计算机组成原理实验 实验报告

实验题目：Lab5 流水线CPU

学生姓名：王章瀚

学生学号：PB18111697

完成日期：2020 年 6 月 19 日

计算机实验教学中心制

2019年09月

1 实验题目

Lab5 流水线CPU

2 实验目的

1. 理解流水线CPU的组成结构和工作原理；
2. 掌握数字系统的设计和调试方法；
3. 熟练掌握数据通路和控制器的设计和描述方法。

3 实验平台

Vivado

4 实验过程

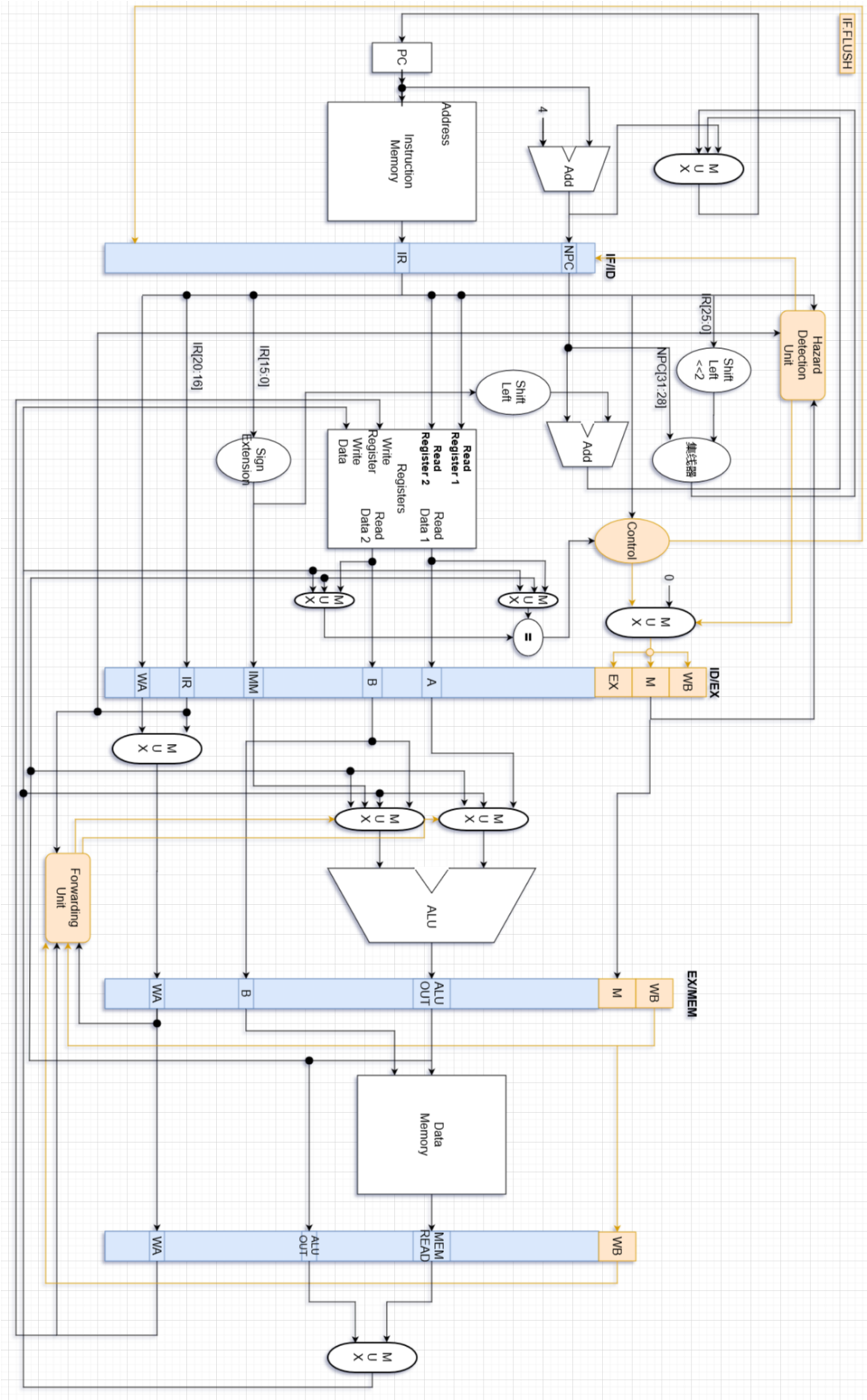
4.1 基本过程

流水线CPU的基本设计其实和单周期CPU比较相似, 所不同的是, 流水线CPU中多了一些模块, 大致可以分为以下几类:

- 段间不再是通过wire直接连接, 而是需要**段间寄存器**
- 相关和冒险时, 需要有**转发单元**
- 如果不能通过转发解决相关, 还需要做**stall**

4.2 数据通路

见下图,


IF-FLUSH
PC
Address
Instruction Memory
4
Add
M U X
IR
NPC
IF/ID
IR[25:0]
Hazard Detection Unit
Shift Left <<2
NPC[31:28]
集线器
IR[20:16]
IR[15:0]
Shift Left
Add
Sign Extension
Read Register 1
Read Register 2
Write Register
Write Data
Registers
Read Data 1
Read Data 2
Control
0
=
WA
IR
IMM
B
A
EX
M
WB
ID/EX
Forwarding Unit
ALU
ALU OUT
EX/MEM
Data Memory
MEM READ

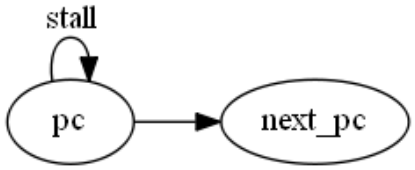
这个数据通路比较复杂, 花了不少时间来画. 相比于老师给出的数据通路, 里面多了

- 一些必要的转发(如给ID段BEQ判等的转发等)
- ID段加入了J指令(老师的图中没有J指令)
- ALU操作数的MUX加入了对IMM的处理
- ...

其余内容大多和老师的相似.

4.3 状态转换

这里的状态转换主要是PC的转换以及分支预测失误时的指令清空. 指令清空比较简单, 只需要把对应段间寄存器置零即可. 而PC的状态转换主要是这样的:



也就是, 当需要流水线停顿的时候, 需要禁止PC的更新(当然, 也要置空ID_EX段, 图中没有呈现)

4.4 代码讲解

首先声明, 由于实验要求中, 并不要求写DBU的代码(不过我写了, 可见附件代码), 因此报告中就不加以描述了.

流水线CPU比单周期的多周期的CPU要复杂得多, 它的代码模块也比较多, 大体可以分为以下这几类来讲解:

- 数据通路的连接
- 控制单元的设计
- 转发单元(Forwarding Unit)
- 冒险探测器
- ALU单元*
- 寄存器文件*

其中, 标*的模块已经在单周期和多周期甚至更早的实验中就已经讲解过多次, 这里再赘述就浪费篇幅了, 因此略去. 主要讲解的代码部分为数据通路, 控制单元, 转发单元, 冒险探测器.

4.4.1 数据通路代码

限于篇幅, 就不将所有代码放出来(若有需要, 可查看附件中的代码), 只放一些关键代码. 而关键代码中, 大部分都是按照前述数据通路图(见4.2), 这一部分只是简单地用wire变量进行连线, 比较简单. 主要提一下的就是各种数据线的src(source)的相关控制:

1. pc_src(pc soource)的数据通路
   会看到,这里对于PC的source有比较多的选择, 这是因为需要考虑分支预测的成功与失败, 各类PC来源等等. 因此看起来比较复杂.

```
register_syn #(.N(32))
    rIF_ID_NPC(.clk(clk),
               .rst(rst || (flush && ~stall)),
               .we(~stall),
               .wd(PC+4),
               .d(IF_ID_NPC));

//// 跳转指令等的PC赋值
always @(posedge clk, posedge rst) begin
    if(rst) begin
        PC <= 32'h0000_0000;
    end
    else begin
        if(pc_we == 1'b1) begin
            case(pc_src)
                3'b000: PC <= PC + 4;
                3'b001: PC <= IF_ID_NPC + (ID_instr_imm << 2);
                3'b010: PC <= {IF_ID_NPC[31: 28], ID_instr_25_0_sll_2};
                3'b011: PC <= PC;
                3'b100: PC <= PC + 4 + (im_instr_imm << 2);
                3'b101: PC <= IF_ID_NPC;
                default: PC <= PC;
            endcase
        end
        else PC <= PC;
    end
end

