

CPU 设计实验报告

姓名：陈迪

学号：201608010613

计科 1604

一、实验目的：

- 1、掌握硬件描述语言 VHDL 和 EDA 工具 Quartus II；利用 VHDL 设计 16 位串行 CPU，实现算术和逻辑运算指令、转移指令、访存指令、堆栈指令和控制指令；
- 2、掌握 CPU 的调试和测试方法。

二、实验内容：

- 完成一个模拟 RISC-V 的基本整数指令集 RV32I 的汇编器设计
- 完成一个模拟 RISC-V 的基本整数指令集 RV32I 的模拟器设计
- 完成一个简单存储器的设计
- 完成一个执行 RISC-V 的基本整数指令集 RV32I 的 CPU 设计（单周期实现）

三、实验步骤：

1、寄存器堆的设计

寄存器堆由 16 个 16 位寄存器组成。其中 reset 是异步清 0 端，reset=0 时将所有寄存器清 0。dr_sel 和 sr_sel 是目标寄存器和源寄存器编号，dr_out 和 sr_out 输出目标寄存器和源寄存器的内容。reg_sel 指定一个寄存器编号，将该寄存器内容送给 reg_out，这两个端口用于调试时观察每个寄存器的值。reg_en 是写允许端。reg_en=“01”时，在 clk 的上升沿将 from_alu 写入 dr_sel 指定的寄存器；reg_en=“10”时，在 clk 的上升沿将 from_mem 写入 dr_sel 指定的寄存器。reg_en 取其他值时不改变寄存器堆的值。

设计方法：

```
subtype WORD is std_logic_vector(15 downto 0);
type REGISTERARRAY is array ( 0 to 15 ) of WORD;
signal reg_bank: REGISTERARRAY
```

则 reg_bank 就是我们所需要的寄存器堆。

写寄存器堆的方法：

```
reg_bank(conv_integer(dr_sel))<=from_alu;
```

读寄存器堆的方法：

```
dr_out <= reg_bank(conv_integer(dr_sel));
```

其中，conv_integer 是 STD_LOGIC_UNSIGNED 程序包提供的函数，将标准逻辑矢量转换成整数，作为 reg_bank 的下标。

2、算数逻辑单元设计

ALU 可以实现 16 种运算。alu_func 是运算功能选择，alu_a 和 alu_b 是两个操作数，c_in 是进位标志输入，用于实现 ADC 和 SBB。alu_o 是运算结果，c、s、z、o 分别是进位标志，符号标志，零标志和溢出标志。

ALU 功能如下图所示：

		功能	alu_func	对标志位的影响
ADD	dr, sr	$dr + sr \rightarrow dr$	0000	影响c, s, z, o
SUB	dr, sr	$dr - sr \rightarrow dr$	0001	
CMP	dr, sr	$dr - sr$	0010	
ADC	dr, sr	$dr + sr + c \rightarrow dr$	0011	
SBB	dr, sr	$dr - sr - c \rightarrow dr$	0100	
INC	dr	$dr + 1 \rightarrow dr$	0101	影响s, z, o 不影响c
DEC	dr	$dr - 1 \rightarrow dr$	0110	
SAL	dr	dr 左移一位, 右补0	0111	影响c, s, z, o
SAR	dr	dr 右移一位, 左补最高位	1000	
SHR	dr	dr 右移一位, 左补0	1001	
MOV	dr, sr	$sr \rightarrow dr$	1010	不影响任何标志位
AND	dr, sr	$dr \text{ AND } sr \rightarrow dr$	1011	影响s, z 将c和o清零
TEST	dr, sr	$dr \text{ AND } sr$	1100	
OR	dr, sr	$dr \text{ OR } sr \rightarrow dr$	1101	
XOR	dr, sr	$dr \text{ XOR } sr \rightarrow dr$	1110	
NOT	dr	$\text{NOT } dr \rightarrow dr$	1111	不影响任何标志位

