西安交通大學

XI'AN JIAOTONG UNIVERSITY

本科实验报告

CPU 设计

课程名称：	计算机组成原理
姓名/学号：	张航宇　2193214567
	倪鹭洋　2216113072
学院：	电信学部
专业：	计试 2101
指导老师：	张克旺

2024 年 1 月 8 日

西安交通大學实验报告

专业：计试 2101
姓名：张航宇
倪鹭洋
日期：2024 年 1 月 8 日

课程名称：计算机组成原理　指导老师：张克旺　成绩：
实验名称：CPU 设计　实验类型：课程大作业　同组学生姓名：

1 实验概述

1.1 实验要求以及目的

实验要求 完成单周期/多周期 CPU 设计与仿真，要求对于单周期和多周期至少完成 10 条指令，尽可能多实现 MIPS32 的指令，并且完善 ALU 的功能。

实验目地

(1) 熟悉 MIPS 处理器指令集

(2) 学习 Verilog 语言，设计各逻辑模块和测试模块

(3) 单周期/多周期 CPU 的数据通路和 CU 的设计

1.2 实验环境

本组使用 VSCode 进行开发和设计，使用 ModelSim 进行调试和仿真；除此之外，我们还利用 Compiler Explorer[GodBolt, 2024] 编写简单的 C++ 程序，并生成 MIPS32 汇编代码，在 Mars 查看机器代码并进行结果比较。

1.3 实验结果简述

我们实现的指令主要包括 R 型指令，分支指令和跳转指令。为使指令正确运行，我们在书中示例的数据通路外 [Wang and Zhang, 2021]，我们还添加了额外的多路选择器，与门或门等逻辑元件；同时我们还为单周期设计了硬布线控制器，为多周期设计了转换状态图并使用 PLA 实现了控制单元。在完善 CPU 数据通路和 ALU 功能的基础上，本组共设计 31 条指令，包括单周期和多周期 CPU。

我们在章节 2详细分析每一个逻辑元件的设计思路和具体实现。接着，我们在章节 3设计对于单周期和多周期 CPU 的 testbench。最后我们在章节 4展示我们实验的结果，并分析本次实验的不足之处。

2 功能模块设计

为达到设计的模块化，我们将所有的功能模块的输入输出均使用 wire 类型。同时，我们在设计功能模块时希望不管是多周期还是单周期都能复用这些元件而不是重新设计。

寄存器设计 该寄存器文件中包含了单周期多周期中所有要使用的寄存器，包含 RF 寄存器文件，PC，IR，SR 状态寄存器以及 TemporaryReg(一种 32bits 寄存器，由于不需要额外控制信号，所以用一个模型)。这些寄存器均使用 always 语句块赋值，当时钟上升沿以及有相应写信号时写入；对于输出，wire 类型直接使用 assign 语句绑定其需要输出的值即可。此外，在 RF 中，除 32 个通用寄存器，我们还设计了用于乘除运算的 **HI**，**LO** 两个寄存器。

具体的参考代码如下：

```
module RegisterFile (
  input wire [4:0] readReg1, // Read register 1 address
  input wire [4:0] readReg2, // Read register 2 address
  input wire [4:0] writeReg, // Write register address
  input wire [31:0] writeData, // Data to write
  // below are added
  input wire [31:0] highresult,
  input wire [31:0] lowresult,
  // above are added
  input wire writeEnable, // Write enable signal
  input wire hilowrite,
  input wire clk, // Clock signal
  output wire [31:0] readData1, // Data from read register 1
  output wire [31:0] readData2 // Data from read register 2
);
  reg [31:0] registers [31:0]; // 32 32-bit registers
  reg [31:0] HI;
  reg [31:0] LO;
  integer i;

  // initial begin
  //   // Initialize all registers to 0
  //   for (i = 0; i < 32; i = i + 1) begin
  //     registers[i] = 32'h00000001;
  //   end
  // end

  assign readData1 = registers[readReg1];
  assign readData2 = registers[readReg2];

  always @(posedge clk) begin
    // Write data to the register if writeEnable is high
    if (writeEnable) begin
      registers[writeReg] <= writeData;
    end

    if (hilowrite) begin
      HI <= highresult;
      LO <= lowresult;
    end
  end
endmodule
```

```
module PC(
  input wire [31:0] nextInstructionAddress,
  input wire pcWrite,
  input wire clk,
  output wire [31:0] currentInstructionAddress
);
  reg [31:0] programCounter;

  initial begin
    programCounter <= 32'h00000000;
  end

  always @(posedge clk) begin
    if (pcWrite) begin
      #0.1
      programCounter <= nextInstructionAddress;
    end
  end

  assign currentInstructionAddress = programCounter;

endmodule

module InstructionRegister(
  input wire [31:0] nextInstruction,
  input wire irWrite,
  input wire clk,
  output wire [31:0] currentInstruction
);
  reg [31:0] instructionRegister;

  initial begin
    instructionRegister <= 32'h00000000;
  end

  always @(posedge clk) begin
    if (irWrite) begin
      instructionRegister <= nextInstruction;
    end
  end

  assign currentInstruction = instructionRegister;

endmodule


module StateRegister(
  input wire [3:0] nextInstructionAddress,
  input wire clk,
```

```
  output wire [3:0] currentInstructionAddress
);
  reg [3:0] currentStateRegister;

  assign currentInstructionAddress = currentStateRegister;

  initial begin
    currentStateRegister <= 32'h00000000;
  end

  always @(posedge clk) begin
    currentStateRegister <= nextInstructionAddress;
  end
endmodule

module TemporaryRegister_32bit(
  input wire [31:0] nextData,
  input wire clk,
  output wire [31:0] currentData
);
  reg [31:0] temporaryRegister;

  initial begin
    temporaryRegister <= 32'h00000000;
  end

  always @(posedge clk) begin
    temporaryRegister <= nextData;
  end

  assign currentData = temporaryRegister;

endmodule
```

内存 由于我们无法实现真正模拟内存的结构，比如让其读写文件。所以我们采用了一种使用寄存器模拟内存的方式工作。因此，内存的设计与寄存器的设计类似，这里不再过多赘述。

考虑到我们所涉及的所有指令均为单机器字长的指令，且所有指令的实际操作数均为 32bits 的数据，因此我们简单的使用了按字编址设计了存储单元的结构。这样在访问内存时，我们可以只使用 [31 2] 这一范围。同时，因为 ModelSim 所能查看的波形的数量有限，我们并没有设计 GB 大小的内存，而是指开辟了 1k*32bits 的内存。