```

2. alu_src和write_date_src的数据通路
   这部分和转发单元一起发挥作用, 保证ALU和Data Memory的输入正确.

```
always @(*) begin
    case(alu_a_src)
        2'b00: alu_a = ID_EX_A;
        2'b10: alu_a = WB_wb_data;
        2'b11: alu_a = EX_MEM_Y;
        default: alu_a = ID_EX_A;
    endcase
    case(alu_b_src)
        2'b00: alu_b = ID_EX_B;
        2'b01: alu_b = ID_EX_IMM;
        2'b10: alu_b = WB_wb_data;
        2'b11: alu_b = EX_MEM_Y;
        default: alu_b = ID_EX_B;
    endcase
    case(write_data_src)
```

```
        2'b00: write_data = ID_EX_B;
        2'b10: write_data = WB_wb_data;
        2'b11: write_data = EX_MEM_Y;
        default: write_data = ID_EX_B;
    endcase
end

```

4.4.2 控制单元代码

这里就不展示模块头等了, 直接上重要部分代码:

1. pc_src的处理 因为设计了思考题中的分支预测, 所以这里的pc_src还比较复杂, 需要考虑的变量包括但不限于

   - 当前是否是分支指令, 如果是, 是beq还是jump
   - 分支预测的结果是跳转还是不跳转
   - 现在看来结果应该是跳转还是不跳转
   - ...

   代码如下,

```
always @(*) begin
    if(opcode == BEQ_op) begin
        if(equal) begin
            if(had_branched) begin
                flush = 1'b0;
                if(shall_branch) begin
                    pc_src = 3'b100;
                end
                else begin
                    pc_src = 3'b000;
                end
            end
            else begin
                pc_src = 3'b001;
                flush = 1'b1;
            end
        end
        else begin
            if(had_branched) begin
                flush = 1'b1;
                pc_src = 3'b101;
            end
            else begin
                flush = 1'b0;
                if(shall_branch) begin
                    pc_src = 3'b100;
                end
                else begin
                    pc_src = 3'b000;
                end
            end
        end
    end
```

```
    else if(shall_branch) begin
        pc_src = 3'b100;
        flush = 1'b0;
    end
    else if(opcode == J_op) begin
        pc_src = 3'b010;
        flush = 1'b1;
    end
    else begin
        pc_src = 3'b000;
        flush = 1'b0;
    end
end

```

2. 各种控制信号

   这个倒是没什么新奇的, 基本和单周期的一样. 代码如下

```
always @(*) begin
    {reg_dst, alu_op, alu_src, mem_read, mem_write, reg_write, mem_to_reg} = 11'h000;
    case(opcode)
        ADD_op: begin
            reg_dst = 1'b1;
            alu_op = 2'b10; // 按 funct
            reg_write = 1'b1;
        end
        ADDI_op: begin
            alu_op = 2'b00; // 做加法
            alu_src = 1'b1; // 选立即数
            reg_write = 1'b1;
        end
        LW_op: begin
            alu_op = 2'b00; // 做加法
            alu_src = 1'b1; // 选立即数
            mem_read = 1'b1;
            reg_write = 1'b1;
            mem_to_reg = 1'b1;
        end
        SW_op: begin
            alu_op = 2'b00; // 做加法
            alu_src = 1'b1; // 选立即数
            mem_write = 1'b1;
        end
        BEQ_op: begin
            alu_src = 1'b1;
        end
        J_op: begin
            alu_src = 1'b1;
        end
        default: {reg_dst, alu_op, alu_src, mem_read, mem_write, reg_write, mem_to_reg} = 11'
    h000;
    endcase
end

```

4.4.3 转发单元代码

转发单元主要是alu_src的转发, equal_src的转发, 和write_data_src的转发.

1. alu_src的转发

   这里alu_a和alu_b基本一样, 限于篇幅, 展示alu_b的(因为它还多了个imm的选择)

```
// alu_b_src
if(alu_src == 1'b1) begin
    alu_b_src = 2'b01;
end
else if(EX_MEM_WA == ID_EX_rt && EX_MEM_reg_write == 1'b1 && EX_MEM_WA != 5'b00000) begin
    alu_b_src = 2'b11;
end
else if(MEM_WB_WA == ID_EX_rt && MEM_WB_reg_write == 1'b1 && MEM_WB_WA != 5'b00000) begin
    alu_b_src = 2'b10;
end
else begin
    alu_b_src = 2'b00;
end