设计方法:

7 条算术运算指令可以调用 Quartus 提供的模块 lpm_add_sub 实现。

在库使用说明中增加:

```
LIBRARY LPM;
```

```
USE LPM.LPM_COMPONENTS.ALL;
```

在构造体的 ARCHITECTURE 和 BEGIN 关键字之间, 用 COMPONENT 语句声明该模块:

```
COMPONENT lpm_add_sub
```

```
GENERIC(lpm_width: NATURAL;
```

```
    lpm_direction: STRING;
```

```
    lpm_type: STRING;
```

```
    lpm_hint: STRING);
```

```
PORT(dataa: IN STD_LOGIC_VECTOR(lpm_width - 1 DOWNT0 0);
```

```
    datab: IN STD_LOGIC_VECTOR(lpm_width - 1 DOWNT0 0);
```

```
    cin: IN STD_LOGIC;
```

```
    result: OUT STD_LOGIC_VECTOR(lpm_width - 1 DOWNT0 0);
```

```
    cout: OUT STD_LOGIC;
```

```
    overflow: OUT STD_LOGIC);
```

```
END COMPONENT;
```

并定义几个信号:

```
SIGNAL addsub_cin: STD_LOGIC;
```

```
SIGNAL addsub_c : STD_LOGIC;
```

```
SIGNAL addsub_o : STD_LOGIC;
```

```
SIGNAL addsub_a : STD_LOGIC_VECTOR(15 DOWNT0 0);
```

```
SIGNAL addsub_b : STD_LOGIC_VECTOR(15 DOWNT0 0);
SIGNAL addsub_r : STD_LOGIC_VECTOR(15 DOWNT0 0);
```

在构造体中，用 GENERIC MAP 和 PORT MAP 语句说明 lpm_add_sub 的连接方式：

```
ALU_ADDSUB: lpm_add_sub
GENERIC MAP (lpm_width => 16,
             lpm_direction => "ADD",
             lpm_type => "LPM_ADD_SUB",
             lpm_hint => "ONE_INPUT_IS_CONSTANT = NO,
CIN_USED = YES")
PORT MAP (dataa => addsub_a,
          datab => addsub_b,
          cin => addsub_cin,
          result => addsub_r,
          cout => addsub_c,
          overflow => addsub_o);
```

实现 7 条算术运算指令时，addsub_a、addsub_b、addsub_cin 的取值如下：

	ADD	SUB	CMP	ADC	SBB	INC	DEC
addsub_a	alu_a	alu_a	alu_a	alu_a	alu_a	alu_a	alu_a
addsub_b	alu_b	NOT alu_b	NOT alu_b	alu_b	NOT alu_b	x"0000"	x"FFFF"
addsub_cin	0	1	1	c_in	NOT c_in	1	0

按照上述取值方法，addsub_r 和 addsub_o 就是这 7 条指令的运算结果和溢出标志。对于 ADD 和 ADC 指令，addsub_c 就是进位标志，对于 SUB、CMP 和 SBB，应将 addsub_c 取反后作为进位标志，而 INC 和 DEC 不影响进位标志，应将 c_in 作为进位标志。

三条移位指令的实现方法：

	运算结果alu_o	进位标志c	溢出标志o
SAL	alu_a(14 DOWNT0 0) & '0'	alu_a(15)	alu_a(15) XOR alu_a(14)
SAR	alu_a(15) & alu_a(15 DOWNT0 1)	alu_a(0)	'0'
SHR	'0' & alu_a(15 DOWNT0 1)	alu_a(0)	alu_a(15)