具体的参考代码如下：

```
module InstructionMemory(
    input wire [31:0] address,
    output wire [31:0] instruction
);
    reg [31:0] memory [0:1023];
```

```
    assign instruction = memory[address[11:2]];

endmodule

module DataMemory(
    input wire [31:0] address,
    input wire [31:0] writeData,
    input wire writeEnable,
    input wire readEnable,
    output wire [31:0] readData
);
    reg [31:0] memory [0:1023];
    reg [31:0] readReg;

    integer i;

    assign readData = readReg;

    always @* begin
        if(readEnable) begin
            readReg = memory[address[11:2]];
        end

        if (writeEnable) begin
            memory[address[11:2]] <= writeData;
        end
    end

endmodule
```

多路选择器 多路选择器是一种选择所要传送的数据的组合逻辑元件，不需要特别的设计。我们使用"?:" 三目运算符根据选择信号的不同选择不同的输出数据。由于数据通路上通常由不同数量，不同宽度的数据需要被选择，因此我们设计了不同版本的多路选择器。

具体的参考代码如下：

```
module MUX2To1_4bit(
    input wire [3:0] i_a,
    input wire [3:0] i_b,
    input wire i_s,
    output wire [3:0] o_y
);
    assign o_y = i_s ? i_b : i_a;
endmodule

module MUX2To1_5bit(
    input wire [4:0] i_a,
    input wire [4:0] i_b,
    input wire i_s,
```

```
    output wire [4:0] o_y
);
    assign o_y = i_s ? i_b : i_a;
endmodule

module MUX4To1_5bit(
    input wire [4:0] i_a,
    input wire [4:0] i_b,
    input wire [4:0] i_c,
    input wire [4:0] i_d,
    input wire [1:0] i_s,
    output wire [4:0] o_y
);
    assign o_y = (i_s == 2'b00) ? i_a : (i_s == 2'b01) ? i_b : (i_s == 2'b10) ? i_c : i_d;

endmodule

module MUX2To1_8bit(
    input wire [7:0] i_a,
    input wire [7:0] i_b,
    input wire i_s,
    output wire [7:0] o_y
);
    assign o_y = i_s ? i_b : i_a;
endmodule

module MUX2To1_16bit(
    input wire [15:0] i_a,
    input wire [15:0] i_b,
    input wire i_s,
    output wire [15:0] o_y
);
    assign o_y = i_s ? i_b : i_a;
endmodule

module MUX2To1_32bit(
    input wire [31:0] i_a,
    input wire [31:0] i_b,
    input wire i_s,
    output wire [31:0] o_y
);
    assign o_y = i_s ? i_b : i_a;
endmodule

module MUX4To1_32bit(
    input wire [31:0] i_a,
    input wire [31:0] i_b,
    input wire [31:0] i_c,
    input wire [31:0] i_d,
    input wire [1:0] i_s,
```

```
    output wire [31:0] o_y
);
    assign o_y = (i_s == 2'b00) ? i_a : (i_s == 2'b01) ? i_b : (i_s == 2'b10) ? i_c : i_d;

endmodule

module MUX5To1_32bit(
    input wire [31:0] i_a,
    input wire [31:0] i_b,
    input wire [31:0] i_c,
    input wire [31:0] i_d,
    input wire [31:0] i_e,
    input wire [2:0] i_s,
    output wire [31:0] o_y
);
    assign o_y = (i_s == 3'b000) ? i_a : (i_s == 3'b001) ? i_b : (i_s == 3'b010) ? i_c :
        (i_s == 3'b011) ? i_d : i_e;

endmodule

module MUX3To1_32bit(
    input wire [31:0] i_a,
    input wire [31:0] i_b,
    input wire [31:0] i_c,
    input wire [1:0] i_s,
    output wire [31:0] o_y
);
    assign o_y = (i_s == 2'b00) ? i_a : (i_s == 2'b01) ? i_b : i_c;

endmodule

module MUX3To1_5bit(
    input wire [4:0] i_a,
    input wire [4:0] i_b,
    input wire [4:0] i_c,
    input wire [1:0] i_s,
    output wire [4:0] o_y
);
    assign o_y = (i_s == 2'b00) ? i_a : (i_s == 2'b01) ? i_b : i_c;

endmodule
```

符号拓展 符号拓展元件最重要的就是根据数据的最高为来进行拓展，结构如下：

```
module SigExt16To32bit(
    input wire [15:0] i_a,
    output wire [31:0] o_y
);
    assign o_y = {{16{i_a[15]}}, i_a};
```

```
endmodule

module SigExt5To32bit(
    input wire [4:0] i_a,
    output wire [31:0] o_y
);
    assign o_y = {{27{i_a[4]}}, i_a};
endmodule
```

分支运算器 分支运算器是为支持更多的分支指令独创的逻辑元件。其设计思路对于多种不同分支所用的各种逻辑门的整合。

首先，不同的分支指令需要用到 PSW 中不同的位，因此其发送的 Branch 微命令也是不同的。而如果使用逻辑门在 CPU 的数据通路上进行连接的话会使数据通路的结构较为复杂，不适于维护添加指令，因此我们使用逻辑模块包裹了所有需要用到的逻辑门根据所有的条件是否存在至少一种满足而进行分支跳转。

其参考代码如下：

```
module SingleBranch(
    input wire [5:0] branch,
    input wire [3:0] psw,
    output wire branchpc
);
    assign branchpc = (branch[5] & psw[0]) | (branch[4] & ~psw[0]) | (branch[3] & ~(psw[3]
        ^ psw[2])) | (branch[2] & (psw[3] ^ psw[2])) |
                    (branch[1] & ~(psw[3] ^ psw[2]) & ~psw[0]) | (branch[0] & ((psw[3] ^
                        psw[2]) | psw[0]));
endmodule
```

ALU 我们设计的 ALU 共实现了 13 中功能，包括有无符号的加减乘除运算，移位，逻辑运算等。对于逻辑运算我们直接使用了 Verilog 提供的运算符运算，而加法器，减法器和有符号的除法器我们自己实现了简易的版本进行使用，对于这些运算。我们使用一些暂存的寄存器，如 resultReg 等暂存这些值。并使最终的输出 wire 连接到这些寄存器，而这些寄存器的输入值则通过 case 语句进行选择。即所有的加法，减法，乘除等操作同时进行，并产生结果，根据 ALUCtrl 信号的 case 语句选择相应的结果数据输出到这些暂存寄存器。ALUCtrl 信号的具体表格如下：

ALUCtrl	操作	ALUCtrl	操作
0000	add	0111	xor
0001	sub	1000	not
0010	sar	1001	imul
0011	shr	1010	idiv
0100	sal	1011	slt
0101	and	1100	addu
0110	or		

