流水线 CPU 处理器设计

无 03　王与进　2020010708

目录

1 Introduction

本流水线 CPU 的总体结构如下：

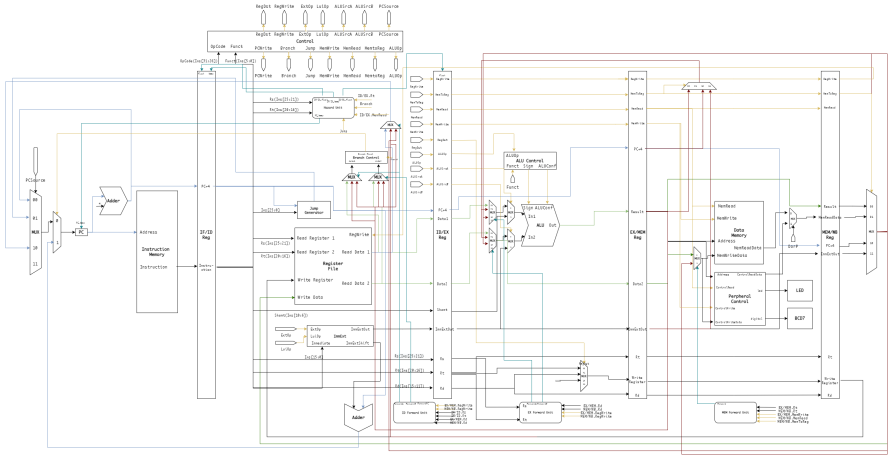

图 1: 流水线总设计图

本 CPU 一共含有 5 个阶段：取指（IF）、译码（ID）、执行（EX）、访存（MEM）、写回（WB）。

本 CPU 支持的指令如下：

R 型 sll, srl, sra, jr, jalr, add, addu, sub, subu, and, or, xor, nor, slt, sltu

I 型 bltz, bgez, beq, bne, blez, bgtz, addi, addiu, slti, sltiu, andi, ori, xori, lui, lw, sw

J 型 j, jal

除了核心的 CPU 模块之外，本项目还有以下几种外设：LED、7 位数码管和系统计时器。

值得一提的是，本项目中的 InstructionMemory 和 DataMemory 均是由 vivado 中的 IP Core 生成的 ROM 生成的，其内存可由.coe 文件初始化。

本 CPU 支持若干转发和冒险处理，详见第二部分。

2 Method

2.1 流水线分段介绍——IF

2.1.1 基本数据通路

在 IF 阶段中，我们在时钟的上升沿时刻从 PC 寄存器中取出 PC，并更新 PC。PC 的来源值可能为：PC+4(默认情况)、jump_address(由立即数生成的跳转地址)、jump_reg_address(由寄存器中的值生成的跳转地址)、beq_address（判断指令生成的跳转地址）。新生成的 PC 地址会被输入 Instruction Memory，输出对应指令。

PC+4 的值以及取出的指令 Instruction 会被写入 IF/ID 阶段之间的寄存器。

```
1    assign w_pc = (pc_source_ID == 2'b01)?jump_address:
2                  (pc_source_ID == 2'b10)?jump_reg_address:
3                  pc_IF;
4
```