即：将被移出的位作为进位标志，将移位前后最高位的异或作为溢出标志。

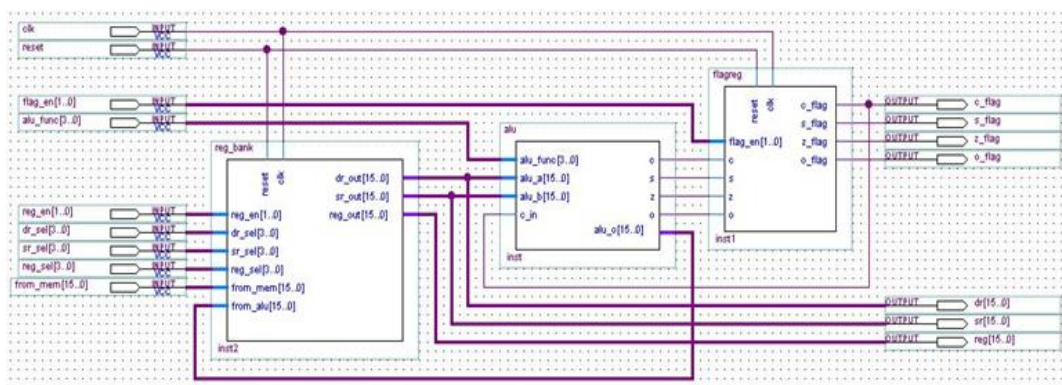
3、标志寄存器的设计

标志寄存器用来保存 c，s，z，o 四个标志位。

flag_en=“00”时不改变四个标志位；flag_en=“01”时同时保存四个标志位；flag_en=“10”时将进位标志清零，用于实现 CLC 指令；flag_en=“11”时将进位标志置 1，用于实现 STC 指令。

4、执行单元的设计

将寄存器堆，ALU 和标志寄存器连接在一起，构成执行单元。



编译后，为执行单元生成一个图形符号。

5、指令译码单元的设计

指令集共 35 条指令，其中单字长（16 位）指令 33 条，双字长（32 位）指令 2 条。

		功能
ADD	<u>dr</u> , <u>sr</u>	$\text{dr} + \text{sr} \rightarrow \text{dr}$
SUB	<u>dr</u> , <u>sr</u>	$\text{dr} - \text{sr} \rightarrow \text{dr}$
CMP	<u>dr</u> , <u>sr</u>	$\text{dr} - \text{sr}$
ADC	<u>dr</u> , <u>sr</u>	$\text{dr} + \text{sr} + \text{c} \rightarrow \text{dr}$
SBB	<u>dr</u> , <u>sr</u>	$\text{dr} - \text{sr} - \text{c} \rightarrow \text{dr}$
INC	<u>dr</u>	$\text{dr} + 1 \rightarrow \text{dr}$
DEC	<u>dr</u>	$\text{dr} - 1 \rightarrow \text{dr}$
SAL	<u>dr</u>	<u>dr</u> 左移一位，右补0
SAR	<u>dr</u>	<u>dr</u> 右移一位，左补最高位
SHR	<u>dr</u>	<u>dr</u> 右移一位，左补0
MOV	<u>dr</u> , <u>sr</u>	$\text{sr} \rightarrow \text{dr}$
AND	<u>dr</u> , <u>sr</u>	$\text{dr} \text{ AND } \text{sr} \rightarrow \text{dr}$
TEST	<u>dr</u> , <u>sr</u>	$\text{dr} \text{ AND } \text{sr}$
OR	<u>dr</u> , <u>sr</u>	$\text{dr} \text{ OR } \text{sr} \rightarrow \text{dr}$
XOR	<u>dr</u> , <u>sr</u>	$\text{dr} \text{ XOR } \text{sr} \rightarrow \text{dr}$
NOT	<u>dr</u>	$\text{NOT } \text{dr} \rightarrow \text{dr}$