表 1: ALUCtrl 与操作对应图

这里我们只展示 ALU 的参考代码，其他诸如加法器等可以在完整的代码中查看：

```
// sign overflow carry zero
module ALU (
  input wire [31:0] operandA,
  input wire [31:0] operandB,
  input wire [4:0] aluCtrl,
  output wire [31:0] result,
  // below are added
  output wire [31:0] highresult,
  output wire [31:0] lowresult,
  // above are added
  output wire [3:0] psw,
  output wire [31:0] test_tmp
);
    reg [31:0] resultReg,pswReg;
    // below are added
    reg [31:0] highresultReg, lowresultReg;
    // above are added
    wire [31:0] resultAdd,resultSub,resultdivuhi,resultdivulo;
    wire carryAdd, carrySub;

    Accumulator32bit add(operandA,operandB,1'b0,resultAdd,carryAdd);
    Accumulator32bit sub(operandA,~operandB + 1,1'b0,resultSub,carrySub);
    signed_divider divu(operandA, operandB,resultdivulo,resultdivuhi);

    assign result = resultReg;
    // below are added
    assign highresult = highresultReg;
    assign lowresult = lowresultReg;
    // above are added
    assign psw = pswReg;

    always @* begin
    case (aluCtrl)
        5'b00000: begin
            // Perform addition using the instantiated module
          resultReg = resultAdd;
          pswReg[1] = carryAdd;
          pswReg[2] = (resultReg[31] != operandA[31] && resultReg[31] != operandB[31]); //
              Check if sign overflow
          pswReg[3] = resultReg[31]; // Check if result is negative
        end
        5'b00001: begin
            // Perform subtraction using the instantiated module
          resultReg = resultSub;
          pswReg[1] = operandA < operandB;
          pswReg[2] = (~resultReg[31] && operandA[31] && ~operandB[31] || resultReg[31] &&
              ~operandA[31] && operandB[31]); // Check if sign overflow
          pswReg[3] = resultReg[31]; // Check if result is negative
```

```
    end
    5'b00010: begin
        // Perform sar
      resultReg = operandB >> operandA;
    end
    5'b00011: begin
        // Perform shr
      resultReg = $signed(operandB) >>> operandA;
    end
    5'b00100: begin
        // Perform sal
      resultReg = operandB << operandA;
    end
    5'b00101: begin
       // Perform bitwise AND
      resultReg = operandA & operandB;
    end
    5'b00110: begin
       // Perform bitwise OR
      resultReg = operandA | operandB;
    end
    5'b00111: begin
       // Perform bitwise XOR
      resultReg = operandA ^ operandB;
    end
    5'b01000: begin
       // Perform sltu
     resultReg = operandA < operandB ? 32'b1 : 32'b0;
    end
    5'b01001: begin
       // Perform mul
      resultReg = operandA * operandB;
      {highresultReg, lowresultReg} = $signed(operandA) * $signed(operandB);
    end
    5'b01010: begin
       // Perform div
     resultReg = resultdivulo;
     lowresultReg = resultdivulo;
     highresultReg = resultdivuhi;
    end
    5'b01011: begin
       // Perform slt
      resultReg = {31'h00000000,resultSub[31]};
      pswReg[3] = resultSub[31];
    end
    5'b01100: begin
       // addu
      resultReg = resultAdd;
      pswReg[1] = carryAdd;
    end
```

```
      5'b01101: begin
          // Perform divu
         resultReg = operandA / operandB;
         // added:
         lowresultReg = operandA / operandB;
         highresultReg = operandA % operandB;
      end
      5'b01110: begin
          // Perform multu
         resultReg = operandA * operandB;
         {highresultReg, lowresultReg} = operandA * operandB;
      end
      5'b01111: begin
          // Perform nor
         resultReg = ~(operandA | operandB);
      end
      5'b10001: begin
          // Perform subu
         resultReg = resultSub;
         pswReg[1] = operandA < operandB;
      end
   endcase
   pswReg[0] = (resultReg == 32'b0); // Check if result is zero
  end

endmodule
```

CU CU 主要分为两个部分，第一个是 ALUCU，第二个是 CU 对于 ALUCU，其主要通过 ALUOp 和 func(Inst(5 0)) 的值给 ALU 发送相应的控制信号，操作表如表 1。对于 CU，单周期和多周期设计有些区别，单周期 CU 我们使用硬布线的方式设计，具体的控制信号表如下：

对于多周期，我们使用 PLA 设计，其控制信号表过于繁杂，我们这里只展示所有指令相应的状态转换图 (图片 1)

参考代码如下：

```
/*
| nums | aluctrl | operate |
| ---- | ------- | ------- |
|   1  | 00000 | add   |
|   2  | 00001 | sub   |
| 3.4 | 00010 | srl(v) |
| 5.6 | 00011 | sra(v) |
| 7.8 | 00100 | sll(v) |
|   9  | 00101 | and   |
|  10  | 00110 | or    |
|  11  | 00111 | xor   |
|  12  | 01000 | sltu  |
|  13  | 01001 | mult  |
|  14  | 01010 | div   |
```

	SignalName	typeR	Rdm	Rshamt	lw	sw	beq	bne	bgez	bltz	bgtz	blez	j	jr	jal
Input	OP5	0	0	0	1	1	0	0	0	0	0	0	0	0	0
	OP4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	OP3	0	0	0	0	1	0	0	0	0	0	0	0	0	0
	OP2	0	0	0	0	0	1	1	0	0	1	1	0	0	0
	OP1	0	0	0	1	1	0	0	0	0	1	1	1	0	1
	OP0	0	0	0	1	1	0	1	1	1	1	0	0	0	1
Output	HILOWr	0	1	0	0	0	0	0	0	0	0	0	0	0	0
	RegDst1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	RegDst0	1	0	1	0	x	x	x	x	x	x	x	x	x	0
	ALUSrcA	0	0	1	0	0	0	0	0	0	0	0	0	x	0
	ALUSrcB	0	0	0	0	0	0	0	1	1	1	1	0	x	0
	ALUSrcB0	0	0	0	1	1	0	0	0	0	0	0	x	x	0
	writetoReg1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	writetoReg0	0	0	0	1	x	x	x	x	x	x	x	x	x	x
	RegWr	1	0	1	1	0	0	0	0	0	0	0	0	0	1
	MemWr	0	0	0	0	1	0	0	0	0	0	0	0	0	0
	MemRd	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	Branch5	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	Branch4	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	Branch3	0	0	0	0	0	0	0	1	0	0	0	0	0	0
	Branch2	0	0	0	0	0	0	0	0	1	0	0	0	0	0
	Branch1	0	0	0	0	0	0	0	0	0	1	0	0	0	0
	Branch0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
	Jump1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	Jump0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
	ALUOP1	1	1	1	0	0	0	0	0	0	0	0	x	1	1
	ALUOP0	0	0	0	0	0	1	1	1	1	1	1	x	0	0

