哈爾濱工業大學

Harbin Institute of Technology

课程设计说明书（论文）

课程名称：数字集成电路课程设计

设计题目：基于单周期 MIPS 的微控制器设计与实现

院　　系：微电子科学与技术系

班　　级：1821403

设 计 者：崔旭洋

学　　号：1182100506

指导教师：付方发

设计时间：2021 年 7 月 25 日-2021 年 7 月 30 日

哈尔滨工业大学

姓　名：　崔旭洋　院（系）：航天学院微电子科学与技术系

专　业：　微电子科学与工程　班　号：1821403

任务起至日期：　2021 年 7 月 25 日 至 2021 年 7 月 30 日

课程设计题目：

基于单周期 MIPS 的微控制器设计

已知技术参数和设计要求：

Load/store，算术逻辑运算，流程控制三部分是 RISC 处理器系统的主要组成部分，本设计以 MIPS 处理器的指令集为例，要求完成一个兼容 MIPS 指令集、具有 load word、store word、add、sub、and、or、slt、beq、j、jr、jal 等 11 条指令的单周期的微控器设计，该微控器可模拟计算机中 CPU 的取指、译码和执行操作。

设计要求：

1. 确定设计采用的单周期 MIPS 微控制器原型的算法
2. 确定设计的体系结构
3. 划分所确定的体系结构，画出模块图，确定模块间的连接关系，端口方向及宽度
4. 确定设计的测试方案、测试点及测试向量
5. 完成设计的 RTL 代码及测试代码
6. 完成设计的验证
7. 使用综合工具 DC，在 0.18 微米工艺条件下完成设计的综合，并给出设计的性能评价（面积、速度等）
8. FPGA 实现与测试，将完成的设计下载到 FPGA 上，并测试功能。（选做）
9. 编写课程设计报告
10. 成果演示与答辩

基本要求：

1. 确定设计采用的算法；
2. 确定设计的体系结构；
3. 划分所确定的体系结构，画出模块图，确定模块间的连接关系，端口方向及宽度；
4. 确定设计的测试方案、测试点及测试向量；
5. 完成设计的RTL代码及测试代码；
6. 完成设计的验证，给出设计的性能评价（面积、速度等）；
7. 撰写课程设计报告。

工作量：

本课程设计拟按照每 4 人为一组分工并协作完成。每位小组成员分别选择 1~4 题之一，作为该组同学的课程设计题目独立完成；在完成个人题目基础上小组成员共同完成第 5 题。

熟悉开发环境、学习工具使用：12 学时

分析题目、确定设计方案：12 学时

设计、验证以及性能评估、整理数据：36 学时

工作计划安排：

2021.7.24 -- 2021.7.25　学习MIPS处理器原理，分析设计题目

2021.7.26 -- 2021.7.27　利用Verilog语言进行系统设计、验证

2021.7.28 –2021.7.29　性能评估、整理数据

2021.7.30 结题答辩，撰写课程设计报告

同组设计者及分工：

崔旭洋：微控制器 RTL 代码全编写，帮助其他人完成代码编写，并负责成果演示与答辩

崔添：存储器部分代码主编写，并负责答辩 PPT 制作

李司：进行 DC 综合，参与微控制器 RTL 代码改错

徐祥升：仿真代码主编写，并制作报告中的流程图和表格

指导教师签字_____

年　　月　　日

教研室主任意见：

教研室主任签字_____

年　　月　　日

***注：此任务书由课程设计指导教师填写。**

一、功能描述

设计采用 Verilog 代码编写 16 位 MIPS 指令集的单周期 CPU，该 CPU 可以对如下指令进行操作，运算，其中为了减少与内存通信的次数，与其交互的指令仅有 SW 存数和 LW 取数指令：

指令	*第15位*	*第14位*	*第13位*	*第12位*	*第11位*	*第10位*	*第9位*	*第8位*	*第7位*	*第6位*	*第5位*	*第4位*	*第3位*	*第2位*	*第1位*	*第0位*	*具体功能描述(参照经典32位指令进行缩减)*
R类	*[3:0]OP*				*[2:0]rs*			*[2:0]rt*			*[2:0]rd*			*[2:0]funct*			*功能*
add	*0000*				*rs*			*rt*			*rd*			*0*	*0*	*0*	*rd=rs+rt*
addu	*0000*				*rs*			*rt*			*rd*			*0*	*0*	*1*	*rd=rs+rt(u)*
sub	*0000*				*rs*			*rt*			*rd*			*0*	*1*	*0*	*rd=rs-rt*
subu	*0000*				*rs*			*rt*			*rd*			*0*	*1*	*1*	*rd=rs-rt(u)*
slt	*0000*				*rs*			*rt*			*rd*			*1*	*0*	*0*	*rd=(rs<rt)?1:0*
and	*0000*				*rs*			*rt*			*rd*			*1*	*0*	*1*	*rd=rs&rt*
or	*0000*				*rs*			*rt*			*rd*			*1*	*1*	*0*	*rd=rs|rt*
jr	*0000*				*rs*			*rt*			*rd*			*1*	*1*	*1*	*PC=rs*
I类	*[3:0]OP*				*[2:0]rs*			*[2:0]rt*			*[5:0]imm*						*功能*
addi	*1000*				*rs*			*rd*			*im*						*rd=rs+im*
subi	*1001*				*rs*			*rd*			*im*						*rd=rs+im(u)*
andi	*1010*				*rs*			*rd*			*im*						*rd=rs&im*
ori	*1011*				*rs*			*rd*			*im*						*rd=rs|im*
lw	*1100*				*rs*			*rd*			*im*						*rd=mem[rs+im]*
sw	*1101*				*rs*			*rd*			*im*						*mem[rs+im]=rd*
beq	*1110*				*rs*			*rd*			*im*						*PC=(rs==rd)?(PC+2+im<<1):PC*
bne	*1111*				*rs*			*rd*			*im*						*PC=(rs!=rd)?(PC+2+im<<1):PC*
J类	*[3:0]OP*				*[11:0]addr*												*功能*
j	*0100*				*addr*												*PC={(PC+2)[15:12],addr}*
jal	*0101*				*addr*												*R7=PC,PC={(PC+2)[15:12],addr}*

其中由于指令位数的限制，I 类指令的立即数范围为 0~63,J 类指令的跳转地址范围为 0x000~0xFFF。

对于配套的内存，此 CPU 仅适应经典的 1 字节 1 地址的存储器，且在通信的时候需要遵

守规定的规范，CPU 和内存可以采用异步时钟，但读数据和写数据的时候均需要信号握手才可以传输成功。

CPU 可以访问的地址为 16 位，即从 0x0000~0xFFFF，传输数据和写数据的接口均为 8 位，内部含有 8 个 16 位的通用寄存器组 R0~R7，以及 16 位的程序计数器 PC，16 位的指令寄存器 IR。CPU 通过两位的 mem_ctrl 信号控制外部内存的读和写，高位控制写，低位控制读，均为 1 有效，0 失效，且在每条指令执行完成后将会传出 done 完成信号。CPU 通过 rst_n 同步清零，0 有效；cpu_en 信号控制 CPU 的使能，1 有效，但当其为 0 的时候也具有复位 CPU 的功能，复位后的 CPU 将清零所有寄存器和状态，即无法继续执行之前的指令，需要重新执行，CPU 的接口如下所示：

```
module MIPS(
    input clk,
    input rst_n,
    input cpu_en,
    input [7:0] data_read,
    input ram_send,
    input ram_receive,
    output reg cpu_send,
    output reg cpu_receive,
    output reg cpu_ready,
    output reg [15:0] data_addr,
    output reg [7:0] data_store,
    output reg [1:0] mem_ctrl,//1:1 for read,0:0 for write
    output reg done
    );
```

二、设计方案

由于设计的 CPU 是单周期处理指令的，不需要实现流水线功能，故设想使用状态机来控制 CPU 进行操作和执行指令，和经典的 MIPS 指令集的 32 位 CPU 一样，CPU 将指令的操作划分为 IF 取指令，ID 译码，EX 执行，MEM 内存交互和 WB 更新寄存器 5 个阶段，同时，为了使 CPU 能连续运行，在 IF 阶段前加入一个 IDLE 阶段对所有寄存器和中间信号清零，具体各状态的转换如下。