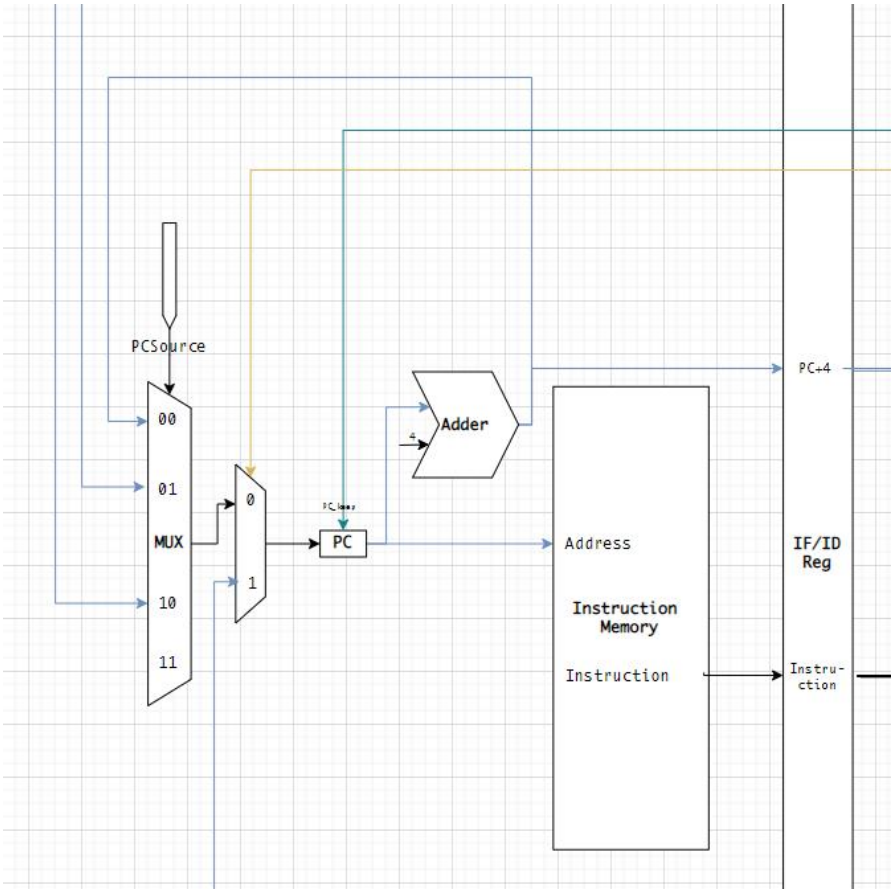

图 2: IF 阶段设计

```
5    assign i_pc = (branch_final == 0)?w_pc:branch_address;
```

Listing 1: PC 更新逻辑

2.1.2 转发/冒险处理

此外，PC 的写入还受到冒险单元控制的 PCkeep 信号控制。这一信号的作用机制将在 ID 阶段详细介绍。

2.2 流水线分段介绍——ID

2.2.1 基本数据通路

在 ID 阶段中，我们首先从 IF/ID 寄存器中取出暂存的指令，取出其 opcode、funct、rs、rd、rt、shamt、immediate、address 的值。

利用 opcode 和 funct 的值，可以在 Control 模块中生成对应的控制信号供后续使用。

Immediate Extension 模块接受 immediate 值，并受 LuiOp 和 ExtOp 控制，生成的数字分为 4 种：无扩展、符号扩展、符号扩展并左移 2 位、左移 16 位。

Register File 模块在读出寄存器中的值只需要两个 read register 的值，而无需时钟上升沿；但在写入寄存器时，需要时钟上升沿以及来自 WB 阶段的 RegWrite 信号为高才能写入。值得注意的是，为了保证寄存器先写后读，我们需要加入以下处理逻辑——如果写入