表 2: 单周期 CU 控制信号表

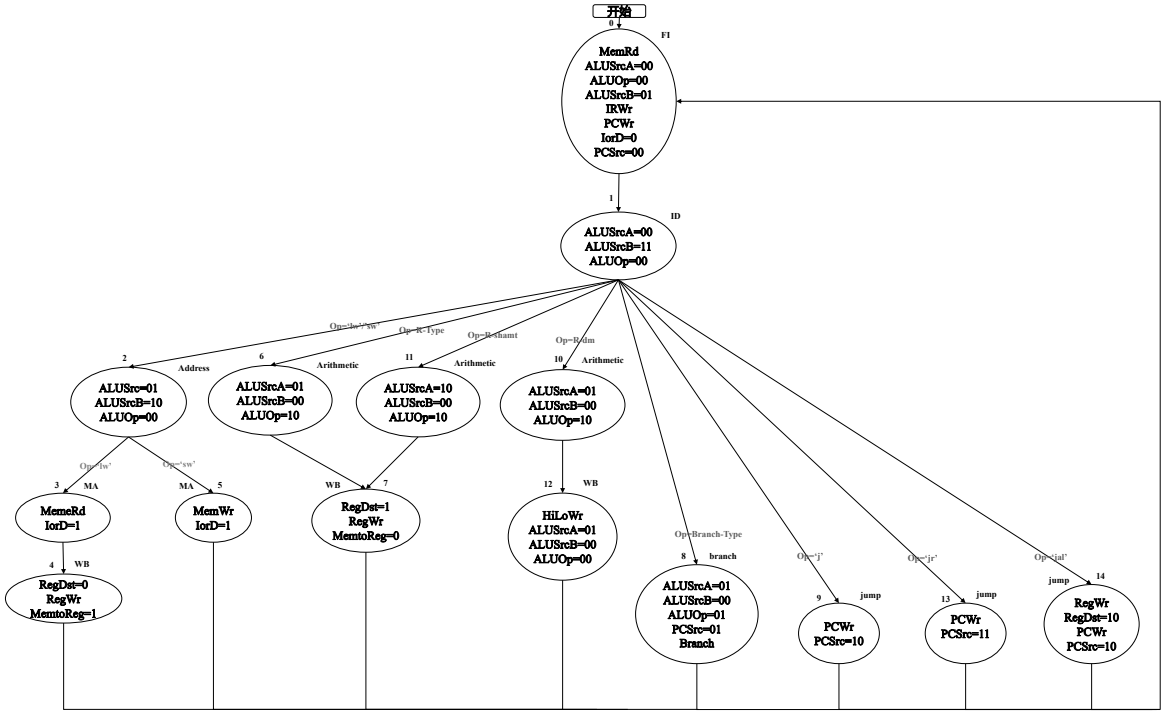

图 1: 多周期状态图

```
| 15 | 01011 | slt   |
| 16 | 01100 | addu  |
| 17 | 01101 | divu  |
| 18 | 01110 | multu |
| 19 | 01111 | nor   |
| 20 | 10001 | subu  |
*/

module ALUCU(
    input wire [5:0] func,
    input wire [1:0] aluOP,
    output wire [4:0] aluCtrl
);
    reg [4:0] aluCtrlReg;
    assign aluCtrl = aluCtrlReg;

    always @* begin
        if(aluOP == 2'b00) begin
            aluCtrlReg = 5'b00000;
        end
        else if (aluOP == 2'b01) begin
            aluCtrlReg = 5'b00001;
        end
        else begin
```

```
        case (func)
            6'b100000: begin //add  1
                aluCtrlReg = 5'b00000;
            end
            6'b100001: begin //addu 2
                aluCtrlReg = 5'b01100;
            end
            6'b100010: begin //sub  3
                aluCtrlReg = 5'b00001;
            end
            6'b100100: begin //and  4
                aluCtrlReg = 5'b00101;
            end
            6'b100101: begin //or   5
                aluCtrlReg = 5'b00110;
            end
            6'b100110: begin //xor  6
                aluCtrlReg = 5'b00111;
            end
            6'b100111: begin //nor  7
                aluCtrlReg = 5'b01111;
            end
            6'b101010: begin //slt  8
                aluCtrlReg = 5'b01011;
            end
            6'b101011: begin //sltu 9
                aluCtrlReg = 5'b01000;
            end
            6'b011010: begin //div  10
                aluCtrlReg = 5'b01010;
            end
            6'b011011: begin //divu 11
                aluCtrlReg = 5'b01101;
            end
            6'b011000: begin //mult 12
                aluCtrlReg = 5'b01001;
            end
            6'b011001: begin //multu 13
                aluCtrlReg = 5'b01110;
            end
            6'b000100: begin //sllv 14
                aluCtrlReg = 5'b00100;
            end
            6'b000110: begin //srlv 15
                aluCtrlReg = 5'b00010;
            end
            6'b000111: begin //srav 16
                aluCtrlReg = 5'b00011;
            end
            6'b000000: begin //sll  17
```

```
                aluCtrlReg = 5'b00100;
            end
            6'b000010: begin //srl  18
                aluCtrlReg = 5'b00010;
            end
            6'b000011: begin //sra  19
                aluCtrlReg = 5'b00011;
            end
            6'b100011: begin //subu 20
                aluCtrlReg = 5'b10001;
            end
        endcase
    end
end

endmodule

/// CU function table is in file `CU_function_table.xlsx`
module MCU(
    input wire [5:0] opCode,
    input wire [4:0] bCode,
    input wire [5:0] funct,
    input wire clk,
    output wire [1:0] regDst,
    output wire [1:0] jump,
    output wire regWrite,
    output wire hiloWrite,
    output wire [5:0] branch, // added
    output wire [1:0] writeToReg,
    output wire [1:0] aluOP,
    output wire memRead,
    output wire memWrite,
    output wire aluSrcA,
    output wire [1:0] aluSrcB
);
    wire typeR = ~opCode[5] & ~opCode[4] & ~opCode[3] & ~opCode[2] & ~opCode[1] &
        ~opCode[0];
    wire lw = opCode[5] & ~opCode[4] & ~opCode[3] & ~opCode[2] & opCode[1] & opCode[0];
    wire sw = opCode[5] & ~opCode[4] & opCode[3] & ~opCode[2] & opCode[1] & opCode[0];
    wire beq = ~opCode[5] & ~opCode[4] & ~opCode[3] & opCode[2] & ~opCode[1] & ~opCode[0];
    // bne wire added
    wire bne = ~opCode[5] & ~opCode[4] & ~opCode[3] & opCode[2] & ~opCode[1] & opCode[0];
    wire j = ~opCode[5] & ~opCode[4] & ~opCode[3] & ~opCode[2] & opCode[1] & ~opCode[0];

    //bgez or bltz
    wire bgezorlte = ~opCode[5] & ~opCode[4] & ~opCode[3] & ~opCode[2] & ~opCode[1] &
        opCode[0];
    wire bgtz = ~opCode[5] & ~opCode[4] & ~opCode[3] & opCode[2] & opCode[1] & opCode[0];
    wire blez = ~opCode[5] & ~opCode[4] & ~opCode[3] & opCode[2] & opCode[1] & ~opCode[0];
    wire typeRdm = typeR & ~funct[5] & funct[4] & funct[3];
```

```
    wire typeRshamt = typeR & ~funct[5] & ~funct[4] & ~funct[3] & ~funct[2];
    wire jr = typeR & ~funct[5] & ~funct[4] & funct[3] & ~funct[2] & ~funct[1] & ~funct[0];
    wire jal = ~opCode[5] & ~opCode[4] & ~opCode[3] & ~opCode[2] & opCode[1] & opCode[0];

    assign regDst[0] = typeR & ~typeRdm;
    assign regDst[1] = jal;
    assign aluSrcB[1] = bgtz | blez | bgezorlte;
    assign aluSrcB[0] = lw | sw;
    assign aluSrcA = typeRshamt;
    assign writeToReg[0] = lw;
    assign writeToReg[1] = jal;
    assign regWrite = (typeR | lw | jal) & ~typeRdm & ~jr;
    assign hiloWrite = typeRdm;

    assign memRead = lw;
