简易计算机系统综合设计设计报告

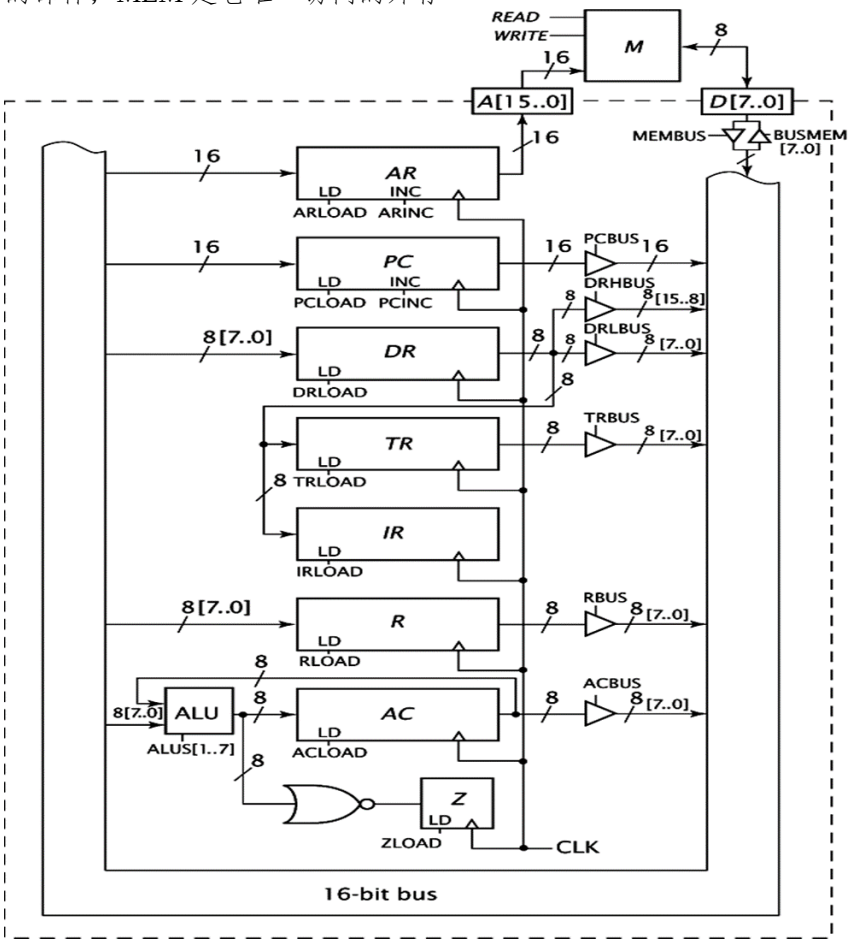
班级 物联 1601 姓名 蒋雨 学号 201611020126

一、设计目的

掌握用 vhdl 语言设计方法。并能完成部分 cpu 的指令。

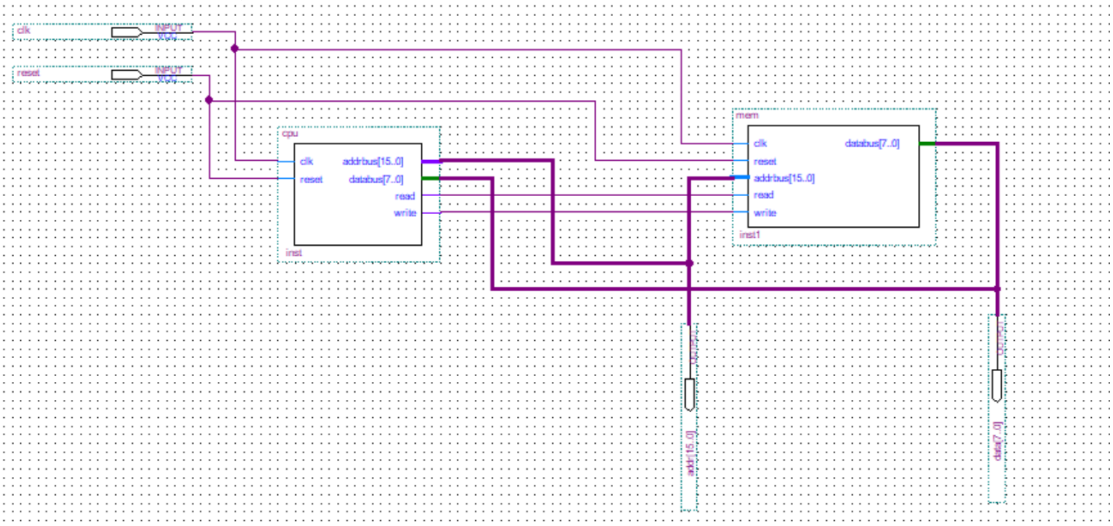
二、设计思路

根据数据通路和指令的每一个状态设计在相应的部件里，把这些寄存器当作 CPU 内部的部件，MEM 是它唯一访问的外存

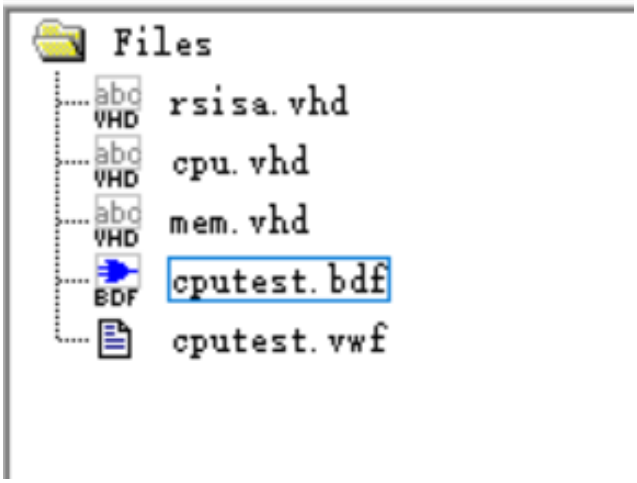


三、详细设计

3.1 设计的整体架构

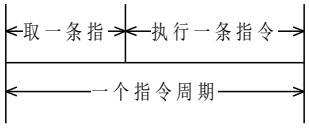