（1）IF 阶段

主要实现 CPU 从内存中读取指令的功能，由于需要与外部设备通信且可能使用异步时钟，为了保证数据的有效性的正确性，如下定义传输数据的过程：

对于 CPU 写数据，即 CPU 向 RAM 传输数据，定义 cpu_send 和 ram_receive 两根控制线，且由 mem_ctrl 的低位控制 ram 开始写，当 cpu 把需要写的数据挂到传输线 data_store 上后拉高 cpu_send，代表此时数据有效，此时 ram 检测到数据有效后立即开始写操作，当 ram 真正把数据从传输线上写到寄存器里的时候 ram_receive 拉高，代表 ram 完成写操作，当且

仅当 cpu_send 和 ram_receive 同时为高时握手成功，代表这一次 CPU 写操作完成，两控制线在下个时钟周期分别清零，具体的时序图如下所示：

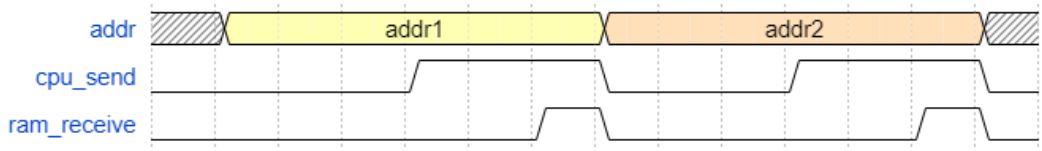


图 1 写操作时序图

对于 CPU 读数据，即 RAM 向 CPU 传输数据，过程和 CPU 写类似，定义了 cpu_receive 和 ram_send 两根控制线，且由 mem_ctrl 的高位控制 ram 开始读，但由于 ram 没有办法向 CPU 一样做到流程控制，在多次读取时可能会出现重复读取导致传输线上的数据不正确的情况，另外加入 cpu_ready 和 avaliable 信号，当 CPU 准备好读取的时候将 cpu_ready 拉高一个时钟周期，形成一个短脉冲，当 ram 端检测到 cpu_ready 上升沿的时候将 avaliable 信号拉高，代表可以开始将寄存器内的值转移到传输线上，而此后当 ram 检测到 cpu_receive 信号为低电平的时候，即 cpu 完成一次读操作时，将 avaliable 拉低防止第二次读取。其他的控制与写操作的相同，当 ram 把需要写的数据挂到传输线 data_read 上后拉高 ram_send，代表此时数据有效，此时 cpu 检测到数据有效后立即开始写操作，当 cpu 真正把数据从传输线上写到寄存器里的时候 cpu_receive 拉高，代表 cpu 完成读操作，当且仅当 cpu_receive 和 ram_send 同时为高时握手成功，代表这一次 CPU 读操作完成，两控制线在下个时钟周期分别清零，具体的时序图如下所示：

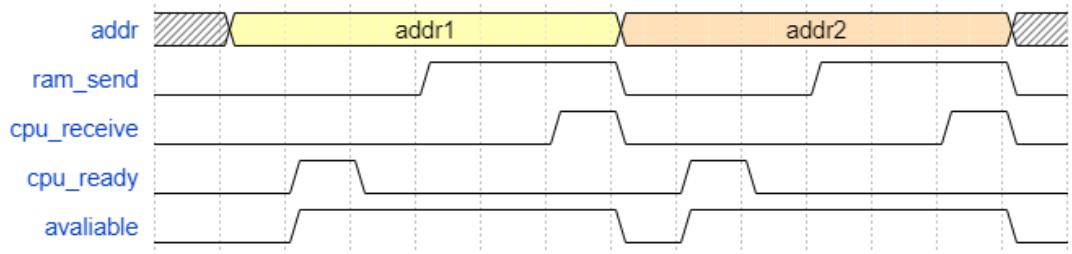


图 2 读操作时序图

由于设计的为 16 位的 CPU，其指令均是 16 位的，而实际中的存储器都是 1 地址 1 字节存储的，故若想要将指令完整的读出来则需要连续两次 CPU 读操作，故设计通过变量 read_cnt 来使用一个小的类状态机控制读的过程，当其为 0 时读第一次，读完第一次后变为 1，开始读第二次，当读完第二次后变为 2，清零控制，转到译码阶段，其中若读的控制信号没有握手成功则不改变状态，继续执行未完成的读操作。并在完成 IF 取值后更新 PC 的值，将其+2，且指令存到 16 位的 IR 指令寄存器中，与 ram 的连线如下所示：

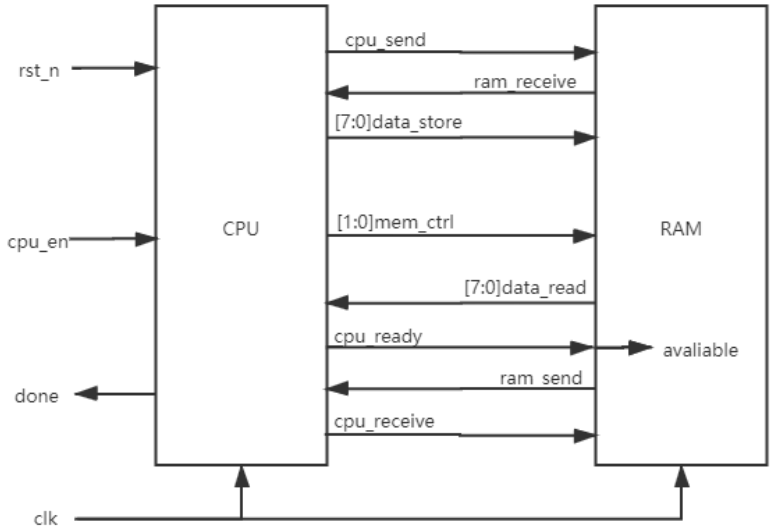


图 3 CPU 和 RAM 连线示意

状态内的判断如下所示：

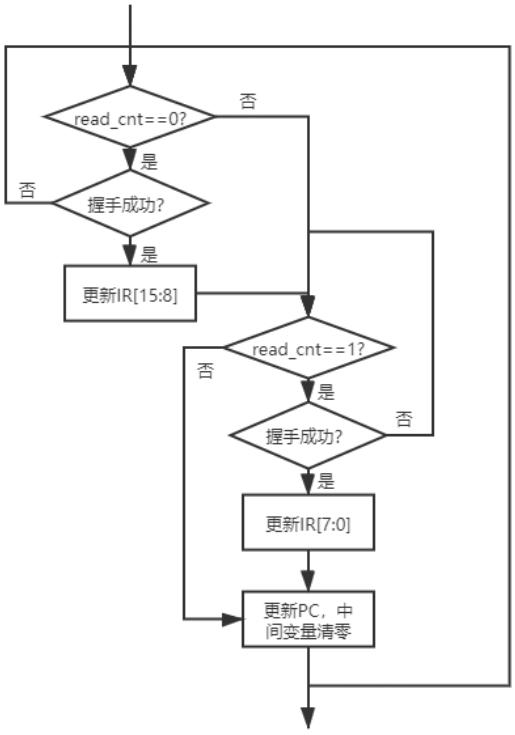


图 4　IF 阶段流程图

（2）ID 阶段

译码阶段主要完成的是指令的翻译，首先根据指令定义的格式取其首 4 位，即[15:12]IR 为 OP，再根据 OP 的值判断是 R 类，I 类还是 J 类指令，根据不同指令的格式取得对应的 im＝IR[5:0]，addr＝IR[11:0]，funct＝IR[2:0]和更新 rt，rs 的值（对通用寄存器组进行译码），