    assign memWrite = sw;
    assign branch[5] = beq; // beq
    assign branch[4] = bne; // bne
    assign branch[3] = bgezorlte & bCode[0]; // bgez
    assign branch[2] = bgezorlte & ~bCode[0]; // blte
    assign branch[1] = bgtz; // bgtz
    assign branch[0] = blez; // blez

    assign jump[0] = j | jal;
    assign jump[1] = jr;
    assign aluOP[1] = typeR;
    assign aluOP[0] = beq | bne | blez | bgezorlte | bgtz; // bne added

endmodule

module MCUMutipleCycle
(
    input wire [5:0] opCode,
    input wire [4:0] bCode,
    input wire [5:0] func,
    input wire [3:0] currentState,
    input wire clk,
    output wire IorD,
    output wire irWrite,
    output wire pcWrite,
    output wire [5:0] branch,
    output wire [1:0] regDst,
    output wire regWrite,
    output wire hilowrite,
    output wire [1:0] aluSrcA,
    output wire [1:0] pcSrc,
    output wire [2:0] aluSrcB,
    output wire memToReg,
    output wire [1:0] aluOP,
    output wire memRead,
```

```
    output wire memWrite,
    output wire [3:0] nextState
);

    ///*** set up temporal logic ***///
    wire [25:0] tmpLogic;
    assign tmpLogic[0] = ~currentState[3] && ~currentState[2] && ~currentState[1] &&
        ~currentState[0]; //0000
    assign tmpLogic[1] = ~currentState[3] && ~currentState[2] && ~currentState[1] &&
        currentState[0]; //0001
    assign tmpLogic[2] = ~currentState[3] && ~currentState[2] && currentState[1] &&
        ~currentState[0]; //0010
    assign tmpLogic[3] = ~currentState[3] && ~currentState[2] && currentState[1] &&
        currentState[0]; //0011
    assign tmpLogic[4] = ~currentState[3] && currentState[2] && ~currentState[1] &&
        ~currentState[0]; //0100
    assign tmpLogic[5] = ~currentState[3] && currentState[2] && ~currentState[1] &&
        currentState[0]; //0101
    assign tmpLogic[6] = ~currentState[3] && currentState[2] && currentState[1] &&
        ~currentState[0]; //0110
    assign tmpLogic[7] = ~currentState[3] && currentState[2] && currentState[1] &&
        currentState[0]; //0111
    assign tmpLogic[8] = currentState[3] && ~currentState[2] && ~currentState[1] &&
        ~currentState[0]; //1000
    assign tmpLogic[9] = currentState[3] && ~currentState[2] && ~currentState[1] &&
        currentState[0]; //1001
    // added
    assign tmpLogic[10] = currentState[3] && ~currentState[2] && currentState[1] &&
        ~currentState[0]; //1010
    assign tmpLogic[11] = currentState[3] && ~currentState[2] && currentState[1] &&
        currentState[0]; //1011
    assign tmpLogic[12] = currentState[3] && currentState[2] && ~currentState[1] &&
        ~currentState[0]; //1100
    assign tmpLogic[13] = currentState[3] && currentState[2] && ~currentState[1] &&
        currentState[0]; //1101
    assign tmpLogic[14] = currentState[3] && currentState[2] && currentState[1] &&
        ~currentState[0]; //1110

    assign tmpLogic[15] = ~opCode[5] && ~opCode[4] && ~opCode[3] && ~opCode[2] && opCode[1]
        && ~opCode[0] && ~currentState[3] && ~currentState[2] && ~currentState[1] &&
        currentState[0]; //000010 0001 -> j 0001

    assign tmpLogic[16] = ~opCode[5] && ~opCode[4] && ~opCode[3] && opCode[2] && ~opCode[1]
        && ~opCode[0] && ~currentState[3] && ~currentState[2] && ~currentState[1] &&
        currentState[0] ||
                      ~opCode[5] && ~opCode[4] && ~opCode[3] && opCode[2] && ~opCode[1] &&
                          opCode[0] && ~currentState[3] && ~currentState[2] &&
                          ~currentState[1] && currentState[0] ||
                      ~opCode[5] && ~opCode[4] && ~opCode[3] && ~opCode[2] && ~opCode[1]
                          && opCode[0] && ~currentState[3] && ~currentState[2] &&
```

```
                    ~currentState[1] && currentState[0] ||
                ~opCode[5] && ~opCode[4] && ~opCode[3] && opCode[2] && opCode[1] &&
                    opCode[0] && ~currentState[3] && ~currentState[2] &&
                    ~currentState[1] && currentState[0] ||
                ~opCode[5] && ~opCode[4] && ~opCode[3] && opCode[2] && opCode[1] &&
                    ~opCode[0] && ~currentState[3] && ~currentState[2] &&
                    ~currentState[1] && currentState[0] ||
                ~opCode[5] && ~opCode[4] && ~opCode[3] && ~opCode[2] && ~opCode[1]
                    && opCode[0] && ~currentState[3] && ~currentState[2] &&
                    ~currentState[1] && currentState[0]; //000100 0001 -> b 0001



assign tmpLogic[17] = ~opCode[5] && ~opCode[4] && ~opCode[3] && ~opCode[2] &&
    ~opCode[1] && ~opCode[0] && ~currentState[3] && ~currentState[2] &&