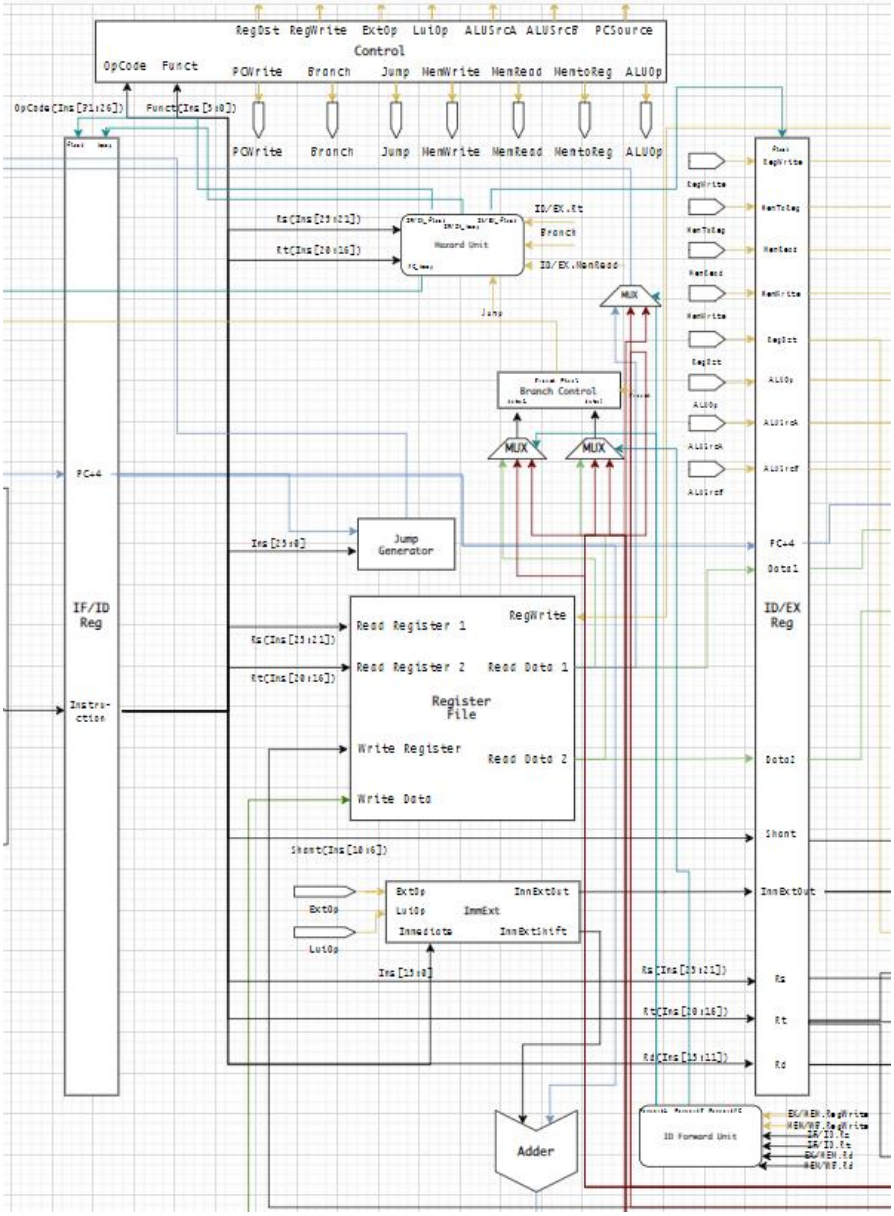

图 3: ID 阶段设计

的寄存器与读入的寄存器相同，我们就让读出的数据直接等于写入的数据，而不是从寄存器堆中取用。

```
assign o_read_data1 = (i_read_register1 == 5'b00000)? 32'h00000000:
    (i_write_register == i_read_register1)?i_write_data:
    RF_data[i_read_register1];
assign o_read_data2 = (i_read_register2 == 5'b00000)? 32'h00000000:
    (i_write_register == i_read_register2)?i_write_data:
    RF_data[i_read_register2];
```

Listing 2: ID Forward Unit

分支指令和跳转指令的指令跳转均在 ID 阶段进行。

分支指令 分支指令中两个判断数来源于寄存器读取或后续阶段转发。是否跳转取决于两个数之间的大小关系和 branch 信号，由 Branch Control 模块控制。

```
assign compare_data_1 = (forward_A_ID == 2'b01)?alu_out_MEM:
(forward_A_ID == 2'b10)?write_register_data_WB:
read_data_1_ID;

```

Listing 3: 判断数的来源

值得注意的是，由于此处的分支指令有所扩充，所以 branch 信号本身被扩充到了 3 位，以适应不同的比较需求。

```
wire [1:0] w_relation;
assign w_relation = (i_data1 < i_data2)?2'b00:
(i_data1 > i_data2)?2'b10:
2'b01;

// 00:< 01: = 10:>
always @(*) begin
    case(i_branch)
        3'b000:begin // no branch
            o_branch <= 0;
        end

        3'b001:begin // beq
            case(w_relation)
            2'b01:begin
                o_branch <= 1;
            end
            default:begin
                o_branch <= 0;
            end
            endcase
        end

        3'b010:begin // bne
            case(w_relation)
            2'b01:begin
                o_branch <= 0;
            end
            default:begin
                o_branch <= 1;
            end
            endcase
        end

        3'b011:begin // blez
            if (i_data1 < 0 || i_data1 == 0)begin
                o_branch <= 1;
            end
            else begin
                o_branch <= 0;
            end
        end

        3'b100:begin // bgtz
            if (i_data1>0)begin
                o_branch <= 1;
            end
            else begin
                o_branch <= 0;
            end
```

```
            end

            3'b101:begin // bltz
                if (i_data1<0)begin
                    o_branch <= 1;
                end
                else begin
                    o_branch <= 0;
                end
            end

            default:begin
                o_branch <= 0;
            end

        endcase
    end

```

Listing 4: Branch Control

跳转指令 跳转指令的目标地址有两种：立即数或寄存器。源自寄存器的跳转指令可能需要转发。

```
        assign jump_reg_address = (forward_A_ID == 2'b01)?alu_out_MEM:
                                  (forward_A_ID == 2'b10)?write_register_data_WB:
                                  read_data_1_ID;

```

Listing 5: 源自寄存器的跳转指令

被写入 ID/EX 阶段之间的寄存器的值如图3所示。

2.2.2 转发/冒险处理

转发模块为 ID Forward Unit，其转发逻辑如下：

```
assign o_forward_A = (i_EX_MEM_reg_write && (i_write_register_MEM == i_IF_ID_Rs))?2'b01:
                     (i_MEM_WB_reg_write && (i_write_register_WB == i_IF_ID_Rs))?2'b10:
                     2'b00;

assign o_forward_B = (i_EX_MEM_reg_write && (i_write_register_MEM == i_IF_ID_Rt))?2'b01:
                     (i_MEM_WB_reg_write && (i_write_register_WB == i_IF_ID_Rt))?2'b10:
                     2'b00;
```