工程由 5 个文件组成，而 rsisa 声明每条指令对应的变量名，cpu 文件是完成 CPU 的内部组成、cpu 可能达到的各个状态，和 cpu 处于各个状态下采取的动作。mem 声明内存的大小、初始化内存，并规定 read、write 有效时内存相应的动作，cputest.bdf 文件就是顶层文件，把 cpu 和 mem 连接起来，进行相应的操作，vwf 文件是用于验证结果是否正确的



指令周期

指令周期与数据通路结构、指令执行方式有关。指令可以串行执行，也可以并行执行。本设计采用串行工作方式，即“读取—译码—执行—再读取……”。串行工作方式虽然工作速度和主机效率都要差一些，但它的控制简单。因此，本机指令周期可以确定为：

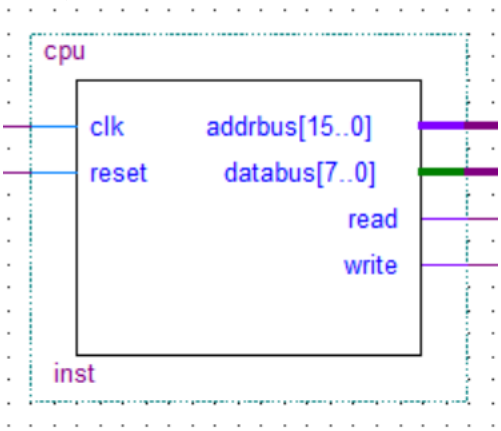



```
signal state: std_logic_vector(5 downto 0);

signal nextstate: std_logic_vector(5 downto 0);
```