    ~currentState[1] && currentState[0] && func[5] && ~func[4] ||
                ~opCode[5] && ~opCode[4] && ~opCode[3] && ~opCode[2] && ~opCode[1]
                    && ~opCode[0] && ~currentState[3] && ~currentState[2] &&
                    ~currentState[1] && currentState[0] && ~func[5] && ~func[4] &&
                    ~func[3] && func[2]; //000000 0001 10xxxx -> R_type 0001
assign tmpLogic[18] = opCode[5] && ~opCode[4] && ~opCode[3] && ~opCode[2] && opCode[1]
    && opCode[0] && ~currentState[3] && ~currentState[2] && currentState[1] &&
    ~currentState[0]; //100011 0010 ->lw 0010
assign tmpLogic[19] = opCode[5] && ~opCode[4] && opCode[3] && ~opCode[2] && opCode[1]
    && opCode[0] && ~currentState[3] && ~currentState[2] && ~currentState[1] &&
    currentState[0]; //101011 0001 ->sw 0001
assign tmpLogic[20] = opCode[5] && ~opCode[4] && ~opCode[3] && ~opCode[2] && opCode[1]
    && opCode[0] && ~currentState[3] && ~currentState[2] && ~currentState[1] &&
    currentState[0]; //100011 0001 ->lw 0001
assign tmpLogic[21] = opCode[5] && ~opCode[4] && opCode[3] && ~opCode[2] && opCode[1]
    && opCode[0] && ~currentState[3] && ~currentState[2] && currentState[1] &&
    ~currentState[0]; //101011 0010 ->sw 0010
assign tmpLogic[22] = ~opCode[5] && ~opCode[4] && ~opCode[3] && ~opCode[2] &&
    ~opCode[1] && ~opCode[0] && ~currentState[3] && ~currentState[2] &&
    ~currentState[1] && currentState[0] && ~func[5] && func[4]; //000000 0001 01xxxx
    -> Rdm 0001
assign tmpLogic[23] = ~opCode[5] && ~opCode[4] && ~opCode[3] && ~opCode[2] &&
    ~opCode[1] && ~opCode[0] && ~currentState[3] && ~currentState[2] &&
    ~currentState[1] && currentState[0] && ~func[5] && ~func[4] && ~func[3] &&
    ~func[2]; //000000 0001 000xxx -> Rshamt 0001
assign tmpLogic[24] = ~opCode[5] && ~opCode[4] && ~opCode[3] && ~opCode[2] &&
    ~opCode[1] && ~opCode[0] && ~currentState[3] && ~currentState[2] &&
    ~currentState[1] && currentState[0] && ~func[5] && ~func[4] && func[3]; //000000
    0001 0010xx -> jr 0001
assign tmpLogic[25] = ~opCode[5] && ~opCode[4] && ~opCode[3] && ~opCode[2] && opCode[1]
    && opCode[0] && ~currentState[3] && ~currentState[2] && ~currentState[1] &&
    currentState[0]; //000011 0001 -> jal 0001


/// *** set up output logic *** ///
```

```
assign pcWrite = tmpLogic[0] || tmpLogic[9] || tmpLogic[13] || tmpLogic[14];
// assign pcWriteCond = tmpLogic[8];
// beq
assign branch[5] = ~opCode[5] & ~opCode[4] & ~opCode[3] & opCode[2] & ~opCode[1] &
    ~opCode[0] & tmpLogic[8];
// bne
assign branch[4] = ~opCode[5] & ~opCode[4] & ~opCode[3] & opCode[2] & ~opCode[1] &
    opCode[0] & tmpLogic[8];
// bgez
assign branch[3] = ~opCode[5] & ~opCode[4] & ~opCode[3] & ~opCode[2] & ~opCode[1] &
    opCode[0] & tmpLogic[8] & bCode[0];
// bgtz
assign branch[2] = ~opCode[5] & ~opCode[4] & ~opCode[3] & opCode[2] & opCode[1] &
    opCode[0] & tmpLogic[8];
// blez
assign branch[1] = ~opCode[5] & ~opCode[4] & ~opCode[3] & opCode[2] & opCode[1] &
    ~opCode[0] & tmpLogic[8];
// bltz
assign branch[0] = ~opCode[5] & ~opCode[4] & ~opCode[3] & ~opCode[2] & ~opCode[1] &
    opCode[0] & tmpLogic[8] & ~bCode[0];

assign IorD = tmpLogic[3] || tmpLogic[5];
assign memRead = tmpLogic[0] || tmpLogic[3];
assign memWrite = tmpLogic[5];
assign irWrite = tmpLogic[0];
assign memToReg = tmpLogic[4];
assign pcSrc[1] = tmpLogic[9] || tmpLogic[13] || tmpLogic[14];
assign pcSrc[0] = tmpLogic[8] || tmpLogic[13];
assign aluOP[1] = tmpLogic[6] || tmpLogic[10] || tmpLogic[11];
assign aluOP[0] = tmpLogic[8];
assign aluSrcB[2] = branch[2] || branch[3] || branch[4] || branch[5];
assign aluSrcB[1] = tmpLogic[1] || tmpLogic[2];
assign aluSrcB[0] = tmpLogic[0] || tmpLogic[1];
assign aluSrcA[0] = tmpLogic[2] || tmpLogic[6] || tmpLogic[8] || tmpLogic[10] ||
    tmpLogic[12];
assign aluSrcA[1] = tmpLogic[11];
assign regWrite = tmpLogic[4] || tmpLogic[7] || tmpLogic[14];
assign regDst[0] = tmpLogic[7];
assign regDst[1] = tmpLogic[14];
assign hilowrite = tmpLogic[12];

assign nextState[3] = tmpLogic[10] || tmpLogic[15] || tmpLogic[16] || tmpLogic[22] ||
    tmpLogic[23] || tmpLogic[24] || tmpLogic[25];
assign nextState[2] = tmpLogic[3] || tmpLogic[6] || tmpLogic[10] || tmpLogic[11] ||
    tmpLogic[17] || tmpLogic[21] || tmpLogic[24] || tmpLogic[25];
assign nextState[1] = tmpLogic[6] || tmpLogic[11] || tmpLogic[17] || tmpLogic[18] ||
    tmpLogic[19] || tmpLogic[20] || tmpLogic[22] || tmpLogic[23] || tmpLogic[25];
assign nextState[0] = tmpLogic[0] || tmpLogic[6] || tmpLogic[11] || tmpLogic[15] ||
    tmpLogic[18] || tmpLogic[21] || tmpLogic[23] || tmpLogic[24];
```