Listing 6: ID Forward Unit

这一转发模块用于将 ALU 计算得到的值或从 Data Memory 中取出的值转发给 ID 阶段供分支判断或寄存器值跳转使用。

冒险处理模块为 Hazard Unit，其处理逻辑如下：

```
    // deal with 5 kinds of hazard
    // load use hazard: keep IF_ID_Register, keep PC, flush ID_EX_Register
    // beq hazard: flush IF_ID_Register, flush ID_EX_Register
    // jump hazard: flush IF_ID_Register
    // R/load-jr/jalr hazard: keep IF_ID_Register, keep PC, flush ID_EX_Register
    // R/load-beq hazard: keep IF_ID_Register, keep PC, flush ID_EX_Register
    wire pc_keep1,pc_keep2,pc_keep3,pc_keep4;

```

```
    // for load use hazard & R/load-jr/jalr
    assign pc_keep1    = ((i_ID_EX_mem_to_reg == 2'b01 || i_ID_EX_mem_to_reg == 2'b11) && ((i_ID_EX_Rt == i_IF_ID_Rs) || (i_ID_EX_Rt == i_IF_ID_Rt)));
    assign pc_keep2    = (i_ID_EX_mem_to_reg == 2'b10 && (i_IF_ID_Rs == 5'b11111 || i_IF_ID_Rt == 5'b11111));
    assign pc_keep3    = ((i_jump == 2'b10 || i_branch) && (i_ID_EX_reg_write && ((i_write_register_EX == i_IF_ID_Rs) || (i_write_register_EX == i_IF_ID_Rt))))?1:0;
    assign pc_keep4    = ((i_jump == 2'b10 || i_branch) && (i_EX_MEM_mem_read && ((i_write_register_MEM == i_IF_ID_Rs) || (i_write_register_MEM == i_IF_ID_Rt))))?1:0;
    assign o_pc_keep   = reset?0:(pc_keep1 || pc_keep2 || pc_keep3 || pc_keep4);
    assign o_IF_ID_keep = o_pc_keep;

    // for beq hazard
    assign o_ID_EX_flush = reset?0:
                           (i_branch_final || o_pc_keep);

    // for jump hazard
    assign o_IF_ID_flush = reset?0:
                           (i_branch_final || i_jump);
endmodule
```

Listing 7: hazard unit

由代码也可以看出，该模块一共处理 5 种类型的冒险：

load use hazard 当前一条指令为 lw，后一条指令为 R 型或计算型 I 型，且 lw 写入的寄存器与后一条指令使用的寄存器相同时，需要保持 PC 和 IF/ID 寄存器不变，并将 ID/EX 寄存器清零。

beq hazard 当分支指令发生跳转时，此时已经进行到 IF 阶段的下一条指令将会被作废，将 IF/ID 和 ID/EX 寄存器清零。

jump hazard 当跳转指令发生跳转时，此时已经进行到 IF 阶段的下一条指令将会被作废，将 IF/ID 寄存器清零。

R/load-jr/jalr hazard 当 jr/jalr 的上一条指令为 R 型/计算型 I 型，或上上一条指令为 lw，且 R 型/计算型 I 型/lw 的目标寄存器与 jr/jalr 用到的寄存器相同时，需要保持 PC 和 IF/ID 寄存器不变，并将 ID/EX 寄存器清零。

R/load-beq hazard 当分支指令的上一条指令为 R 型/计算型 I 型，或上上一条指令为 lw，且 R 型/计算型 I 型/lw 的目标寄存器与分支指令用到的寄存器相同时，需要保持 PC 和 IF/ID 寄存器不变，并将 ID/EX 寄存器清零。

2.3 流水线分段介绍——EX

2.3.1 基本数据通路

在 EX 阶段，ALU Control 会根据 ALUOp 和 Funct 生成 ALUConf 信号，用于 ALU 的运算。

同时，此处会根据 RegDst 选择写入的目标寄存器（rt，rd 或 ra），在 WB 阶段发挥作用。

```
assign write_register_EX = (reg_dst_EX == 2'b00)?rt_EX:
    (reg_dst_EX == 2'b01)?rd_EX:
    (reg_dst_EX == 2'b10)?5'b11111:
    rt_EX;
```