为了加快译码速度，这里不加入判断条件，直接更新所有中间变量的值，由后续 EX 执行过程再判断究竟取哪些寄存器里的值，此阶段的流程如下所示：

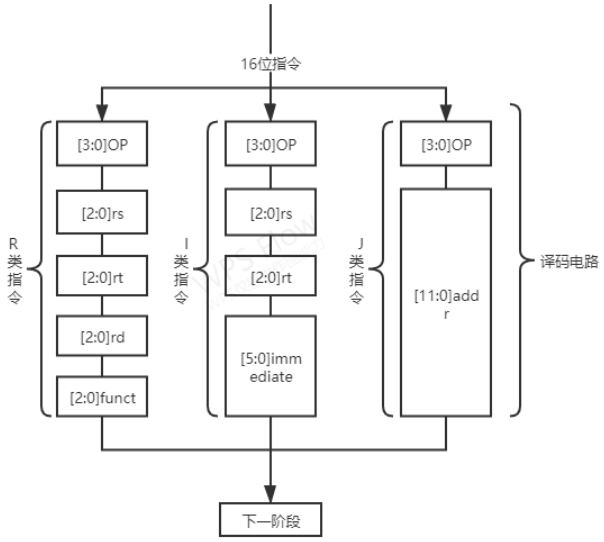


图 5　ID 阶段流程图

（3）EX 阶段

执行阶段主要根据指令的类型对寄存器进行操作，具体的流程图如下所示：

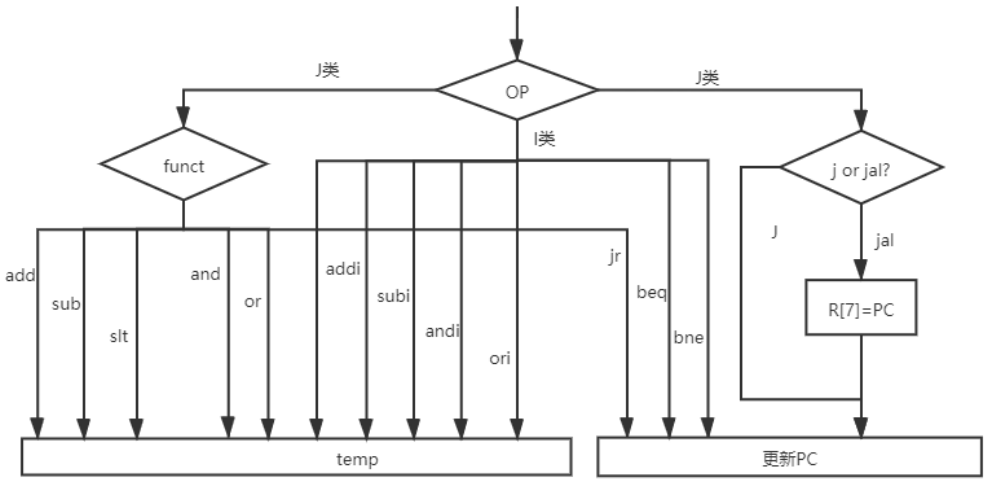


图 6　EX 阶段流程图

对于运算类指令，将结果暂存到 temp 内，在 WB 阶段更新对应的寄存器的值，而对于跳转类指令，即 jr，j，jal，beq，bne，则对 PC 的值进行更新，若是需要对内存进行操作的 LW 和 SW，则清零 temp，保持 PC，并直接跳转到 MEM 阶段

（4）MEM 阶段

MEM 阶段是 CPU 对 RAM 的操作，首先通过 OP 判断是 LW 还是 SW 指令，然后遵循

与 IF 阶段相同的通信原则连续写或者读两次 RAM，该阶段的流程图如下所示：

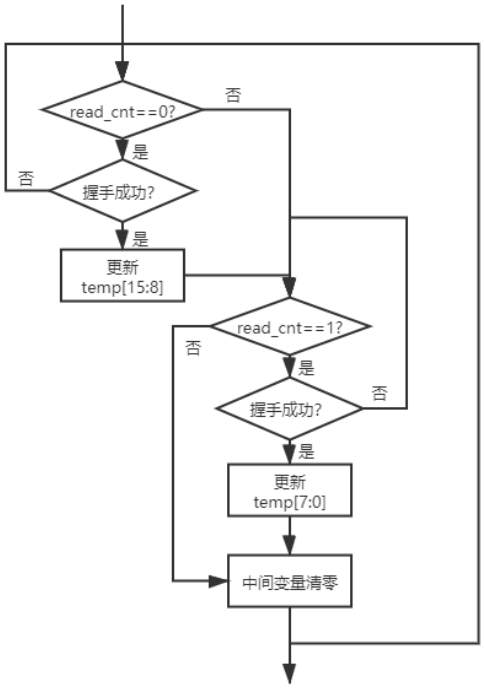


图 7-1　LW 流程图

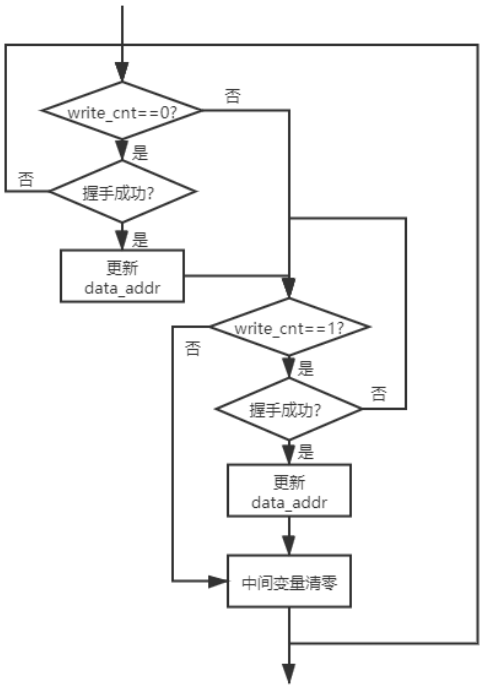


图 7-2　SW 流程图

若既不是 SW 也不是 LW 指令，则置零 mem_ctrl，并保持 temp 的值，清零传输线上的值，然后将状态跳转到 WB

（5）WB 阶段

WB 阶段会根据指令的类型用 temp 更新对应通用寄存器组的值，具体流程如下所示：

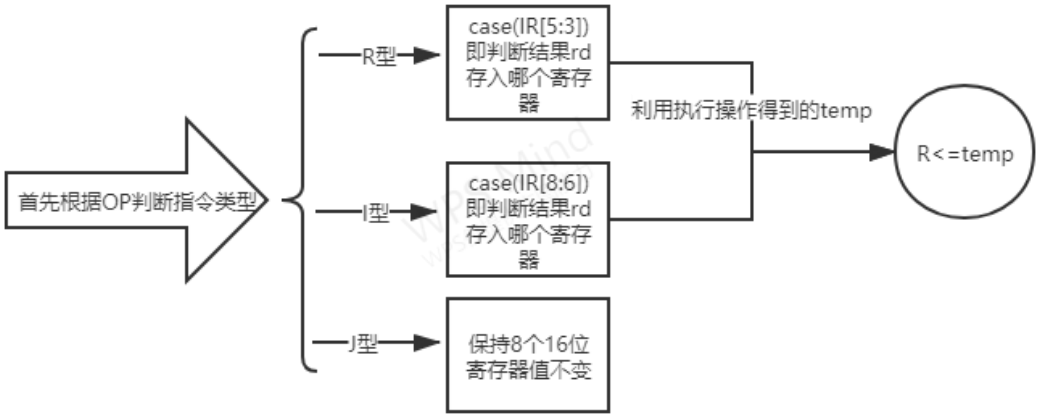


图 8 WB 阶段流程图

三、设计代码

根据之前通信的规则和状态转换设计出如下端口和状态：

```
module MIPS(
    input clk,
    input rst_n,
    input cpu_en,
    input [7:0] data_read,
    input ram_send,
    input ram_receive,
    output reg cpu_send,
    output reg cpu_receive,
    output reg cpu_ready,
    output reg [15:0] data_addr,
    output reg [7:0] data_store,
    output reg [1:0] mem_ctrl,//1:1 for read,0:0 for write
    output reg done
    );

    reg [15:0] R [7:0];//register group
    reg [15:0] PC,IR;
    reg [3:0] OP;
    reg [2:0] funct;//[2:0]for real funct

    reg [2:0] state,next_state;
    parameter IDLE = 3'b000,
              IF = 3'b001,
              ID = 3'b010,
              EX = 3'b011,
              MEM = 3'b100,
              WB = 3'b101;
```