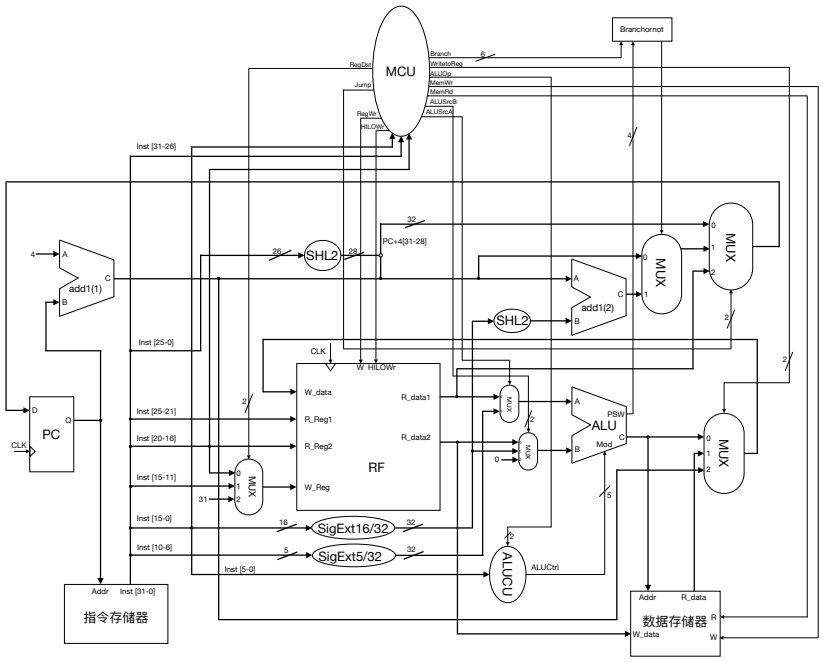

图 2: 单周期数据通路

```
endmodule
```

单周期 CPU 数据通路 对于单周期数据通路的设计，由于其每个阶段所用到的元件均不同，而且存在元件冗余。因此我们在编写代码是也是根据该元件所属的阶段的不同将其放置在不同位置。图片 2展示了具体的数据通路。具体代码如下：

```
module SingleCycleCPU(
    input wire clk
);
    ///*** fetch wire ***///
    wire [31:0] instruction;
    wire [31:0] currentInstructionAddress;
    wire [31:0] nextInstructionAddress;
    wire [31:0] tmpNextInstructionAddress;

    ///*** decode wire ***///
    wire [1:0] regDst;
    wire [1:0] jump;
    wire regWrite;
    wire hiloWrite;
    wire [5:0] branch;

    wire [1:0] writeToReg;
    wire [1:0] aluOP;
    wire memRead;
```

```
wire memWrite;
wire [1:0] aluSrcB;
wire aluSrcA;

wire [4:0] regDstMuxOut;
wire [31:0] regWrData;
wire [31:0] regRdData1;
wire [31:0] regRdData2;

wire [31:0] jumpAddress;
wire [31:0] extendedImmediate,extendedImmediate2;

///*** execute wire ***///
wire [4:0] aluCtrl;
wire [31:0] aluSrcDataA,aluSrcDataB;

wire [3:0] psw;
wire [31:0] aluResult;
wire [31:0] branchAddress;

///*** memory access wire ***///
wire [31:0] memReadData;
wire [31:0] tmpNextBranchInstructionAddress;

///*** fetch instruction ***///
PC pc(nextInstructionAddress, 1'b1, clk, currentInstructionAddress);
wire [3:0] pcAddpsw_useless;
wire [31:0] pcAddtest_tmp_useless;
wire [31:0] pcAddhightest_tmp_useless;
wire [31:0] pcAddlowtest_tmp_useless;
ALU pcAdd(currentInstructionAddress, 32'h00000004, 5'b00000,
    tmpNextInstructionAddress,pcAddhightest_tmp_useless,pcAddlowtest_tmp_useless,pcAddpsw_useless,pcAddtest
InstructionMemory imem(currentInstructionAddress, instruction);

///*** decode instruction ***///
wire [31:0] highresult;
wire [31:0] lowresult;
MCU
    mcu(instruction[31:26],instruction[20:16],instruction[5:0],clk,regDst,jump,regWrite,hiloWrite,branch,wr
MUX3To1_5bit
    regDstMux(instruction[20:16],instruction[15:11],5'b11111,regDst,regDstMuxOut);
RegisterFile
    regFile(instruction[25:21],instruction[20:16],regDstMuxOut,regWrData,highresult,lowresult,regWrite,hilo
assign jumpAddress = {tmpNextInstructionAddress[31:28],instruction[25:0],2'b00};
SigExt16To32bit sigExt16To32(instruction[15:0],extendedImmediate);
SigExt5To32bit sigExt5To32(instruction[10:6],extendedImmediate2);

///*** execute instruction ***///
ALUCU aluCU(instruction[5:0],aluOP,aluCtrl);
MUX3To1_32bit aluSrcBMux(regRdData2,extendedImmediate,32'h00000000,aluSrcB,aluSrcDataB);
```

```
    MUX2To1_32bit aluSrcAMux(regRdData1,extendedImmediate2,aluSrcA,aluSrcDataA);
    wire [31:0] aluAddtest_tmp_useless;
    ALU
        alu(aluSrcDataA,aluSrcDataB,aluCtrl,aluResult,highresult,lowresult,psw,aluAddtest_tmp_useless);
    wire [3:0] branchALUpsw_useless;
    wire [31:0] branchALUtest_tmp_useless;
    wire [31:0] branchALUhightest_tmp_useless;
    wire [31:0] branchALUlowtest_tmp_useless;
    ALU branchALU(tmpNextInstructionAddress,(extendedImmediate <<
        2),5'b00000,branchAddress,branchALUhightest_tmp_useless,branchALUlowtest_tmp_useless,branchALUpsw_usele

    ///*** memory access ***///
    DataMemory dmem(aluResult,regRdData2,memWrite,memRead,memReadData);
    MUX4To1_32bit
        writeToRegMux(aluResult,memReadData,tmpNextInstructionAddress,tmpNextInstructionAddress,writeToReg,regW
    wire branchpc;
    SingleBranch branchornot(branch,psw,branchpc);
    MUX2To1_32bit
        branchMux(tmpNextInstructionAddress,branchAddress,branchpc,tmpNextBranchInstructionAddress);
    MUX3To1_32bit
        jumpMux(tmpNextBranchInstructionAddress,jumpAddress,regRdData1,jump,nextInstructionAddress);

endmodule
```

多周期 CPU 数据通路 对于多周期数据通路的设计，其主要的 Memory，ALU 元件只有一个，不同的指令阶段要对这些元件进行复用，所以我们没有采用和单周期 CPU 相同的方式编写代码，而是根据不同功能放置这些模块。图片 3展示了具体的数据通路。具体代码如下：

```
///*** since multiple cycle cpu does not have redundant modules, the code style will not
    be similar as single cycle style which seperates into small stages. Instead, the code
    will be seperated according to different functions.***///

module MultipleCycleCPU(
    input wire clk
);

    ///*** wires ***///

    //control wires
    wire IorD;
    wire irWrite;
    wire pcWrite;
    wire pcWriteCond;
    wire [1:0] regDst;
    wire regWrite;
    wire [1:0] aluSrcA;
    wire [1:0] pcSrc;
    wire [2:0] aluSrcB;
```

