC 自增 1。

表 2: 指令操作码与指令寄存器更新的逻辑关系

指令	操作码	零标识 ZF	进位标志 CF	是否跳转	更新后的 PC 值
JMP	10111	X	X	是	$PC = PC + offset$
JMPC	11000	X	0	否	$PC = PC + 1$
		X	1	是	$PC = PC + offset$
JNC	11001	X	0	是	$PC = PC + offset$
		X	1	否	$PC = PC + 1$
JMPZ	11010	0	X	否	$PC = PC + 1$
		1	X	是	$PC = PC + offset$
JNZ	11011	0	X	是	$PC = PC + offset$
		1	X	否	$PC = PC + 1$
其他	–	X	X	否	$PC = PC + 1$

```
module pc(
    input clk,  //clock signal, "fetch" in clock.v
    input rst,  //reset the program counter
    input [10:0] offset,   //offset when jmp instruction is needed
    input pc_inc,   //when this signal in low voltage, counter increments by 1
                    //in high voltage, a jmp is executed
    input pc_ena,   //enable signal, control whether to update PC or not
    input [1:0] sw,
    output reg [15:0] pc_value//current value of PC
    );
    always @ (posedge clk or negedge rst)
    begin
        if(!rst)
            pc_value=(sw*32)+10;
        else
        begin
            if(pc_ena&&pc_inc&&offset[10]) //jmp, offset is negative
                pc_value=pc_value-offset[9:0];
            else if(pc_ena&&pc_inc&&(!offset[10]))  //jmp, offset is positive
                pc_value=pc_value+offset[9:0];
            else if(pc_ena)
                pc_value=pc_value+1'b1;
        end
    end
endmodule
```

代码 2: PC 模块

下面是此模块的符号和仿真结果。

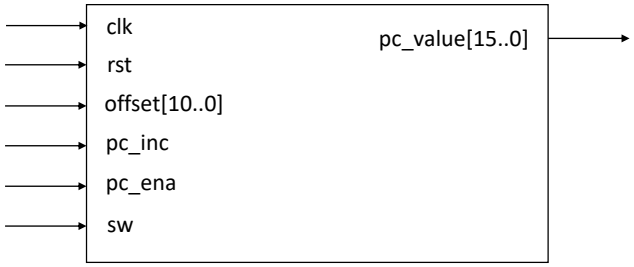


图 14: PC 模块符号

三　指令寄存器

指令寄存器（Instruction Register, instreg）用于暂存当前正在执行的指令。它的时钟信号是 clk，在 clk 的上升沿触发。指令寄存器将数据总线送来的指令存入 16 位寄存器中。但总线上传输的数据不需要寄存，因此控制器的 ir_ena 信号用来指示数据类型，控制总线数据是否需要寄存。复位（rst 置为高位）时，指令寄存器被清零。

由于每条指令为 2 个字节，16 位，其中高 5 为为操作码，低 11 位是偏移地址或目的操作数寄存器和源操作数寄存器，设计中采用的数据总线为 16 位，每取出一条指令就访存一次。其中，各端口的说明如下：

输入信号：

clk——时钟信号；

rst——复位信号；

data——数据总线输入；

ir_ena——IR 更新控制。

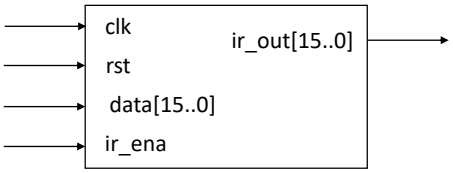
输出信号：

ir_out——当前执行的指令，16 位二进制串。

```
module instreg(
    input clk,  //clock, "fetch" in clock.v
    input rst,  //reset the instruction register
    input [15:0] data,  //data from the bus
    input ir_ena,       //whether to update the instruction register or not
    output reg [15:0] ir_out    //current value of the instruction register
    );
    always @ (posedge clk)
    begin
        if(!rst)
        begin
            ir_out=16'h0000;
        end
        else if(ir_ena) //now bus tranfer instruction to variable data
        begin
            ir_out=data;//in this case, data is the instruction from bus
        end
    end
endmodule
```