```

2. equal_src的转发

   来源有寄存器堆, alu_y, memory读出结果, EX.MEM段的Y. 代码如下:

```
// equal_a_src
if(ID_EX_WA == IF_ID_IR[25:21] && ID_EX_reg_write == 1'b1 && ID_EX_WA != 5'b00000) begin
    equal_a_src = 2'b01;
end
else if(EX_MEM_WA == IF_ID_IR[25:21] && EX_MEM_reg_write == 1'b1 && EX_MEM_WA != 5'b00000)
    begin
    if(EX_MEM_mem_read) begin
        equal_a_src = 2'b10;
    end
    else begin
        equal_a_src = 2'b11;
    end
end
else begin
    equal_a_src = 2'b00;
end

```

3. write_data_src的转发

   来源有ID.EX的B, WB段的写回数据, EX.MEM段的Y. 代码如下:

```
// write_data_src
if(EX_MEM_WA == ID_EX_rt && EX_MEM_reg_write == 1'b1 && EX_MEM_WA != 5'b00000) begin
    write_data_src = 2'b11;
end
else if(MEM_WB_WA == ID_EX_rt && MEM_WB_reg_write == 1'b1 && MEM_WB_WA != 5'b00000) begin
    write_data_src = 2'b10;
end
else begin
    write_data_src = 2'b00;
```

```
end

```

4.4.4 冒险探测器代码

这一部分主要是为了lw指令(就实验要求的6条指令而言)准备的, 因为它在MEM段才出数据. 代码如下, 在判断到冒险的时候就置stall为1'b1, pc_we为1'b0即可.

```
module hazard_detection_unit
(
input mem_read,
input [31:0] IF_ID_IR,
input [31:0] ID_EX_IR,
output reg pc_we,
output reg stall
);

always @(*) begin
    if(mem_read == 1'b1
        && (ID_EX_IR[20:16] == IF_ID_IR[25:21]
            || ID_EX_IR[20:16] == IF_ID_IR[20:16])) begin
        pc_we = 1'b0;
        stall = 1'b1;
    end
    else begin
        pc_we = 1'b1;
        stall = 1'b0;
    end
end

endmodule
```