流程控制中的状态机同步清零如下所示：

```
    always@(posedge clk)
    begin
       if(~(rst_n&cpu_en))
       begin
          {R[0],R[1],R[2],R[3],R[4],R[5],R[6],R[7]} <= 128'd0;
          {PC,IR} <= 32'd0;
          {rs,rt} <= 32'd0;
          im <= 6'd0;
          addr <= 12'd0;
          temp <= 16'd0;
          {cpu_receive,cpu_send} <= 2'b00;
          next_state <= IDLE;
          data_addr <= 16'd0;
          data_store <= 8'd0;
          OP <= 4'd0;
          IR <= 16'd0;
          funct <= 3'd0;
          //hold_MEM <= 1'b0;
          done <= 1'b0;
          mem_ctrl <= 2'd0;//disable
          read_cnt <= 2'd0;
          write_cnt <= 2'd0;
       end
```

在 IDLE 状态对后面操作的控制变量和完成信号清零：

```
       else
       begin
          case(state)
             IDLE:
             begin
                done <= 1'b0;
                next_state <= IF;
                read_cnt <= 2'd0;
                write_cnt <= 2'd0;
                {cpu_receive,cpu_send} <= 2'b00;
                mem_ctrl <= 2'b00;//invalid
             end
```

取值阶段连续对 RAM 进行两次读的操作：

```
IF:
begin
   mem_ctrl <= 2'b10;//read
   {cpu_receive,cpu_send} <= 2'b00;//m

   if(read_cnt == 2'd0)
   begin
      data_addr <= PC;
      delay_en <= 1'b1;

      //signal receive control part
      if(cpu_receive && ram_send)//pri
      begin
         cpu_receive <= 1'b0;
         delay_en <= 1'b0;
         read_cnt <= 2'd1;
      end
      else if(ram_send)//cpu read cont
      begin
         IR[15:8] <= data_read;
         if(IR[15:8] == data_read)
            cpu_receive <= 1'b1;
         else
            cpu_receive <= 1'b0;
      end
      else
      begin
         IR[15:8] <= IR[15:8];
         read_cnt <= 2'd0;
      end
   end

   else if(read_cnt == 2'd1)
   begin
      data_addr <= PC + 16'd1;

      delay_en <= 1'b1;

      //signal receive control part
      if(cpu_receive && ram_send)//pr
      begin
         cpu_receive <= 1'b0;
         delay_en <= 1'b0;
         read_cnt <= 2'd2;
      end
      else if(ram_send)//cpu read con
      begin
         IR[7:0] <= data_read;
         if(IR[7:0] == data_read)
            cpu_receive <= 1'b1;
         else
            cpu_receive <= 1'b0;
      end
      else
      begin
         IR[7:0] <= IR[7:0];
         read_cnt <= 2'd1;
      end
   end

   else if(read_cnt == 2'd2)
   begin
      next_state <= ID;
      PC <= PC + 16'd2;
      read_cnt <= 2'd0;
      mem_ctrl <= 2'b00;//disable
      {cpu_receive,cpu_send} <= 2'b00;
   end

   else
   begin
      next_state <= IF;
      PC <= PC;
   end
```

在阶段开始即对通信控制信号清零，并通过 read_cnt 控制进行读操作的次数。其在读取开始的时候需要产生一个 cpu_ready 信号防止重复读取，脉冲信号的产生如下所示：

```
always@(posedge clk)//pulse generator
begin
   if(~(rst_n&cpu_en))
   begin
      cpu_ready <= 1'b0;
      delay_cnt <= 1'b0;
   end
   else if(delay_en)
   begin
      if(delay_cnt)
         cpu_ready <= 1'b0;//cpu_ready control part
      else
      begin
         cpu_ready <= 1'b1;
         delay_cnt <= 1'b1;
      end
   end
   else
   begin
      cpu_ready <= 1'b0;
      delay_cnt <= 1'b0;
   end
end
```

在每个读操作中将控制信号握手清零设为最高优先级，用 if 写在前面并进行 read_cnt 的状态转换，然后判断传输线上的数据是否有效，若有效则更新 IR 部分位的值，如果没有握手成功则保持 read_cnt 的值直到握手成功。在整个 IF 取值完成后将状态跳转到 ID，并把 PC+2 准备执行下一条指令，最后清零所有控制信号。

ID 阶段则直接根据已有格式对指令做译码，为了加快译码速度这里不再判断指令的类别，直接使用组合电路加载寄存器的值，通用寄存器组的编码如下所示：

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7
R[0]	R[1]	R[2]	R[3]	R[4]	R[5]	R[6]	R[7]
3’b000	3’b001	3’b010	3’b011	3’b100	3’b101	3’b110	3’b111

然后在后面 EX 阶段选取需要的寄存器的值：

```
ID:
begin
   OP <= IR[15:12];
   case(IR[11:9])
      3'd0:rs <= R[0];
      3'd1:rs <= R[1];
      3'd2:rs <= R[2];
      3'd3:rs <= R[3];
      3'd4:rs <= R[4];
      3'd5:rs <= R[5];
      3'd6:rs <= R[6];
      3'd7:rs <= R[7];
      default:rs <= 16'd0;
   endcase
   case(IR[8:6])
      3'd0:rt <= R[0];
      3'd1:rt <= R[1];
      3'd2:rt <= R[2];
      3'd3:rt <= R[3];
      3'd4:rt <= R[4];
      3'd5:rt <= R[5];
      3'd6:rt <= R[6];
      3'd7:rt <= R[7];
      default:rt <= 16'd0;
   endcase
   funct <= IR[2:0];
   im <= IR[5:0];
   addr <= IR[11:0];
   next_state <= EX;
end
```

执行阶段则根据指令类型进行相应的运算，更新 temp 暂存器的值或者 PC 的值，由于 MIPS 指令集的 CPU 不需要标志寄存器，而且数据在内存中存放是以补码的形式存放，在执行的过程中不再区分有符号和无符号操作，直接进行运算，而对于立即数或者跳转指令的执行，则对立即数进行位扩展到 16 位后再运算：

```
EX:
begin
   if(OP == 4'b0000)//R sort
   begin
      case(funct)
         3'b000,3'b001:temp <= rs + rt;//add or addu
         3'b010,3'b011:temp <= rs - rt;//sub or subu
         3'b100:temp <= (rs < rt) ? 16'd1:16'd0;//slt
         3'b101:temp <= rs & rt;//and
         3'b110:temp <= rs | rt;//or
         3'b111:PC <= rs;//jr
         default:temp <= 16'd0;
      endcase
   end
   else if(OP[3])//I sort
   begin
      case(OP)
         4'b1000:temp <= rs + {10'd0,im};//addi
         4'b1001:temp <= rs - {10'd0,im};//subi
         4'b1010:temp <= rs & {10'd0,im};//andi
         4'b1011:temp <= rs | {10'd0,im};//ori
         4'b1110:PC <= (rs==rt) ? (PC+16'd2+im<<1):PC;//beq
         4'b1111:PC <= (rs!=rt) ? (PC+16'd2+im<<1):PC;//bne
         default:temp <= 16'd0;
      endcase
   end

   else if(OP[2])//J sort
   begin
      if(OP == 4'b0100)//j
         PC <= {PC_1[15:12],addr};
      else if(OP == 4'b0101)//jal
      begin
         R[7] <= PC;
         PC <= {PC_1[15:12],addr};
      end
   end
   else
      next_state <= IDLE;

   next_state <= MEM;
   read_cnt <= 2'd0;
   write_cnt <= 2'd0;
   data_addr <= 16'd0;
end
```