	功能
CLC	进位标志清0
STC	进位标志置1
JMP <u>addr</u>	无条件转移
JC <u>addr</u>	C=1则转移
JNC <u>addr</u>	C=0则转移
JS <u>addr</u>	S=1则转移
JNS <u>addr</u>	S=0则转移
JZ <u>addr</u>	Z=1则转移
JNZ <u>addr</u>	Z=0则转移
JO <u>addr</u>	O=1则转移
JNO <u>addr</u>	O=0则转移
NOP	空操作
HLT	停机
LDR <u>dr</u> , <u>sr</u>	<u>sr</u> 中是一个存储单元地址，将该单元内容送 <u>dr</u>
STR <u>dr</u> , <u>sr</u>	将 <u>sr</u> 内容送入 <u>dr</u> 指定的存储单元
PUSH <u>sr</u>	<u>sr</u> → [SP], SP+1
POP <u>dr</u>	SP-1, [SP] → <u>dr</u>

相对转移指令。执行时将PC的值加上一个带符号的偏移量，作为后继指令的地址。

堆栈指令。CPU初始化时将SP置为0x 0280。

双字长指令：

	功能
JMPA <u>addr</u>	绝对转移指令，将16位地址 <u>addr</u> 送入PC作为后继指令地址
MVRD <u>dr</u> , <u>data</u>	将16位立即数 <u>data</u> 送入 <u>dr</u>

指令编码采用定长操作码，所有指令的操作码都是 8 位。为了将汇编指令转换为二进制形式的机器指令，需要编写一个规则文件。规则文件是一个纯文本文件，以行为单位，每行指定一条汇编指令的机器码。

注释：以/开头的行是注释。

常量定义：R0 IS 0 定义了一个常量 R0，其值为 0

指令格式定义：包括三个字段，第一字段是指令名，第二字段是操作数声明，第三字段说明如何生成该指令的机器码。

指令译码单元共产生 9 个控制信号。

dr_sel <= ir(7DOWNT0 4);

sr_sel <= ir(3 DOWNT0 0);

alu_func <= ir(11 DOWNT0 8);

mem_wr: 为 1 表示写存储器，为 0 表示读存储器。

SIGNAL OP : STD_LOGIC_VECTOR(7 DOWNT0 0);

OP <= ir(15 DOWNT0 8);

mem_wr <= '1' WHEN S = '0' AND OP = "10010000" ELSE -- STR 指令

指令

```

        '1' WHEN S = '0' AND OP = "10100000" ELSE -- PUSH
        '0';

```

类似地写出以下 5 个控制信号的代码：

reg_en: 寄存器堆的写允许信号。

flag_en: 标志寄存器的写允许信号。

jmp_relv: 类型为 STD_LOGIC

为 '1' 表示进行相对转移，为 '0' 表示不进行相对转移。

sp_en: 类型为 STD_LOGIC_VECTOR(1 DOWNTO 0)

为 "01" 表示堆栈指针 SP 加 1，为 "10" 表示堆栈指针 SP 减 1。

addr_sel: 类型为 STD_LOGIC_VECTOR(2 DOWNTO 0)