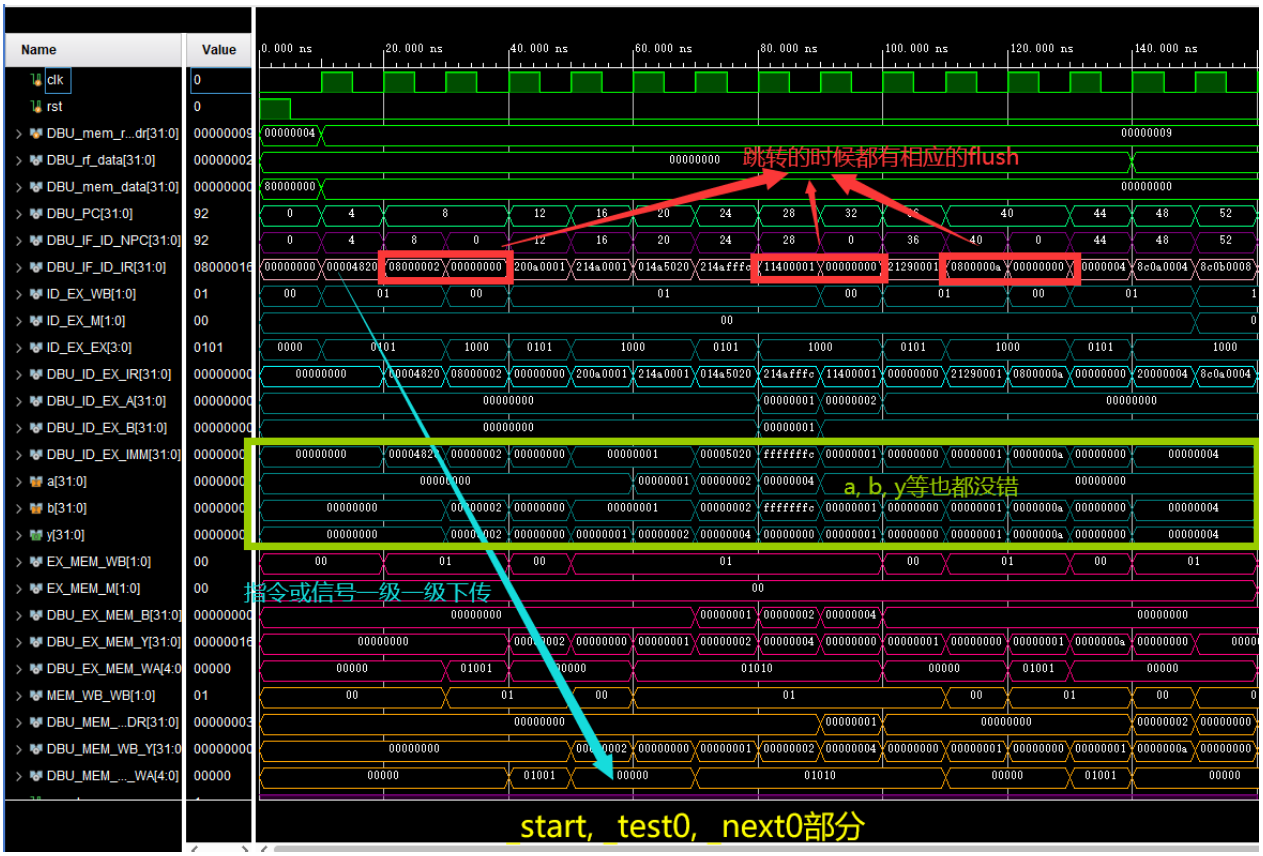
5 实验结果

再次声明, 由于实验不要求实现DBU, 实验结果就不展示DBU模块了.
这次使用的测试代码是助教给出的第二个测试代码(见附录9.1或提交的代码附件)
下面分部分展示仿真结果(花花绿绿的不是出错, 只是这样方便观察流水线各段):

5.1 start, test0, next0三个部分

见下图, 可以看到

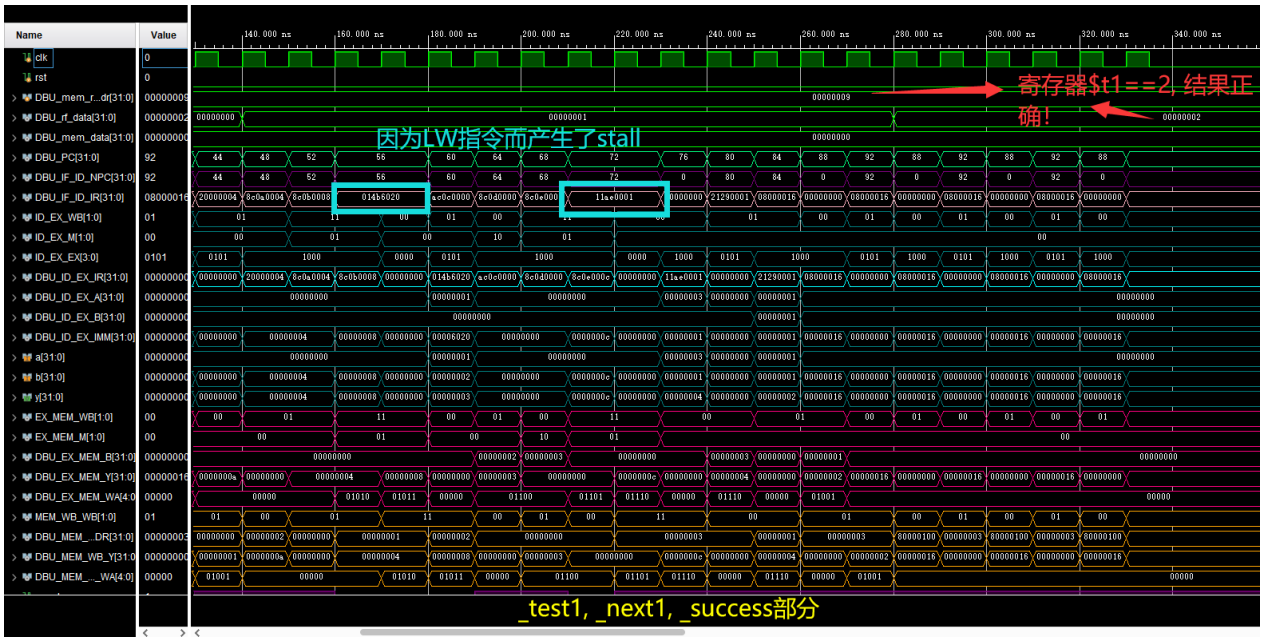
1. 红色框和箭头标识的部分是跳转指令, 都有相应的flush操作
2. 绿色大框标识了输入给ALU的a,b, 以及ALU的输出y. 都是正确的. 这还说明了转发机制是足够完整的.
3. 浅蓝色箭头标志了信号或指令的流水线一级一级往下传.