Listing 8: 目标寄存器的选择

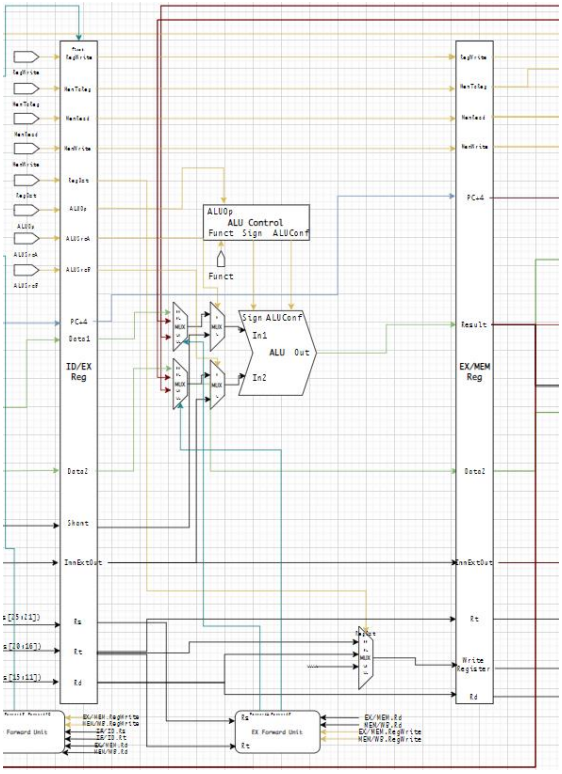

图 4: EX 阶段设计

2.3.2 转发/冒险处理

转发模块为 EX Forward Unit，其转发逻辑如下：

```
    wire w_cond1;
    assign w_cond1 = (i_EX_MEM_reg_write && i_write_register_MEM);

    wire w_cond2;
    assign w_cond2 = (i_MEM_WB_reg_write && i_write_register_WB);

    // from EX_MEM Register: 01
    assign w_forward_A = (w_cond1 && (i_write_register_MEM == i_ID_EX_Rs))?2'b01:2'b00;
    assign w_forward_B = (w_cond1 && (i_write_register_MEM == i_ID_EX_Rt))?2'b01:2'b00;

    // from MEM_WB Register: 10
    assign o_forward_A = (w_cond2 && (i_write_register_WB == i_ID_EX_Rs)&&!w_forward_A)?2'
    b10:w_forward_A;
    assign o_forward_B = (w_cond2 && (i_write_register_WB == i_ID_EX_Rt)&&!w_forward_B)?2'
    b10:w_forward_B;
```

Listing 9: EX Forward Unit

此转发模块用于将 ALU 产生的计算结果或 Data Memory 中载入的数据转发给 ALU 用于计算，这适用于 read after write hazard 和 load use hazard。

2.4 流水线分段介绍——MEM

2.4.1 基本数据通路

在 MEM 阶段，我们需要使用上一步 ALU 计算得到的地址，从 Data Memory 中读出数据（MemRead 为高电平）或写入数据（MemWrite 为高电平）。其中，Data Memory 的字节地址为 0x00000000 0x000007FF。