由于此代码是用微操作状态转移的方法，一个指令不一定在一个时钟周期内周期完成
首先是取指令阶段：
FETCH1: AR<=PC
FETCH2: DR<=M PC<=PC+1
FETCH3: IR<=BUS AR<=PC

3.2 各模块的具体实现

CPU：



模块有 2 个输入接口，4 个输出接口
输入接口：

clk：控制时钟。
reset：复位操作。

输出接口：

read：当执行到需要内存的 read 指令执行时，相应的输出 0 或 1
write：当执行到需要内存的 write 指令执行时，相应的输出 0 或 1
addrbus:输出此时地址总线的地址
databus:输出此时数据总线上的值

功能实现：

首先是各个寄存器，在这里就把它定义为 CPU 内的信号，进行相应的赋值

```
architecture cpu_behav of cpu is


    signal pc: std_logic_vector(15 downto 0);
    signal ac: std_logic_vector(7 downto 0);
    signal r: std_logic_vector(7 downto 0) ;
    signal ar: std_logic_vector(15 downto 0);
    signal ir: std_logic_vector(7 downto 0);
    signal dr: std_logic_vector(7 downto 0);
    signal tr: std_logic_vector(7 downto 0);
    signal z: std_logic;
    signal thebus: std_logic_vector(15 downto 0);
```

然后是寄存器上的信号，比如 LOAD 信号等等也声明为变量
第三步是给相应的微操作赋予相应的操作码，如下：

```
constant fetch1:    std_logic_vector(5 downto 0) := "000000";-- ar<=pc
constant fetch2:    std_logic_vector(5 downto 0) := "000001";-- dr<=m  pc<=pc+1
constant fetch3:    std_logic_vector(5 downto 0) := "000010";--ir<=thebus ar<=pc
```

接下来就是重头戏，CPU 设计采用的是多进程的模式，下面三个进程就分别描述了时钟上升沿要进行的操作还有根据当前状态确定下一步，以及根据当前的状态使相应的控制信号有效

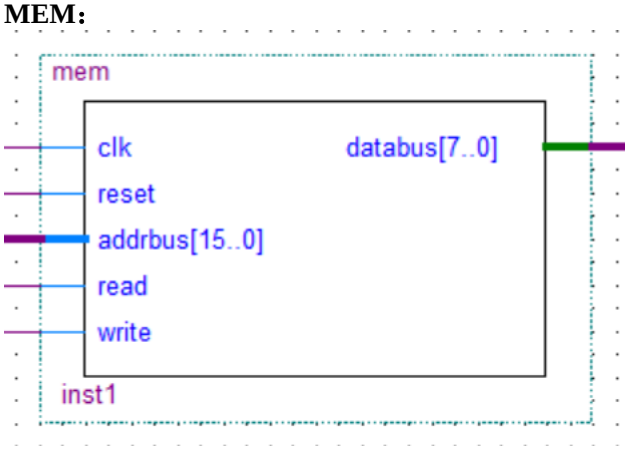
```
update_regs: process(clk)

gen_controls: process(state)

for_nextstate: process(state, ir, z)
```

rsisa.vhd：

```
constant RSNOP: std_logic_vector(7 downto 0) := "00000000";
constant RSLDAC: std_logic_vector(7 downto 0) := "00000001";
constant RSSTAC: std_logic_vector(7 downto 0) := "00000010";
constant RSMVAC: std_logic_vector(7 downto 0) := "00000011";
constant RSMOVR: std_logic_vector(7 downto 0) := "00000100";
constant RSJUMP: std_logic_vector(7 downto 0) := "00000101";
constant RSJMPZ: std_logic_vector(7 downto 0) := "00000110";
constant RSJPNZ: std_logic_vector(7 downto 0) := "00000111";

constant RSADD: std_logic_vector(7 downto 0) := "00001000";
constant RSSUB: std_logic_vector(7 downto 0) := "00001001";
constant RSINAC: std_logic_vector(7 downto 0) := "00001010";
constant RSCLAC: std_logic_vector(7 downto 0) := "00001011";
constant RSAND: std_logic_vector(7 downto 0) := "00001100";
constant RSOR: std_logic_vector(7 downto 0) := "00001101";
constant RSXOR: std_logic_vector(7 downto 0) := "00001110";
constant RSNOT: std_logic_vector(7 downto 0) := "00001111";
```