5.2 test1, next1, success三个部分

见下图, 可以看到

1. 红色箭头标识了最终结果的正确性, 也就是$t1=2
2. 浅蓝色箭头标志了由于LW指令而产生了相关, 因此需要stall, 停顿流水线



6 思考题

这次思考题要求实现分支预测器.

我实现的分支预测器是一个简单的二位动态分支预测.

事实上, 前面的讲的代码就是使用了分支预测器的, 只不过默认不跳转, 所以表现出来和没有分支预测器没有任何区别.

下面用一个能表现出区别的示例代码(见附录9.2[1])来展现.

可以看到图中原来一个循环中IF.ID_NPC是要经历16,20,24,28,0的循环, 过了几次后, 变成16,20,24,0了.

反过来也是可以的, 但是限于篇幅就不展示了, 助教如果感兴趣可以自己运行一下我的代码(附件里都有)

7 心得体会

本次流水线设计是目前最难的实验了. 除了前面提到的这种, 在ID段做BEQ的方法, 我还尝试了在MEM段做BEQ的方法, 在IF段做Jump的方法等. 均有实现. 但综合考虑, 似乎在ID段做BEQ能权衡好关键路径的长度等问题.

此外, 还巩固了分支预测的知识, 并且实现了它.

[1] 特别感谢黄致远同学提供测试代码

8 意见建议

流水线的数据通路示例比较不完整了, 需要自己补全很多东西. 可以稍微完整一点.

9 附录

9.1 附录1

```
# Test cases for MIPS 5-Stage pipeline

.data
    .word 0,1,2,3,0x80000000,0x80000100,0x100,5,0

_start:
    add $t1, $0, $0     # $t1 = 0                                                   00004820
    j _test0            #                                                         08000002

_test0:
    addi $t2, $0, 1      # $t2 = 1                                                  200a0001
    addi $t2, $t2, 1      # $t2 = 2                                                 214a0001
    add $t2, $t2, $t2      # $t2 = 4                                                014a5020
    addi $t2, $t2, -4     # $t2 = 0                                                 214afffc
    beq $t2, $0, _next0   # if $t2 == $0: $t1++, go next testcase, else: go fail   11400001
    j _fail                 #                                                       08000015
_next0:
    addi $t1, $t1, 1      # $t1++                                                   21290001
    j _test1                #                                                     0800000a

_test1:
    addi $0, $0, 4      # $0 += 4                                                   20000004
    lw $t2, 4($0)       # $t2 = MEM[1]                                              8c0a0004
    lw $t3, 8($0)       # $t3 = MEM[2]                                              8c0b0008
    add $t4, $t2, $t3   #                                                         014b6020
    sw $t4, 0($0)       # MEM[0] = $t4                                              ac0c0000
    lw $t5, 0($0)       # $t5 = MEM[0]                                              8c0d0000
    lw $t6, 12($0)      # $t6 = MEM[3]                                              8c0e000c
    beq $t5, $t6, _next1    #                                                     11ae0001
    j _fail                 #                                                     08000015

_next1:
    addi $t1, $t1, 1        #                                                     21290001
    j _success              #                                                     08000016

_fail:
    j _fail                 #                                                     08000015

_success:
    j _success   # if success: $t1 == 2                                           08000016
```

9.2 附录2

```
1   #calculate 4*5
2   #save to &=$s1
3   _start:
4           addi $t0,$0,4           # t0=4   0                  20080004
5           addi $t1,$0,5           # t1=5  4                   20090005
6   _loop2:
7           add $t2,$t0,$0          # mov $t0 to $t2    8       01005020
8   _loop1:
9           addi $t2,$t2,-1         # t2 = t2 - 1    12         214affff
10          addi $s0,$s0,1          # s0 = s0 + 1   16          22100001
11          beq $t2,$zero,_loop1out #       20                 11400001
12          j _loop1                #       24                 08000003
13  _loop1out:
14          addi $t1,$t1,-1         # t1 = t1 - 1   28          2129ffff
15          beq  $t1,$zero,_finish  #       32                 11200001
16
17          j _loop2                #       36                 08000002
18  _finish:
19          sw   $s0,8($0)          # sw to 0x8     40          ac0a0008
20          j   _finish             #       44                 0800000a
21
22
```