代码 3: 指令寄存器模块

下面是此模块的符号和仿真结果。



指令寄存器模块符号

四　地址寄存器

地址寄存器（MAR）是 CPU 与存储器的一个接口，用于存放待访问的寄存器单元的地址。它的输入和输出关系如下表所示。

其各端口信号如下：

输入信号：

clk——时钟信号；

rst——复位信号；

mar_ena——MAR 输入使能信号；

mar_sel——MAR 输入选择信号，用于选择地址输入；

ir_addr1, *ir_addr2*——来自寄存器的地址；

pc_addr——PC 数据；

表 3: 地址寄存器的输入、输出逻辑关系

输入				输出
clk	rst	mar_ena	mar_ena	mar_addr
1	1	X	XX	0x0000
1	0	1	00	*pc_addr*
1	0	1	01	*ir_addr1*
1	0	1	10	*ir_addr2*
1	0	1	11	*sp*
0	X	X	X	0xXXXX

sp_addr——栈指针（SP）数据。

输出信号：

mar_addr——地址寄存器的输出。

下面是此模块的符号和仿真结果。

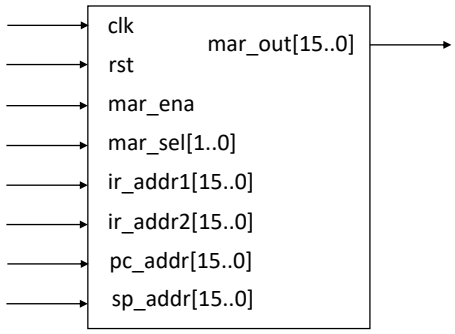


图 15: 地址寄存器模块的符号

```
module mar(
    input clk,  //clock signal, "fetch" in clock.v
    input rst,  //reset signal
    input mar_ena,  //whether enable memory address register or not
    input [1:0] mar_sel,//select the address source
    input [15:0] ir_addr1,  //data from register
    input [15:0] ir_addr2,  //data from register
    input [15:0] pc_addr,  //data from PC
    input [15:0] sp_addr,  //data from sp
    output reg [15:0] mar_addr  //memory address
    );
    parameter pc=2'b00,//from PC
              dr=2'b01,//from reg_out1[DR]
              sr=2'b10,//from reg_out2[SR]
              sp=2'b11;//from sp

    always @ (posedge clk)
    begin
        if(!rst)
        begin
            mar_addr=16'h0000;  //reset the address
        end
        else if(mar_ena)
        begin
            case(mar_sel)
            pc:
            mar_addr=pc_addr;
            dr:
            mar_addr=ir_addr1;
            sr:
            mar_addr=ir_addr2;
            sp:
            mar_addr=sp_addr;
            endcase
        end
    end
endmodule
```

代码 4: 地址寄存器模块

五　数据寄存器

数据寄存器（MDR）用于将寄存将要被输出到数据总线或送往 ALU 的数据。各端口的说明如下：

输入信号：

clk——时钟信号；

rst——复位信号；

mdr_ena——MDR 输入使能信号；

mdr_sel——MDR 输入选择信号，用于选择地址输入；
reg_in1,*reg_in2*——来自寄存器的输入数据；
mem_in——来自存储器的输入数据。
输出信号：
mar_addr——向数据总线的数据输出；
reg_out——向指令寄存器（instreg）的输出。