与此同时，CPU 对于外设的控制也会在这一部分进行，详情见“外设介绍”一节。

此阶段在进行转发时，会根据 MemtoReg 的值转发不同的值（ALU 计算结果、立即数扩展值、PC+4）。

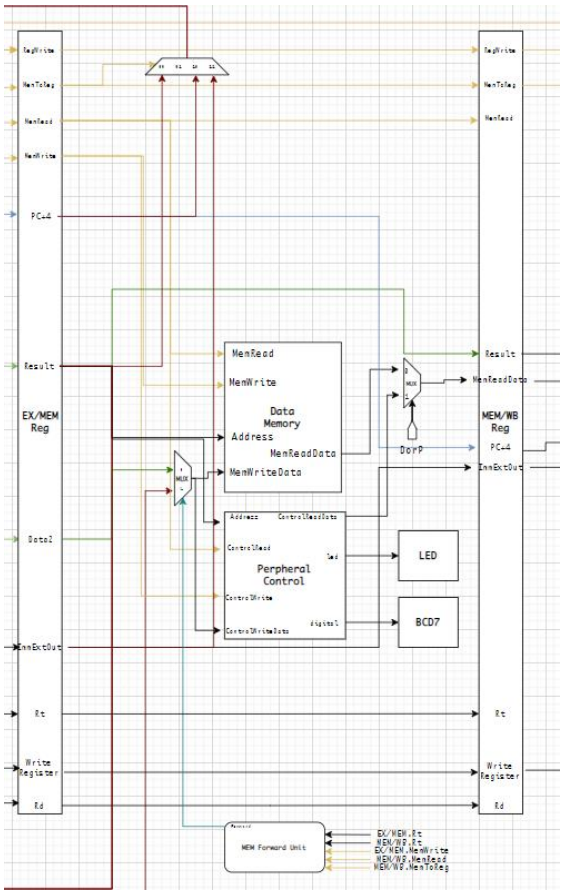

图 5: MEM 阶段设计

2.4.2 转发/冒险处理

转发模块为 MEM Forward Unit，其转发逻辑如下：

```
assign o_forward = (i_EX_MEM_Rt && i_EX_MEM_mem_write && (i_MEM_WB_mem_to_reg == 2'b01
|| i_MEM_WB_mem_to_reg == 2'b11) && i_EX_MEM_Rt == i_MEM_WB_Rt)?1:(i_EX_MEM_Rt == 5'
b11111 && i_EX_MEM_mem_write && i_MEM_WB_mem_to_reg == 2'b10)?1:0;
```

Listing 10: MEM Forward Unit

该模块处理以下冒险：

lw-sw hazard 当前一条指令为 lw，当前指令为 sw，且 lw 写入的寄存器和 sw 的数据来源寄存器相同时，需要将 Data Memory 读出的数据转发给 MemWriteData。

lui-sw hazard 当前一条指令为 lui，当前指令为 sw，且 lw 写入的寄存器和 sw 的数据来源寄存器相同时，需要将 Immediate Extension 生成的数据转发给 MemWriteData。

jal-sw hazard 当前一条指令为 jal，当前指令为 sw，且 sw 的数据来源寄存器为 ra 时，需要将 PC+4 转发给 MemWriteData。

2.5 流水线分段介绍——WB

2.5.1 基本数据通路

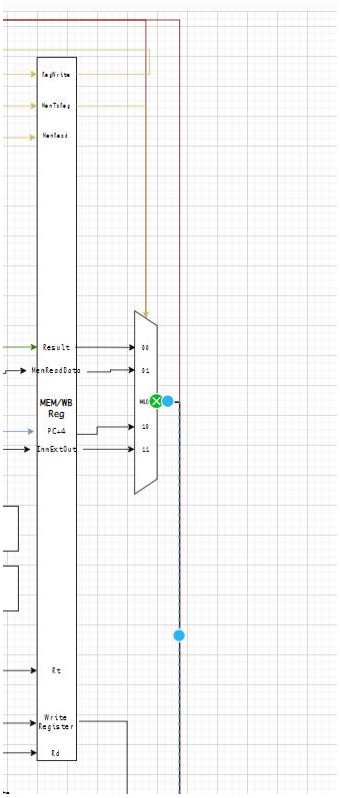

图 6: WB 阶段设计

本阶段会根据 MemtoReg 的值，选择 ALU 计算结果、Data Memory 读出的结果、立即数扩展结果或是 PC+4 的值，将其写回至 Register File。

```
assign write_register_data_WB = (mem_to_reg_WB == 2'b00)?alu_out_WB:
(mem_to_reg_WB == 2'b01)?mem_read_data_WB:
(mem_to_reg_WB == 2'b10)?pc_WB:
(mem_to_reg_WB == 2'b11)?imm_ext_out_WB:
alu_out_WB;
```

Listing 11: 写回数据的选择

2.6 外设介绍

本项目共有 3 个外设：LED、7 位数码管、系统时钟计数器。其中，访问 LED 的地址为 0x4000000C，访问七段数码管的地址为 0x40000010，访问系统时钟计数器的地址为 0x40000014。

对外设的控制在流水线的 MEM 阶段进行。CPU 会根据地址的大小生成 DorP 信号，从而选择从 Data Memory 还是从 Perpheral Control 中读取/写入数据。

```
assign o_control_read_data = reset?32'b0:
    (~i_control_read)?32'b0:
    (i_address == LED_ADDRESS)?{24'b0, r_led[7:0]}:
    (i_address == DIGITAL_ADDRESS)?{20'b0, r_digital[7:0]}:
    (i_address == SYS_CLK_COUNTER_ADDRESS)?r_clk:
    32'b0;
```