用于确定 S=0 时地址总线上的值。"000" 表示 SP，"001" 表示 SP-1，

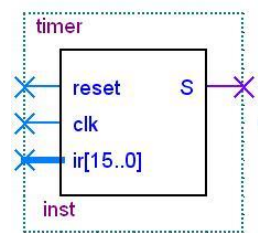
"010" 表示 DR，"011" 表示 SR，"100" 表示 PC。

6、节拍发生器

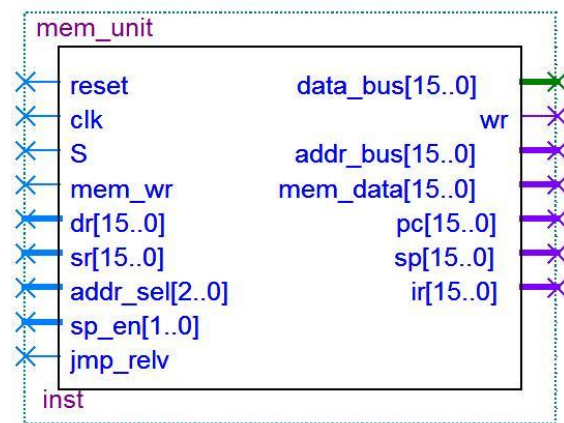
节拍发生器用来产生节拍信号 S。

reset=0 时将 S 清 0；

clk 出现上升沿时，若 S=0 且 ir 中保存的是 HLT 指令，则 S 继续为 0，CPU 进入死锁状态；否则对 S 取反。

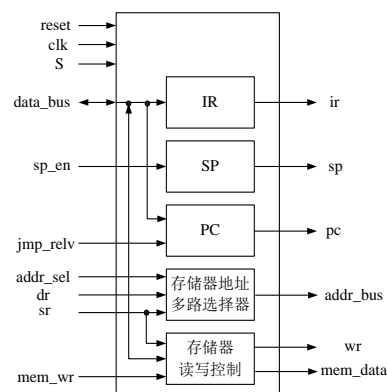


7、访存单元的设计



data_bus 连接数据总线，端口方向为 INOUT；wr 连接存储器的读写控制端；addr_bus 连接地址总线。

访存单元的内部结构：



IR: 指令寄存器。取指时将数据总线上传来的指令锁存进 IR。初始化时可以向 IR 存入 NOP 指令。

SP: 堆栈指针。执行堆栈指令后根据 sp_en 的值对 SP 进行修改。初始化时向 SP 存入 0280 H。

PC: 程序计数器

```
SIGNAL OP      : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL offset: STD_LOGIC_VECTOR(15 DOWNTO 0);
OP <= ir(15 DOWNTO 8);
offset <= "11111111" & ir(7 DOWNTO 0) WHEN ir(7) = '1' ELSE
         "00000000" & ir(7 DOWNTO 0);    -- 将偏移量扩展为 16 位
PROCESS (reset, clk)
    VARIABLE tmp: STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
    IF reset = '0' THEN
        tmp := x "0000";
    ELSIF clk'EVENT AND clk = '1' THEN
        IF S = '1' THEN                -- 取指令之后 PC 加 1
            tmp := tmp + 1;
        ELSIF jmp_relv = '1' THEN      -- 进行相对转移
            tmp := tmp + offset;
        ELSIF OP = "11000000" THEN     -- JMPA 指令
            tmp := data_bus;
        ELSIF OP = "11010000" THEN     -- MVRD 指令
            tmp := tmp + 1;
        END IF;
    END IF;
    pc <= tmp;
END PROCESS;
```

存储器地址选择:

```
PROCESS (reset, S, addr_sel, sp, dr, sr, pc)
    VARIABLE tmp: STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
    IF reset = '0' THEN
        tmp := x "0000";
    ELSIF S = '1' THEN                -- 取指令
        tmp := pc;
    ELSE
        IF addr_sel = "000" THEN      -- PUSH
            tmp := sp;
        ELSIF addr_sel = "001" THEN   -- POP
            tmp := sp - 1;
        ELSIF addr_sel = "010" THEN   -- STR
            tmp := dr;
        ELSIF addr_sel = "011" THEN   -- LDR
```



```

        tmp := sr;
    ELSIF addr_sel = "100" THEN      -- 双字长指令
        tmp := pc;
    ELSE
        tmp := "XXXXXXXXXXXXXXXXXX";
    END IF;
END IF;
addr_bus <= tmp;
END PROCESS;

```

存储器读写控制:

```

-- mem_wr = '1' : wr <= clk,    写存储器
-- mem_wr = '0' : wr <= '1'    读存储器