MEM 阶段根据 OP 的数据进行相应的 SW 和 LW 操作，其中 LW 的操作与 IF 中的读指令操作类似，只不过将存放的地方从 IR 变为 temp，而 SW 的操作也比较类似，不过由于可以使用 write_cnt 控制流程，不再需要 cpu_ready 信号防止重写入，只需要在握手成功后清零控制信号然后及时跳转即可，LW 相关代码如下所示：

```
MEM:
begin
   case(OP)
      4'b1100://lw
         begin
            mem_ctrl <= 2'b10;//read
            {cpu_receive,cpu_send} <= 2'b00;//
            if(read_cnt == 2'd0)
            begin
               data_addr <= rs + {10'd0,im};
               delay_en <= 1'b1;

               //signal receive control part
               if(cpu_receive && ram_send)//pr
               begin
                  cpu_receive <= 1'b0;
                  delay_en <= 1'b0;
                  read_cnt <= 2'd1;
               end
               else if(ram_send)//cpu read con
               begin
                  temp[15:8] <= data_read;
                  if(temp[15:8] == data_read)
                     cpu_receive <= 1'b1;
                  else
                     cpu_receive <= 1'b0;
               end
               else
               begin
                  temp[15:8] <= temp[15:8];
                  read_cnt <= 2'd0;
               end
            end
            else if(read_cnt == 2'd1)
            begin
               data_addr <= rs + {10'd0,im} + 16'd1;
               delay_en <= 1'b1;

               //signal receive control part
               if(cpu_receive && ram_send)//priority
               begin
                  cpu_receive <= 1'b0;
                  delay_en <= 1'b0;
                  read_cnt <= 2'd2;
               end
               else if(ram_send)//cpu read control pa
               begin
                  temp[7:0] <= data_read;
                  if(temp[7:0] == data_read)
                     cpu_receive <= 1'b1;
                  else
                     cpu_receive <= 1'b0;
               end
               else
               begin
                  temp[7:0] <= temp[7:0];
                  read_cnt <= 2'd1;
               end
            end

            else if(read_cnt == 2'd2)
            begin
               next_state <= WB;
               read_cnt = 2'd0;
               mem_ctrl <= 2'b00;//disable
               {cpu_receive,cpu_send} <= 2'b00;
            end

            else
            begin
               next_state <= MEM;
               temp <= temp;
            end

         end
```

SW 相关代码如下所示：

```
4'b1101://sw
   begin
      mem_ctrl <= 2'b01;//write
      {cpu_receive,cpu_send} <= 2'b00;//must be
      if(write_cnt == 2'd0)
      begin
         data_addr <= rs + {10'd0,im};
         data_store <= rt[15:8];

         //signal send control part
         if(cpu_send && ram_receive)
         begin
            cpu_send <= 1'b0;
            write_cnt <= 2'd1;
         end
         else if(data_store == rt[15:8])
            cpu_send <= 1'b1;
         else
         begin
            cpu_send <= 1'b0;
            write_cnt <= 2'd0;
         end
      end

      else if(write_cnt == 2'd1)
      begin
         data_addr <= rs + {10'd0,im} + 16'd1;
         data_store <= rt[7:0];

         //signal send control part
         if(cpu_send && ram_receive)
         begin
            cpu_send <= 1'b0;
            write_cnt <= 2'd2;
         end
         else if(data_store == rt[7:0])
            cpu_send <= 1'b1;
         else
         begin
            cpu_send <= 1'b0;
            write_cnt <= 2'd1;
         end
      end

      else if(write_cnt == 2'd2)
      begin
         next_state <= WB;
         write_cnt <= 2'd0;
         mem_ctrl <= 2'b00;//disable
         {cpu_receive,cpu_send} <= 2'b00;
      end

      else
      begin
         next_state <= MEM;
         write_cnt <= write_cnt;
      end
   end
```

WB 阶段主要对寄存器进行更新，根据 OP 的数据判断指令的类别，再使用组合逻辑将 temp 的值更新到对应的寄存器里中去，而对于 J 类指令，则保持寄存器组的值，在所有寄存器值更新完成后拉高 done，代表指令处理完成，默认调回 IDLE 准备执行下一条指令。

```
begin
   if(OP == 4'b0000)//R sort
   begin
      case(IR[5:3])
         3'd0:R[0] <= temp;
         3'd1:R[1] <= temp;
         3'd2:R[2] <= temp;
         3'd3:R[3] <= temp;
         3'd4:R[4] <= temp;
         3'd5:R[5] <= temp;
         3'd6:R[6] <= temp;
         3'd7:R[7] <= temp;
      endcase
   end
   else if(OP[3])//I sort
   begin
      if((OP[3:2] == 2'b10) | (OP == 4'b1100))//write to rt
      begin
         case(IR[8:6])
            3'd0:R[0] <= temp;
            3'd1:R[1] <= temp;
            3'd2:R[2] <= temp;
            3'd3:R[3] <= temp;
            3'd4:R[4] <= temp;
            3'd5:R[5] <= temp;
            3'd6:R[6] <= temp;
            3'd7:R[7] <= temp;
         endcase
      end
   end
   else//J sort + remaining I sort
      {R[0],R[1],R[2],R[3],R[4],R[5],R[6],R[7]} <= {R[0],R[1],R[2],R[3],R[4],R[5],R[6],R[7]};
   done <= 1'b1;
   next state <= IDLE;
```

四、验证方案及结果分析

在任务中我完成了 python 程序的编写以及辅助完成了仿真代码的编写，验证方案主要是在 testbench 文件中加入设计的存储器，在每个 rst_n 清零时使用$readmemh 系统任务对虚拟地址加载，而输入的导入则是通过在 txt 文件中根据格式编写汇编程序，然后使用 python 将其自动翻译成 1 地址 1 字节存储格式的机器指令代码到另一个 txt 文件中，通过之前的 $readmemh 将文件读入，初始化存储器，ram 的编写则需遵循之前的通信规则，根据响应的信号进行握手并更新传输线上的值，翻译汇编程序的 python 代码如下所示：

```
import os
import numpy as np
import shutil

dict_funct = {'ADD':'000','ADDU':'001','SUB':'010','SUBU':'011','SLT':'100','AND':'101','OR':'110','JR':'111'}
dict_OP = {'ADDI':'1000','SUBI':'1001','ANDI':'1010','ORI':'1011','LW':'1100','SW':'1101','BEQ':'1110','BNE':'1111','J':'0100','JAL':'0101'}
dict_reg = {'R0':'000','R1':'001','R2':'010','R3':'011','R4':'100','R5':'101','R6':'110','R7':'111'}

sort_I = {'ADDI','SUBI','ANDI','ORI','LW','SW','BEQ','BNE'}
sort_J = {'J','JAL'}
mif = open('D:\\test.txt',"w")
text = open('D:\\instruction.txt',"r")
str_temp = text.readlines()
addr = 0
```

```
for i in str_temp:
    temp = i.split()
    machine = []
    if temp[0] in dict_funct:
        if temp[0] != 'JR':
            machine.append('0000')
            machine.append(dict_reg[temp[1]])
            machine.append(dict_reg[temp[2]])
            machine.append(dict_reg[temp[3]])
            machine.append(dict_funct[temp[0]])
        else:
            machine.append('0000')
            machine.append(dict_reg[temp[1]])
            machine.append('000')
            machine.append('000')
            machine.append('111')
    elif temp[0] in sort_I:
        machine.append(dict_OP[temp[0]])
        machine.append(dict_reg[temp[1]])
        machine.append(dict_reg[temp[2]])
        imm = str(bin(int(temp[3]))).strip().split('0b')[-1]
        while len(imm) < 6:
            imm = '0' + imm
        machine.append(imm)
    elif temp[0] in sort_J:
        machine.append(dict_OP[temp[0]])
        data_addr = str(bin(int(temp[1]))).strip().split('0b')[-1]
        while len(data_addr) < 12:
            data_addr = '0' + data_addr
        machine.append(data_addr)

    else:
        print("\n\nerror!\n\n")
    string = ''
    for j in machine:
        string = string + j
    print(string)
    hex1 = 0
    hex2 = 0
    for j in range(0,8):
        hex1 = hex1 + ( (int(string[j])) << (7-j) )
    for j in range(8,16):
        hex2 = hex2 + ( (int(string[j])) << (15-j) )
    print("%x,%x"%(hex1,hex2))

    mif.writelines(str(hex(hex1)).strip().split('0x'))
    mif.writelines('\n')
    addr = addr + 1

    mif.writelines(str(hex(hex2)).strip().split('0x'))
    mif.writelines('\n')
    addr = addr + 1

for addr in range(addr,4096):
    mif.writelines("00\n")

print('finish')
```