Listing 12: 外设控制器的选择

3 Experiment

3.1 文件结构

```
├── asm % 汇编代码
│   ├── kmp_MARS.asm
│   └── kmp_Pipeline.asm
├── ip % ip core生成文件
│   └── blk_mem_gen_instruction
│       ├── blk_mem_gen_instruction.veo
│       └── blk_mem_gen_instruction.vho
├── README.md
├── report % 报告
│   └── report.pdf
├── settings %.xdc配置文件和.coe文件
│   ├── CPU_fpga.xdc
│   ├── CPU_test.xdc
│   ├── data.coe
│   ├── instruction.coe
│   └── instruction_debug.coe
├── src
│   ├── core % CPU各部件
│   │   ├── ALUControl.v
│   │   ├── ALU.v
│   │   ├── BranchControl.v
│   │   ├── Control.v
│   │   ├── CPU_on_board.v
│   │   ├── CPU.v
│   │   ├── DataMemory_1.v
│   │   ├── DataMemory.v
│   │   ├── EX_Forward.v
│   │   ├── EX_MEM_Register.v
│   │   ├── Hazard.v
│   │   ├── ID_EX_Register.v
│   │   ├── ID_Forward.v
│   │   ├── IF_ID_Register.v
│   │   ├── ImmediateExtension.v
│   │   ├── InstructionMemory_1.v
│   │   ├── InstructionMemory.v
│   │   ├── MEM_Forward.v
│   │   ├── MEM_WB_Register.v
```

```
38 │   │   ├── PC.v
39 │   │   ├── PeripheralControl.v
40 │   │   └── RegisterFile.v
41 │   └── peripheral % 外设
42 │       ├── BCD7.v
43 │       └── LED.v
44 ├── test % testbench
45 │   └── CPU_tb.v
46 └── utils % 杂项脚本
47     ├── char_to_data.py
48     ├── gen_data.py
49     └── gen_instruction.py
```

Listing 13: 文件结构

3.2 数据初始化

由于 Instruction Memory 和 Data Memory 均由 IP Core 生成，所以我们可以使用.coe 文件对其进行初始化。Instruction Memory 的初始化文件为 instruction.coe，Data Memory 的初始化文件为 data.coe。

对于 instruction.coe，我们可以直接使用 Mars 生成汇编文件的机器码；对于 data.coe，我们需要将字符串转换为对应的 ASCII 码，且子字符串和母字符串的末尾均以换行符（ASCII 值为 20）结尾。

3.3 汇编代码简介

本项目使用 KMP 算法实现子字符串的查找。大致思路如下：

(1) 从数据存储器中读入字符串。

(2) 生成子字符串的 next 数组。

(3) 根据 next 数组进行匹配。

(4) 将计算结果显示在外设上。

这里重点介绍软件译码的过程。首先，我们需要将结果的 4 位全部提取出来：

```
1  addi $s0, $v0, 0 # s0 is the raw result
2  sll $s1, $s0, 28
3  srl $s1, $s1, 28 # get the first 4 bits
4  sll $s2, $s0, 24
5  srl $s2, $s2, 28 #
6  sll $s3, $s0, 20
7  srl $s3, $s3, 28 #
8  sll $s4, $s0, 16
9  srl $s4, $s4, 28 # get the last 4 bits
```

Listing 14: 提取各位数字

之后显示各位数字。由于 bcd 的控制信号长 12 位，高四位分别控制对应数码管的开与关。因此，如果想控制不同的数码管，只需要将生成的数字信号分别与 0x100、0x200、0x400、0x800 相加即可。

与此同时，为了实现“每位显示 1ms”的效果，可以借助系统时钟计数器，在时钟周期为 100ns 的情况下，时钟计数器每计数 10000 次，显示状态就会发生一次跳转（显示下一个位数）。

```
    # write the BCD
    lui $s6, 0x4000
    addi $s6, $s6, 0x0014
    addi $t5, $0, 10000
    addi $t6, $0, 20000
    addi $t7, $0, 30000

display_1:
    addi $t4, $0, 0x100
    addi $a0, $s1, 0
    jal bcd
    lw $s7, 0($s6) # read the num of sys_clk nums
    sub $t3, $s7, $t5
    bgtz $t3, display_2
    j display_1
display_2:
    addi $t4, $0, 0x200
    addi $a0, $s2, 0
    jal bcd
    lw $s7, 0($s6)
    sub $t3, $s7, $t6
    bgtz $t3, display_3
    j display_2
display_3:
    addi $t4, $0, 0x400
    addi $a0, $s3, 0
    jal bcd
    lw $s7, 0($s6)
    sub $t3, $s7, $t7
    bgtz $t3, display_4
    j display_3
display_4:
    addi $t4, $0, 0x800
    addi $a0, $s4, 0
    jal bcd
    lw $s7, 0($s6)
    sub $t3, $s7, $t5
    bltz $t3, display_1
    j display_4
```