表 4: 地址寄存器的输入、输出逻辑关系

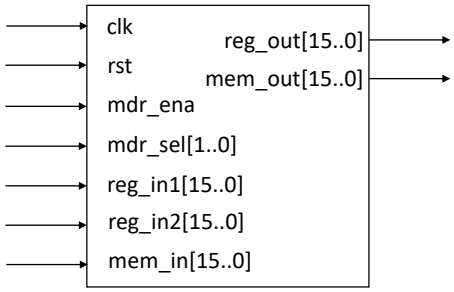
输入				输出	
clk	rst	mdr_ena	mdr_ena	mar_addr	mem_out
1	1	X	XX	0xXXXX	0xXXXX
1	0	1	00	0xXXXX	0xXXXX
1	0	1	01	*reg_in1*	0xXXXX
1	0	1	10	*reg_in2*	0xXXXX
1	0	1	11	0xXXXX	*mem_in*
0	X	X	X	0xXXXX	0xXXXXX

```
module mdr(
    input clk,  //clock
    input rst,  //reset
    input mdr_ena,  //mdr enable
    input [1:0] mdr_sel,  //data
select
    input [15:0] reg_in1,   //signal
from register
    input [15:0] reg_in2,
    input [15:0] mem_in,    //signal
from memory
    output reg [15:0] mem_out,
//output to data bus
    output reg [15:0] reg_out
//output to instruction register
    );

    parameter regin1=2'b01,
              regin2=2'b10,
              memin=2'b11;
    always @ (posedge clk or negedge
rst)
begin
        if(!rst)
        begin
            reg_out=16'h0000;
            mem_out=16'h0000;
        end
    else if(mdr_ena)
        begin
            case(mdr_sel)
            regin1:
            begin
                reg_out=reg_in1;
                mem_out=16'hz;
            end
            regin2:
            begin
                reg_out=reg_in2;
                mem_out=16'hz;
            end
            memin:
            begin
                reg_out=16'hz;
                mem_out=mem_in;
            end
            default:
            begin
                reg_out=16'hz;
                mem_out=16'hz;
            end
            endcase
        end
    end
endmodule
```

代码 5: 数据寄存器模块

下面是此模块的符号和仿真结果。



代码 5: 数据寄存器的模块符号

六　寄存器组

寄存器组（Register Array, regarray）用于存储指令执行过程中需要的数据以及指令的执行结果。它一共包含 8 个 16 位的通用寄存器，双端口读出、双端口写入，只能按字访问，即一次性读写 16 位数据。其端口信号说明如下：

输入信号：
clk——时钟信号；
rst——复位信号；
reg_read1——端口 1 读取控制信号（高电平有效）；
reg_read2——端口 2 读取控制信号（高电平有效）；
addr1——端口 1 地址信号；
addr2——端口 2 地址信号；
reg_write1——端口 1 写入控制信号（高电平有效）；
reg_write2——端口 2 写入控制信号（高电平有效）；
data_in1, *data_in2*, *data_in3*——3 个输入信号。
输出信号：
reg_out1——读端口 1 输出数据；
reg_out2——读端口 2 输出数据。

寄存器组的 3 路 16 位输入数据分别为 data_in1, data_in2, data_in3。2 路 16 位读出数据分别为 reg_out1, reg_out2。2 个端口的读取控制信号分别为 reg_read1, reg_read2，写入控制信号分别为 reg_write1, reg_write2。从寄存器组读出的数据可以送到 ALU 的数据输入端的多路选择器，也可以是访存指令地址。数据寄存器原理上只能按字节读写，但是为了支持涉及到 8 位数据的操作指令[5]，使用以下方法来支持它们。当进行 8 位数据的写操作时：

[5]MOVIL $IMM, %DR，MOVIH $IMM, %DR，具体功能见第4页表格。

1. 读数据阶段：将目的寄存器（DR）的数据取出；
2. 指令执行阶段：将指令中的 8 位立即数（IMM）拼接到目的寄存器（DR）的相应 1 字节的位置上；
3. 回写阶段：将拼接后的结果写回目的寄存器（DR）中。