功能实现：
通俗的说就是把指令与操作对应然后打包给 CPU 使用

MEM：



RAM 5 个输入接口， 1 个输出接口
输入接口：
address[]：输入的地址
clock：控制 ram 的时钟
write：当它为 1 时，允许写操作
read：当 read 为 1 时，允许读操作
reset：当 reset 为 1 时，允许对 RAM 进行清零操作
输出接口：
databus[]：输出查询地址对应的数据
功能实现：
往 mem 里不同的地址写上相应的指令和数据，实现简单的从 1 加到 n
系统测试

4.1 测试环境

QUARTUS 和 GHDL 进行波形仿真

4.2 测试代码（在 mem 里）

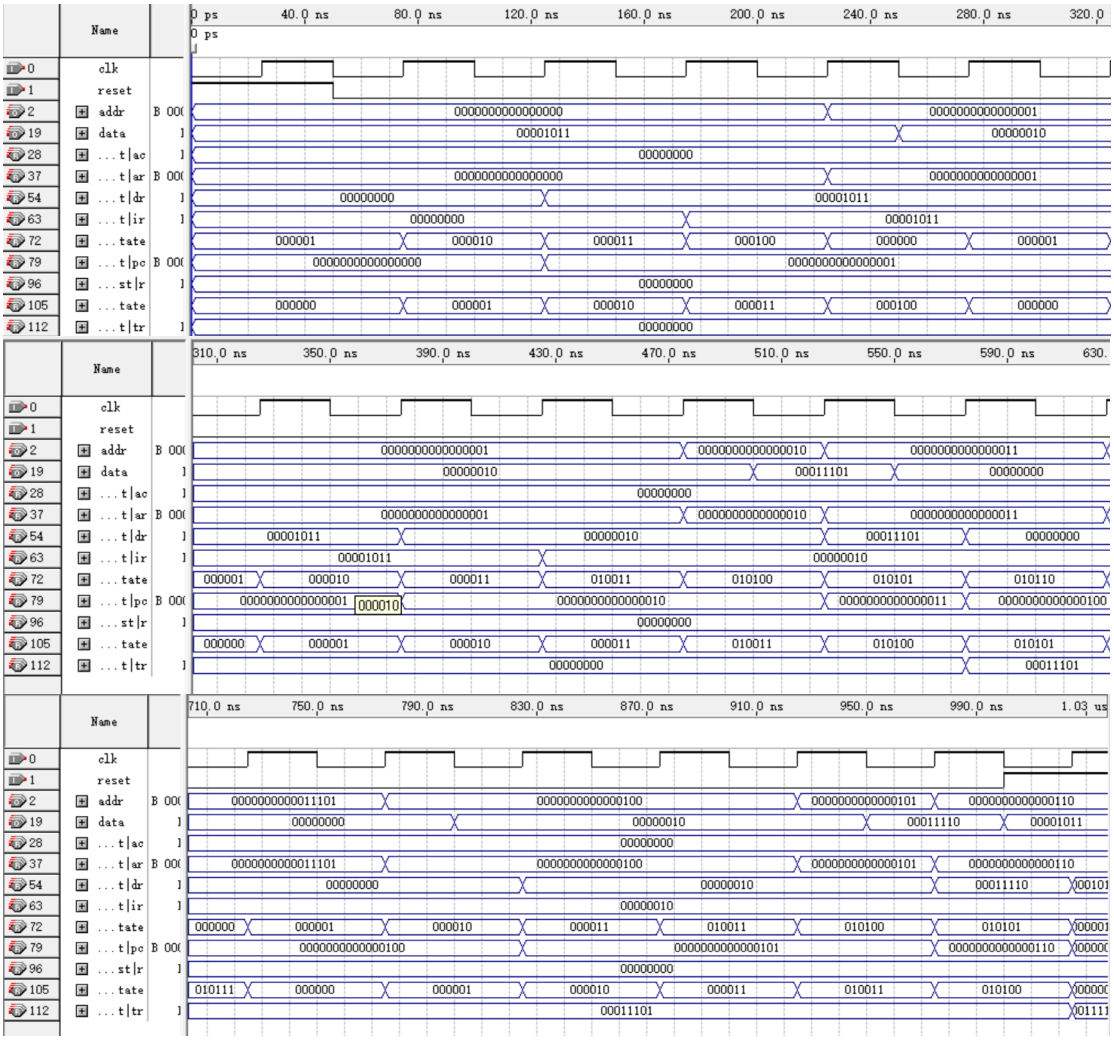
```
 0 => RSCLAC,

      1 => RSSTAC,--m[total_total]<=ac        total=0

      2 => std_logic_vector(to_unsigned(total_addr, 8)),

      3 => X"00",

      4 => RSSTAC,--m[i_addr]<=ac        i=0

      5 => std_logic_vector(to_unsigned(i_addr, 8)),

      6 => X"00",

      7 => RSLDAC,    -- loop        --ac<=m[i_addr]        ac=i

      8 => std_logic_vector(to_unsigned(i_addr, 8)),

      9 => X"00",

      10 => RSINAC, --ac++

      11 => RSSTAC, --i=ac

      12 => std_logic_vector(to_unsigned(i_addr, 8)),

      13 => X"00",

      14 => RSMVAC, --r=ac

      15 => RSLDAC, --ac=total

      16 => std_logic_vector(to_unsigned(total_addr, 8)),

      17 => X"00",

      18 => RSADD, --ac=ac+r

      19 => RSSTAC, --total=ac

      20 => std_logic_vector(to_unsigned(total_addr, 8)),

      21 => X"00",

      22 => RSLDAC, --ac=n

      23 => std_logic_vector(to_unsigned(n_addr, 8)),

      24 => X"00",

      25 => RSSUB, --ac=ac-r

      26 => RSJPNZ, --
```

```
        27 => std_logic_vector(to_unsigned(loop_addr, 8)),

        28 => X"00",

        29 => X"00",    -- total

        30 => X"00",    -- i

        31 => "00001000",    -- n

        others => RSNOP

);
databus <= memdata(to_integer(unsigned(addr))) when (write='0') else "ZZZZZZZZ";
result<=memdata(29);
```