包含了存储器的 testbench 代码如下所示：

```
reg clk,rst_n,cpu_en;
wire done;
wire [15:0] data_addr;
wire [7:0] data_write;
reg [7:0]data_read;
wire [1:0] mem_ctrl;

wire wen,ren;
wire [15:0] addr;

wire cpu_send,cpu_receive;
wire cpu_ready;

reg     ram_send;
reg     ram_receive;

assign addr = data_addr;
assign wen = mem_ctrl[0];
assign ren = mem_ctrl[1];

MIPS CPU(.clk(clk),
         .rst_n(rst_n),
         .cpu_en(cpu_en),
         .data_read(data_read),
         .ram_send(ram_send),
         .ram_receive(ram_receive),
         .cpu_send(cpu_send),
         .cpu_receive(cpu_receive),
         .cpu_ready(cpu_ready),
         .data_addr(data_addr),
         .data_store(data_write),
         .mem_ctrl(mem_ctrl),
         .done(done));
```

```
initial
begin
   clk = 1'b0;
   forever #10 clk = ~clk;
end

initial
begin
   rst_n = 0;
   cpu_en = 0;

   #200
   rst_n = 1;
   cpu_en = 1;
   #20000
   $stop;
end

 reg   [7:0]  bram [4095:0];

 always @(posedge clk or negedge rst_n)
 begin
    if (!rst_n)
      begin
         $readmemh("D:/test.txt",bram);
         data_read <= 8'd0;
      end
    else if (wen) begin
         bram[addr] <= data_write;
    end
    else if (ren) begin
         data_read <= bram[addr];
    end
    else begin
```

```
   else begin
     data_read <= 8'd0;
   end
end


always@(posedge clk or negedge rst_n)// in case of writing
begin
  if(!rst_n)
     ram_receive <= 1'b0;
  else if(wen)
  begin
     if(cpu_send && ram_receive)//priority
        ram_receive <= 1'b0;
     else if((bram[addr] == data_write) && cpu_send)
        ram_receive <= 1'b1;
     else
        ram_receive <= 1'b0;
  end
  else
     ram_receive <= 1'b0;
end

reg avaliable;//prevent re-reading
always@(posedge cpu_ready or negedge cpu_receive)
begin
  if(cpu_ready)
     avaliable <= 1'b1;
  else if(!cpu_receive)
     avaliable <= 1'b0;
  else
     avaliable <= 1'b0;
end
```

```
always@(posedge clk or negedge rst_n)// in case of reading
begin
  if(!rst_n)
     ram_send <= 1'b0;
  else if(ren)
  begin
     if(cpu_receive && ram_send)//priority
        ram_send <= 1'b0;
     else if((data_read == bram[addr]) && (!cpu_receive) && avaliable)
        ram_send <= 1'b1;
     else
        ram_send <= 1'b0;
  end
  else
     ram_send <= 1'b0;
end
```

所有代码已经开源到 github 上，https://github.com/cuixuyang615/16-bits-MIPS-CPU-design

仿真测试结果如下所示：

测试代码 1	测试代码 2	测试代码 3
ADDI R0 R0 25	SUBI R0 R0 25	SUBI R0 R0 25
ADDI R1 R1 25	SUBI R1 R1 25	JR R0 R0 R0/JAL 0
ADD R0 R1 R2	ORI R1 R2 50	
BEQ R1 R2 16	ADDU R0 R0 R1	
ANDI R2 R3 50	SUBU R0 R0 R1	
SLT R1 R3 R4	BNE R1 R2 1	
SW R2 R3 50		
LW R3 R5 50		
SUB R5 R0 R6		
J 0		

其对应的机器指令如下所示：

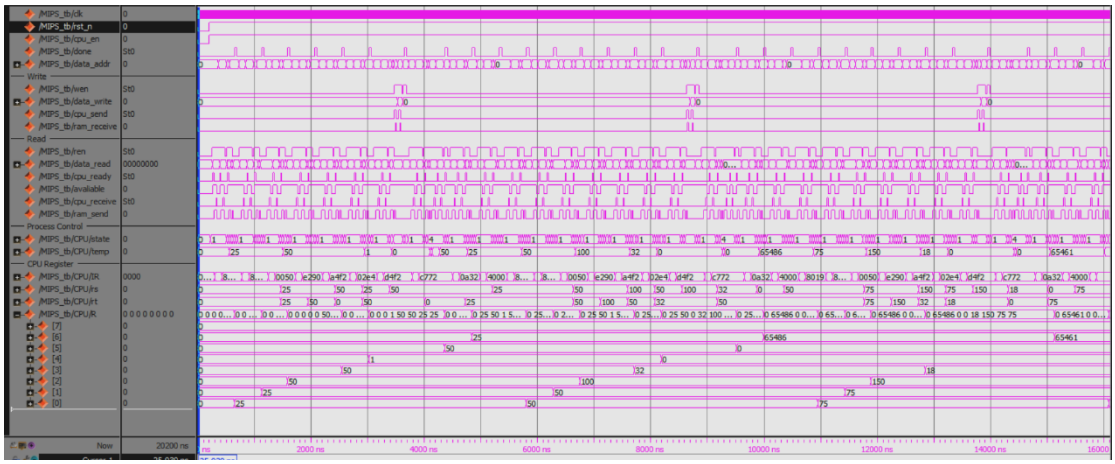
（对应编码）第一部分：
1000000000011001
80,19
1000001001011001
82,59
0000000001010000
0,50
1110001010010000
e2,90
1010010011110010
a4,f2
0000001011100100
2,e4
1101010011110010
d4,f2

1100011101110010
c7,72
0000101000110010
a,32
0100000000000000
40,0
（对应编码）第二部分：
1001000000011001
90,19
1001001001011001
92,59
1011001010110010
b2,b2
0000000000001001
0,9

0000000000001011
0,b
1111001010000001
f2,81

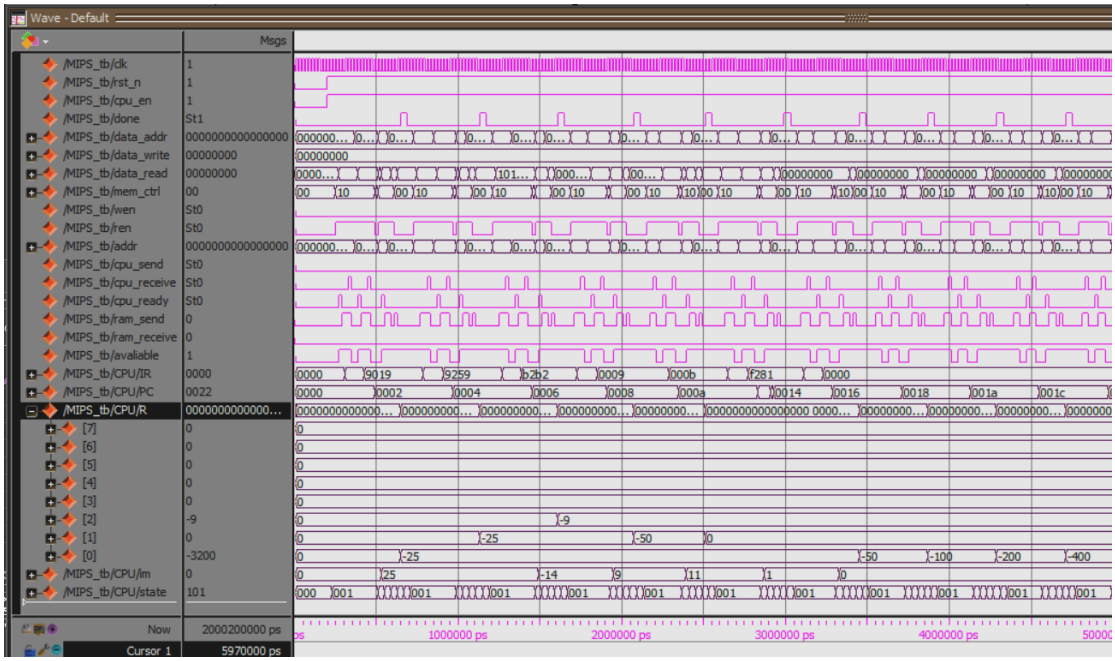
（对应编码）第三部分：
1001000000011001
90,19
0000000000000111（JR）
0,7
1001000000011001
90,19
0101000000000000（JAL 0）
50,0