Listing 15: 译码（1）

其中，具体的软件译码如下：

```
bcd:
    addi $t0, $0, 0
    beq  $a0, $t0, bcd_0
    addi $t0, $0, 1
    beq  $a0, $t0, bcd_1
    addi $t0, $0, 2
    beq  $a0, $t0, bcd_2
    addi $t0, $0, 3
...

bcd_0:
    addi $t1, $0, 0x3F
    j write_bcd
bcd_1:
    addi $t1, $0, 0x06
    j write_bcd
```

```
bcd_2:
    addi $t1, $0, 0x5B
    j write_bcd
bcd_3:
    addi $t1, $0, 0x4F
    j write_bcd
...
```

Listing 16: 译码（2）

其中，0x3F、0x06 等代表数码管显示的端口。

3.4 资源及时序分析

本 CPU 的资源使用情况如下图：

使用 100MHz 的时钟进行仿真，结果如下：

Resource	Utilization	Available	Utilization %
LUT	3747	20800	18.01
LUTRAM	256	9600	2.67
FF	2068	41600	4.97
IO	22	210	10.48

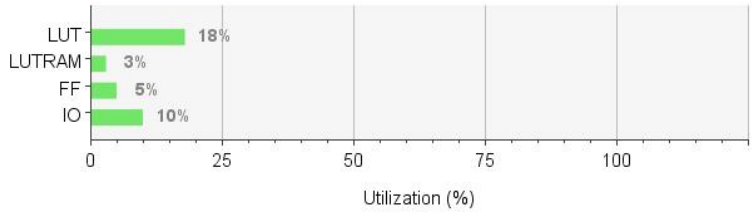

图 7: 资源分析总结

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Bonded IOB (210)	BUFGCTRL (32)
CPU	3747	2068	152	64	22	1
U_ALU (ALU)	530	0	0	0	0	0
U_ALUControl (ALUControl)	10	0	0	0	0	0
U_BCD7 (BCD7)	0	12	0	0	0	0
U_BranchControl (BranchControl)	64	0	0	0	0	0
U_Control (Control)	25	0	0	0	0	0
U_DataMemory (DataMemory)	306	32	128	64	0	0
U_EX_Forward (EX_Forward)	14	0	0	0	0	0
U_EX_MEM_Register (EX_MEM_Register)	429	286	0	0	0	0
U_Hazard (Hazard)	22	0	0	0	0	0
U_ID_EX_Register (ID_EX_Register)	498	332	0	0	0	0
U_ID_Forward (ID_Forward)	16	0	0	0	0	0
U_IF_ID_Register (IF_ID_Register)	33	64	0	0	0	0
U_ImmediateExtension (ImmediateExtension)	33	0	0	0	0	0
U_InstructionMemory (InstructionMemory)	144	0	24	0	0	0
U_MEM_Forward (MEM_Forward)	5	0	0	0	0	0
U_MEM_WB_Register (MEM_WB_Register)	423	282	0	0	0	0
U_PC (PC)	1	32	0	0	0	0
U_PeripheralControl (PeripheralControl)	75	36	0	0	0	0
U_RegisterFile (RegisterFile)	749	992	0	0	0	0

图 8: 资源分析

可以看出，本项目一共占用了大约 10% 的板上资源。

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.262 ns	Worst Hold Slack (WHS):	0.158 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	5233	Total Number of Endpoints:	5233	Total Number of Endpoints:	1855

All user specified timing constraints are met.

图 9: 时序分析

可以计算得出，本 CPU 的主频约为 102.7MHz。

3.5 仿真验证及 CPI 分析

在本实验中，母字符串为 “linux is not unix is not unix is not unix”，子字符串为 “unix”，理论正确结果为 3。

仿真波形如下：

图 10: 仿真

可以看到，数码管交替输出 14f、23f、43f、83f，也就是个位、十位、百位、千位分别输出 3、0、0、0，这表明仿真正确。

由于我在设计时已经考虑了所有可能的数据转发与冒险，因此在编写汇编程序时，无需插入额外的 nop 指令。如果忽略流水线的预热，那么流水线额外执行的周期数仅取决于 Hazard Unit 产生的高电平数 (o_IF_ID_keep||o_ID_EX_flush||o_IF_ID_flush)。执行完毕整个程序（指得到子字符串个数）共需要 1301 个周期，而高电平数为 369，因此 CPI 为 1.396。考虑到汇编程序并没有做重排优化，如果重排优化之后，CPI 应该会更低。

4 Conclusion

本实验通过在 FPGA 上实现 5 级流水线 CPU，最终 CPU 的主频达到了 102.7MHz，CPI 达到了 1.396。但本实验并没有实现动态分支预测、延迟槽、锁相环等更加高级的功能，稍显遗憾。感谢老师和助教在这门课上的付出！