```

```

PROCESS(clk, mem_wr, data_bus, sr)
BEGIN
    IF mem_wr = '1' THEN -- 写存储器 (STR 指令, PUSH 指令)
        data_bus <= sr;
        wr <= clk;
    ELSE -- 读存储器
        data_bus <= "ZZZZZZZZZZZZZZZZ";
        wr <= '1';
        mem_data <= data_bus;
    END IF;
END PROCESS;

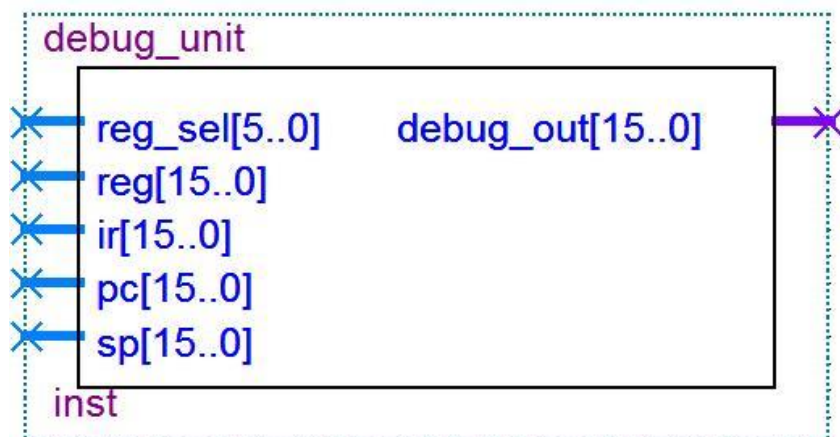
```

8、调试单元的设计

调试单元是一个多路选择器，用于在调试过程中观察 CPU 内部主要寄存器的值。

reg_sel 是数据选择端。取 63 时将 ir 送到输出；取 62 时将 pc 送到输出；取 16 时将 sp 送到输出；取 0~15 时，将来自寄存器堆的 reg 送到输出。

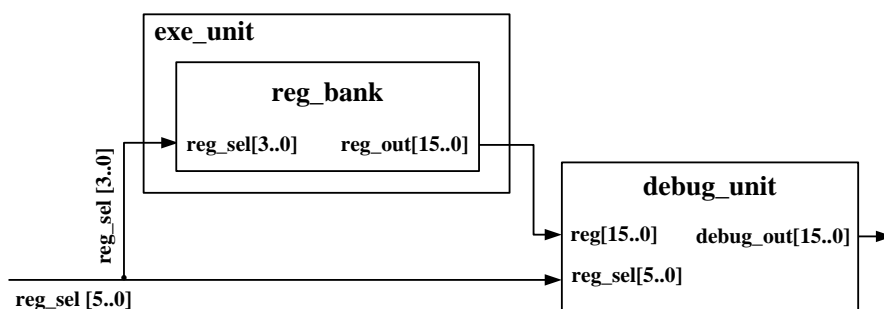
调试过程中，调试程序 debugcontroller 会在 reg_sel 输入端循环地输入 0~63，并将调试单元的输出在程序界面上进行显示。



寄存器堆和调试单元的连接关系:

将 reg_sel[5..0] 的最低四位 reg_sel[3..0] 送给寄存器堆的寄存器选

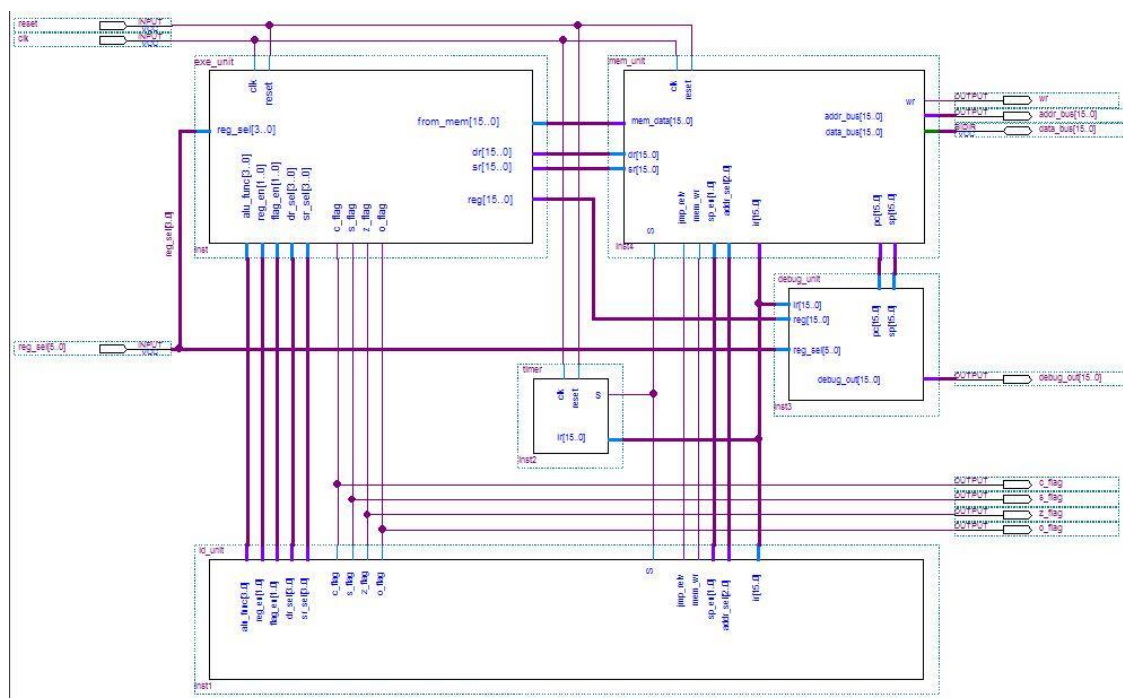
择端，将寄存器堆的输出端 reg_out 作为调试单元的输入端 reg。当 reg_sel[5..0] 的最高两位为“00”，而最低四位在 0 到 15 之间变化时，debug_out 输出的恰好是寄存器 R0 到 R15。



9、上机调试

上机调试步骤：

- 将 exe_unit, id_unit, mem_unit, timer, debug_unit 连接成完整的 CPU：



CPU 外部端口包括：

输入端：reset, clk, reg_sel[5..0]

输出端：addr_bus[15..0], debug_out[15..0], wr, c_flag, z_flag, s_flag, o_flag

输入输出端：data_bus[15..0]

共 61 个引脚。

芯片类型：Cyclone 家族的 EP1C6Q240C8

引脚分配：

reset: 240 clk: 29 wr: 75

c_flag: 86 z_flag: 85 s_flag: 83 o_flag: 84

reg_sel[5..0]: 17, 16, 15, 14, 13, 12

addr_bus[15..0]: 64, 63, 62, 61, 60, 59, 58, 57, 48, 47, 46, 45, 44, 43, 42, 41

```
data_bus[15..0]:
```

```
237, 236, 235, 234, 226, 225, 224, 223, 217, 216, 215, 214, 203, 202, 201, 200
```

```
debug_out[15..0]:
```

```
181, 180, 179, 178, 177, 175, 174, 173, 165, 164, 163, 162, 161, 160, 159, 158
```

然后将以上程序下载到实验箱芯片中；

➤ 设置实验箱：

SW22 接 USB，REGSEL 接 1，CLKSEL 接 0，FDSEL 接 1

➤ 调试步骤：

1. 打开规则文件
2. 打开汇编源文件
3. Compile Code
4. Upload BIN
5. CPU 复位
6. Begin Debug
7. 运行程序，可以选择半时钟周期运行、单时钟周期运行或运行至断点（双击某一行可以设置断点）。运行过程中通过 Reg Window 观察 CPU 内部各寄存器的值。
8. End Debug

四、实验心得：

通过对本门课程的学习，复习巩固了硬件描述语言 VHDL；掌握了工具 Quartus II 的使用；深入了解了 CPU 的基本结构和原理；掌握了测试和调试的步骤，收获很大。