代码 1 的结果如下所示：



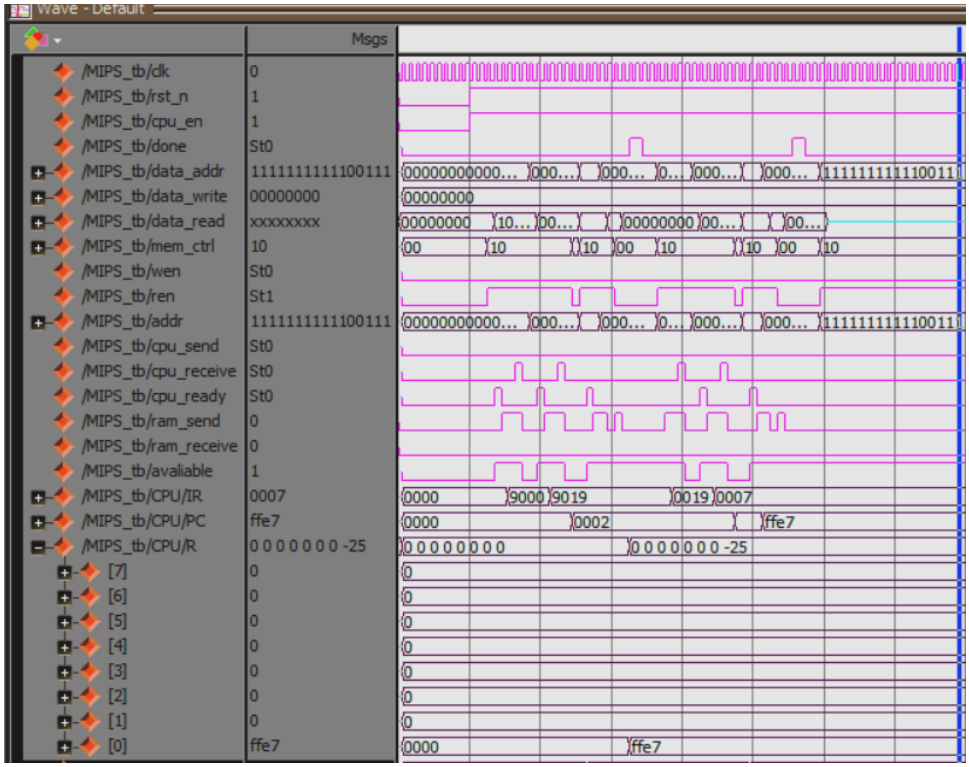
可以看到在当 state 到 1，即 IF 结束时 IR 均能更新到指定的值，观察寄存器的值可以看到在第一次循环中 beq 由于 R1 和 R2 分别为 25 和 50 而不等没有跳转，之后的 LW 也成功读出 SW 存放的值，slt 运算也成功，但第二个循环中 R1 和 R2 的值被更新到相同的 50，触发跳转，跳转到了一个非法（没有初始化的空间中），导致后面均执行的是加 0 指令，使后面循环的 R6 出错

代码 2 的结果如下所示：

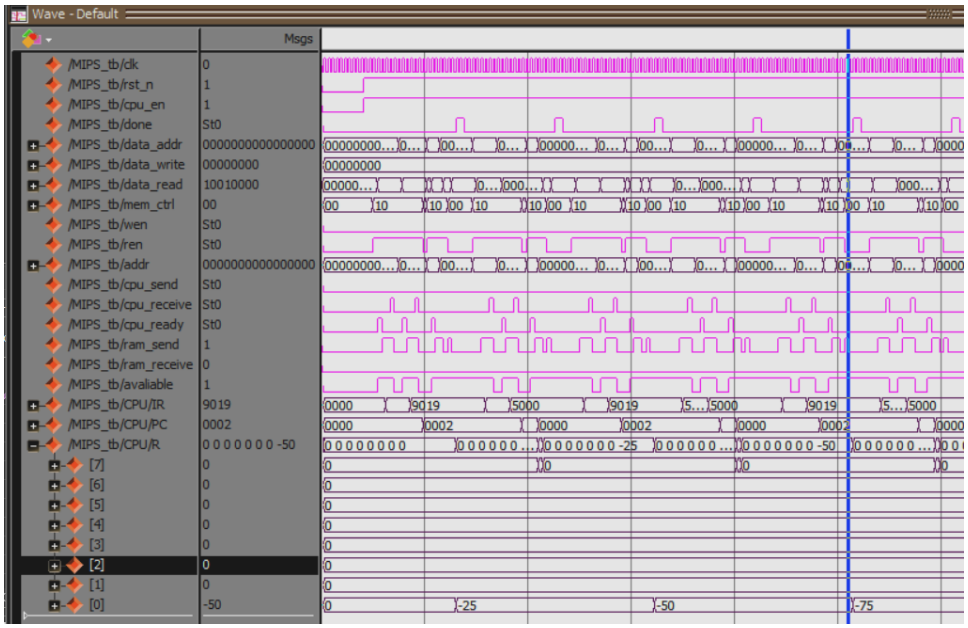


可以看到在当 state 到 1，即 IF 结束时 IR 均能更新到指定的值，SUB 类指令也执行正确，成功把 R0 和 R1 更新到 0-25=-25，后续操作也都正确，且在当指令为 0xf281 即 bne 时正确进行了条件跳转

代码 3 的结果如下所示：



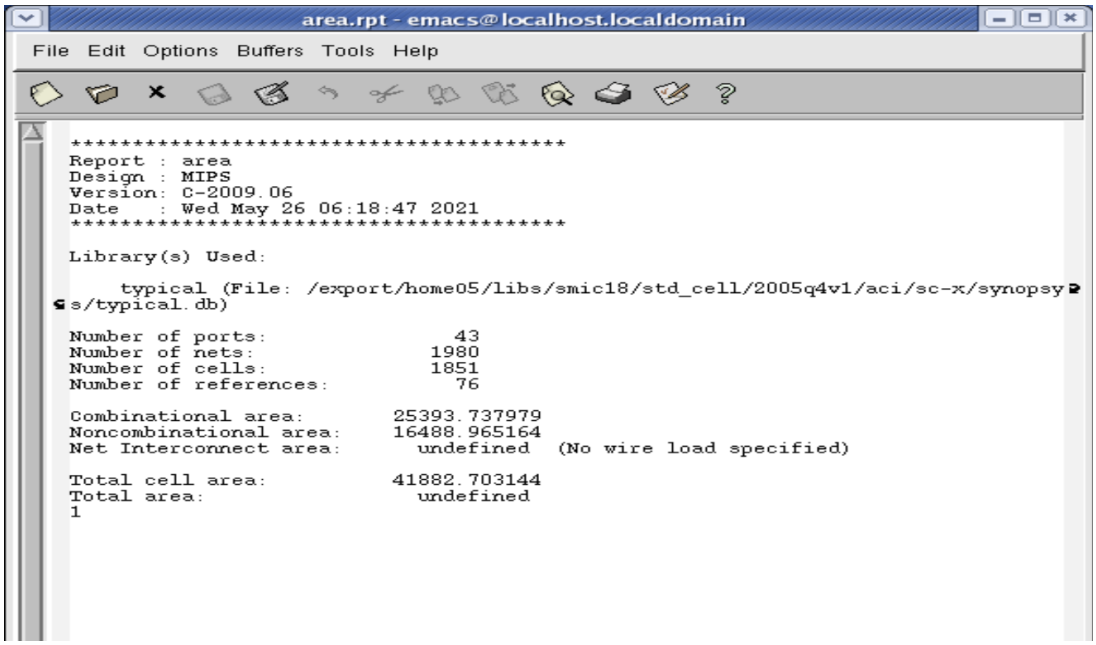
JR 的指令跳转正确，在 R0 由减法更新成-25 后成功使 PC 跳转到-25（只是验证，实际中不允许这么操作跳转到非法地址）