这样的设计可以简化寄存器组的读写控制逻辑，实现代码如下：

```
module regarray(
    input clk,
    input rst,
    input reg_read1,
    input reg_read2,
    input [2:0] addr1,//address
    input [2:0] addr2,
    input reg_write1,//write control
signal
    input reg_write2,
    input [15:0] data_in1,//three
data inputs
    input [15:0] data_in2,
    input [15:0] data_in3,
    output reg [15:0]
reg_out1,//output data from port 1
or 2
    output reg [15:0] reg_out2,
    output [15:0] port
    );
    reg [15:0] register [7:0];
    assign port = register[7];
    parameter num=8;//number of
registers
    integer i;
    always @ (posedge clk)
    begin
        if(!rst)
        begin
            for(i=0;i<num;i=i+1)

register[i]=16'h0000;//reset the
data in each register
        end
        else
if(reg_read1&&reg_read2)
begin

reg_out1=register[addr1];

reg_out2=register[addr2];
end
else if(reg_read1)
begin

reg_out1=register[addr1];
            reg_out2=16'hz;
end
else if(reg_read2)
begin
            reg_out1=16'hz;

reg_out2=register[addr2];
end
else
begin
            case({reg_write1,
reg_write2})
            2'b11:
            begin

register[addr1]=data_in1;

register[addr2]=data_in2;
            end
            2'b01:
            begin

register[addr1]=data_in3;
            end
            2'b10:
            begin

register[addr1]=data_in1;
            end
            endcase
        end
    end
endmodule
```

代码 6: 寄存器组模块

此模块的符号如下：

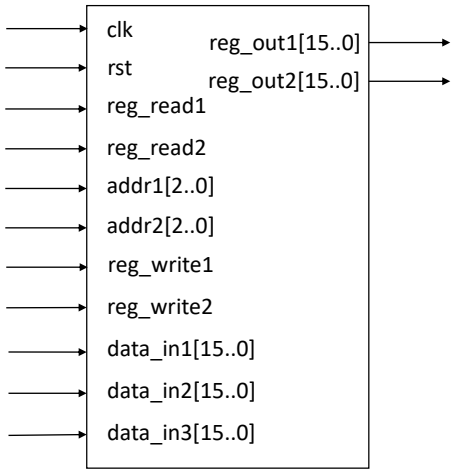


图 16: 寄存器组的模块符号

七　栈指针寄存器

栈指针寄存器（SP）是用于存储栈顶地址的。入栈（PUSH）和出栈（POP）指令访问的存储单元是由 SP 发出的。我们规定 CPU 栈顶起始地址为 0x0200，也即初始化时 SP 的值。当执行入栈指令（PUSH）时，控制信号 sp_push 有效，SP 自增 1；执行出栈（POP）指令时，控制信号 sp_pop 有效，SP 自减 1。其各端口信号说明如下：

输入信号：

clk——时钟信号；

rst——复位信号；

sp_push——入栈控制信号；

sp_pop——出栈控制信号。

输出信号：

sp_value——栈顶地址。

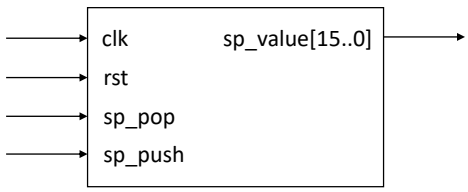


图 17: 栈指针寄存器的模块符号

```
module sp(
    input clk,
    input rst,
    input sp_pop,
    input sp_push,
    output reg [15:0] sp_value//stack top pointer, similiar to esp
    );

    always @ (posedge clk)
    begin
        if(!rst)
        begin
            sp_value=16'h0200;
        end
        else
        begin
            if(sp_push)
            begin
                sp_value=sp_value+1'b1;
            end
            else if(sp_pop)
            begin
                sp_value=sp_value-1'b1;
            end
        end
    end
endmodule
```

代码 7: 栈指针寄存器模块