4.3 测试结果

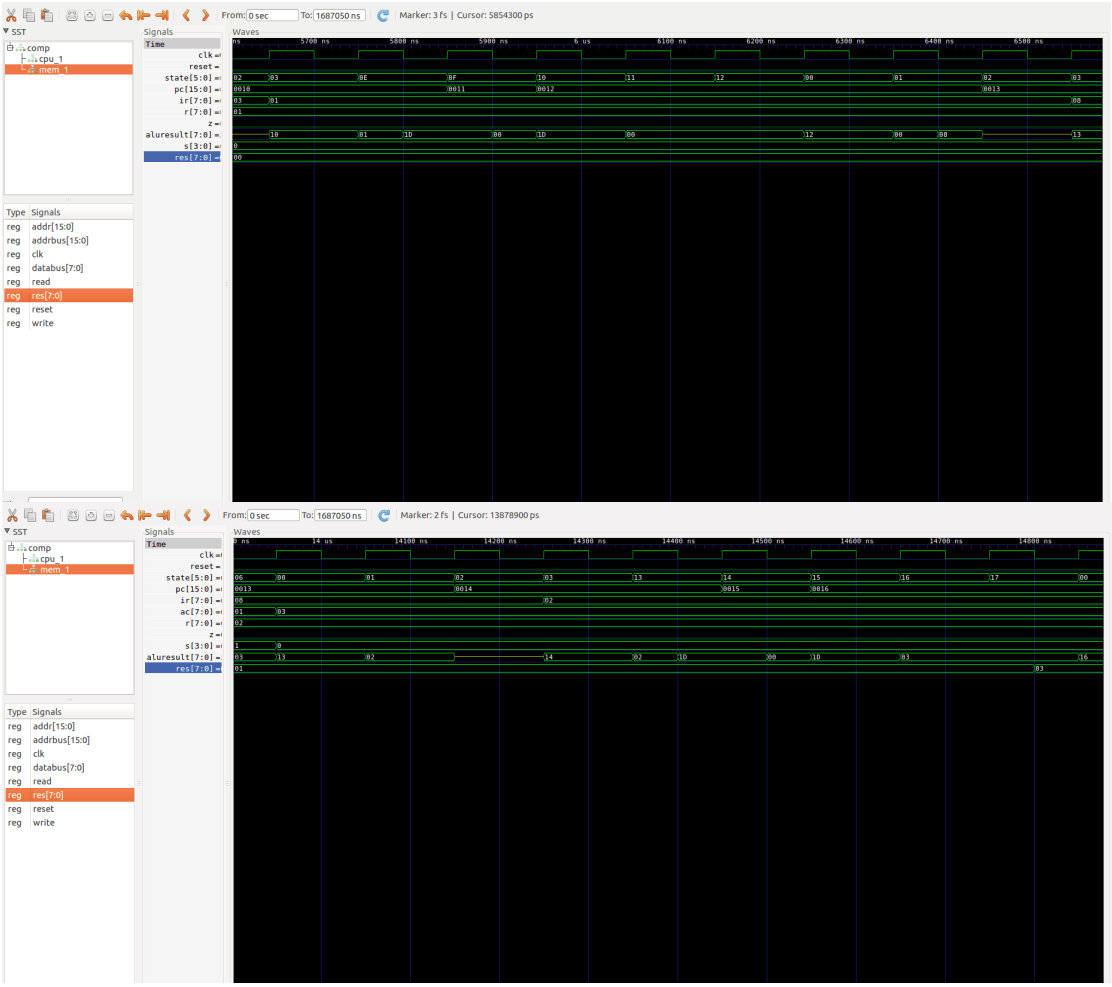
1.QUATUS 在这里测试的时候出现了一个问题，我明明把计算结果 result 变量添加波形仿真图里了，但是一仿真后就没有这个变量了，并且执行的时候明明根本执行不到 LDAC 那一步，我想把 endtime 设置久一点，但是我的 QUARTUS 超过 1 微秒就编译不出来了，一直显示在仿真中，所以我最后还用了 GHDL 进行验证，结果符合预期

Quartus 仿真结果如下：



GHDL 仿真结果如下：

从 result 就可以直观的看出结果



四、总结

以前学习数电的时候也设计过 CPU 但是那个时候是完全用数字逻辑的方式实现的，一个周期内就要完成指令所有的动作，而现在刚开始做的时候没什么头绪，后来参考了几位同学的代码觉得其实这种实现方式比以前的更简单，因为我们只需要根据当前状态来得到相应操作和控制信号就可以了，但是在最后仿真的时候遇到了问题，就是 AC 的值一直为 0，我想把结束时间拉长一点就一直编译不出来，最后还是用了 ghdl 进行验证，代码是没问题的，这个课程让我收获到很多，超级感谢吴强老师了，一直帮我们改错！