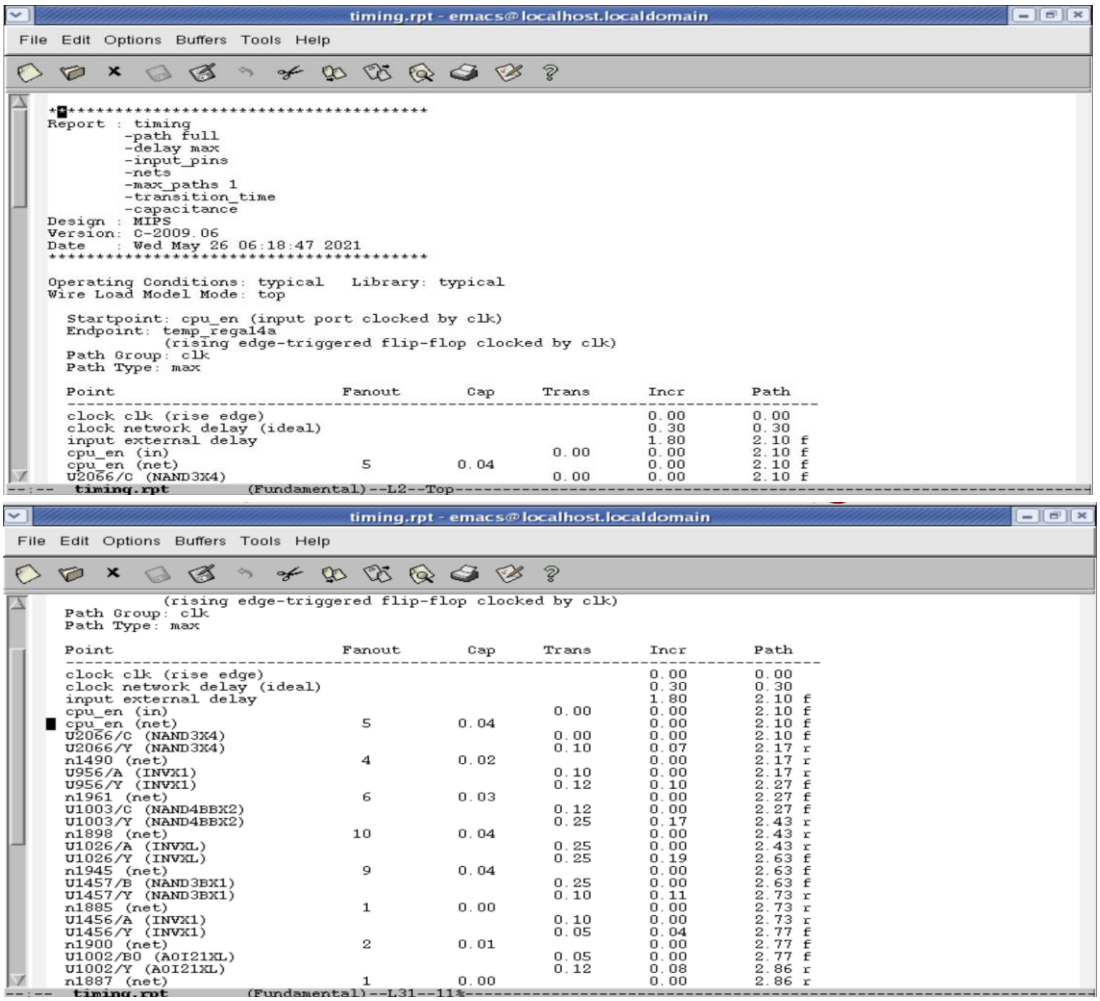
JAL 的指令跳转正确，在 R0 由减法更新成-25 后成功使 PC 跳转到 0，且将 R7 更新成了 2，验证正确。

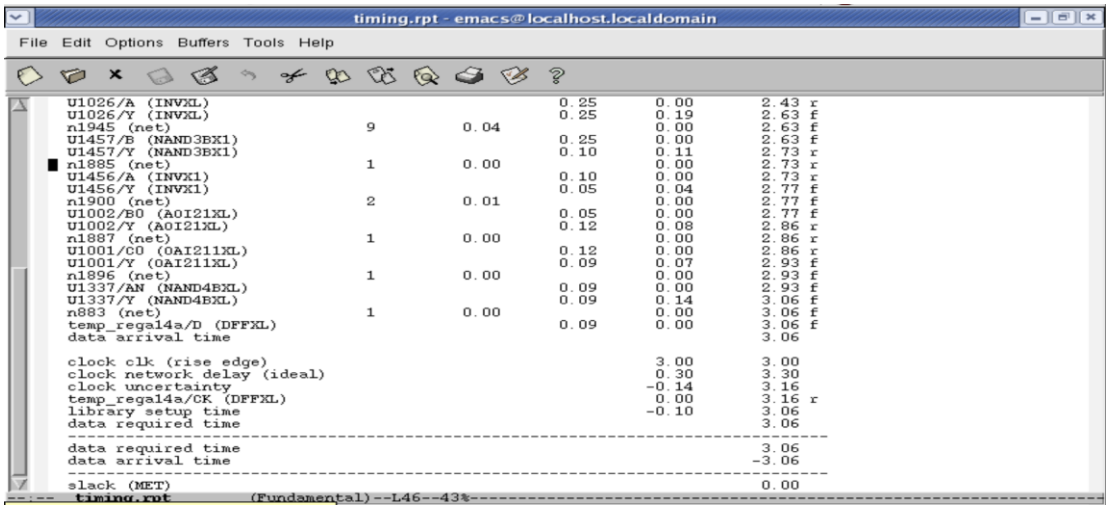
五、性能评估：

使用 DC 分析可以看到综合出的设计总面积为 51882.7 μ m^2，其中组合逻辑的面积为 25393.7 μ m^2，连线和时序逻辑等占据了 16488.9 μ m^2 的面积：



通过调整时序约束的值可以将 slack 调整到 0，以得到最快的时序如下所示：





最终得到的最高时序约束为 3.06ns，即理论上可以加入 326.8MHz 的时钟作为驱动

六、结论

得到了基于 Verilog 的使用 MIPS 指令集的 16 位 CPU 的 RTL 设计，CPU 采用单周期处理方式，后期可以拓展成流水线结构加快连续指令处理速度。CPU 内置 8 个 16 位的通用寄存器，IR 寄存器和 PC 寄存器均为 16 位，通过 rst_n 低电平同步复位，cpu_en 高电平同步使能，可以连续处理指令，其可以通过规定的协议与外部存储器握手通讯，但配套的内存存储必须为 1 地址 1 字节的格式。经验证 CPU 可以正确完成 ADD，ADDU，SUB，SUBU，SLT，AND，OR，JR，ADDI，SUBI，ANDI，ORI，LW，SW，BEQ，BNE，J 和 JAL 共 18 条指令的操作，通过 DC 分析得到其可以被综合成实际的电路，综合出的面积为 51882.7 μm^2，时序约束最小为 3.06ns，理论上最高可以加入 326.8MHz 的时钟作为驱动，使用 Quartus II 工具综合出的 RTL 电路图如下所示：