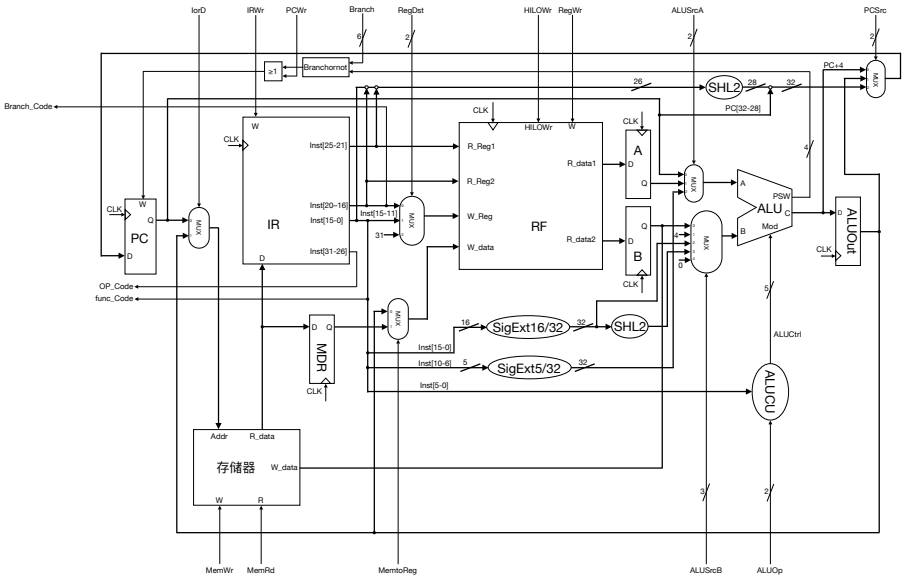

图 3: 多周期数据通路

```
wire memToReg;
wire [1:0] aluOP;
wire memRead;
wire memWrite;
wire [3:0] nextState;
wire [3:0] currentState;
wire [5:0] branch;
wire branchpc;

wire [4:0] aluCtrl;
//data wires
wire [31:0] nextInstructionAddress;
wire [31:0] currentInstructionAddress;
wire [31:0] memAccessAddress;

wire [31:0] aluTmpResult;
wire [31:0] aluResult;

wire [31:0] memTmpReadData;
wire [31:0] memReadData;

wire [31:0] tmpRegRdDataA;
wire [31:0] regRdDataA;
wire [31:0] tmpRegRdDataB;
wire [31:0] regRdDataB;
```

```
wire [31:0] instruction;

wire [3:0] psw;
wire [31:0] aluSrcDataA;
wire [31:0] aluSrcDataB;

wire [31:0] regWrData;
wire [4:0] regWrDst;
wire hilowrite;

wire [31:0] extendedImmediate;
wire [31:0] extendedImmediate2;

///*** cu ***///
MCUMutipleCycle
    mcu(instruction[31:26],instruction[20:16],instruction[5:0],currentState,clk,IorD,irWrite,pcWrite,branch
ALUCU aluCU(instruction[5:0],aluOP,aluCtrl);

///*** memory ***///
DataMemory dmem(memAccessAddress, regRdDataB, memWrite, memRead, memTmpReadData);

///*** alu ***///
wire [31:0] aluAddtest_tmp_useless;
wire [31:0] highresult;
wire [31:0] lowresult;
ALU
    alu(aluSrcDataA,aluSrcDataB,aluCtrl,aluTmpResult,highresult,lowresult,psw,aluAddtest_tmp_useless);

///*** register files ***///
RegisterFile
    regFile(instruction[25:21],instruction[20:16],regWrDst,regWrData,highresult,lowresult,regWrite,hilowrit

///*** extension ***///
SigExt16To32bit sigExt16To32(instruction[15:0],extendedImmediate);
SigExt5To32bit sigExt5To32(instruction[10:6],extendedImmediate2);

///*** register ***///
MultiBranch branchornot(branch, psw, branchpc);
PC pc(nextInstructionAddress, (pcWrite || branchpc), clk, currentInstructionAddress);
StateRegister stateRegister(nextState, clk, currentState);
InstructionRegister ir(memTmpReadData, irWrite, clk, instruction);
TemporaryRegister_32bit mdr(memTmpReadData, clk, memReadData);
TemporaryRegister_32bit aluOut(aluTmpResult, clk, aluResult);
TemporaryRegister_32bit rdDataARegister(tmpRegRdDataA, clk, regRdDataA);
TemporaryRegister_32bit rdDataBRegister(tmpRegRdDataB, clk, regRdDataB);

///*** mux ***///
MUX2To1_32bit memAddrMux(currentInstructionAddress, aluResult, IorD, memAccessAddress);
MUX2To1_32bit regWrDataMux(aluResult, memReadData, memToReg, regWrData);
MUX3To1_5bit regDstMux(instruction[20:16],instruction[15:11],5'b11111,regDst,regWrDst);
```

```
    MUX3To1_32bit
        aluSrcAMux(currentInstructionAddress,regRdDataA,extendedImmediate2,aluSrcA,aluSrcDataA);
    MUX5To1_32bit aluSrcBMux(regRdDataB,32'h00000004, extendedImmediate,
        (extendedImmediate<<2), 32'h00000000, aluSrcB, aluSrcDataB);
    MUX4To1_32bit pcSrcMux(aluTmpResult, aluResult,
        {currentInstructionAddress[31:28],instruction[25:0],2'b00}, regRdDataA, pcSrc,
        nextInstructionAddress);

endmodule
```

3 CPU 测试

3.1 TestBench 编写

tb_SigelCycleCPU.v 在单周期的 CPU 测试文件中，我们首先设置了时钟周期，每 100ns 反转一次，一个时钟周期 200ns，起初的时钟周期为 0，等待第一个上升沿到来
在寄存器文件和内存的初始化问题上，我们采用了从文件读入的预设置方法，在单周期的数据通路，同时存在指令存储器和数据存储器，均需要初始化。

```
module SingleCycleCPU_tb;
    reg clk;
    // Instantiate the singleCycleCPU module
    SingleCycleCPU cpu_inst (
        .clk(clk)
    );

    // Clock generation
    always #100 clk = ~clk;


    // Test stimulus
    initial begin
        // Initialize inputs

        /*
            AC030040 sw $3, 64($0)
            8C240002 lw $4, 2($1)
            00a63820 add $7, $5, $6
            1509FFFF bne $8, $9, -1
            1109FFFF beq $8, $9, -1
            08000002 j 2
        */

        clk <= 1'b0;

        $readmemh("./TestData/lw_RF.txt",cpu_inst.regFile.registers);
        $readmemh("./TestData/lw_Mem.txt",cpu_inst.imem.memory);
```

```
        $readmemh("./TestData/lw_Mem.txt",cpu_inst.dmem.memory);

        #6000
        $finish;
    end
endmodule
```

tb_multiplecyclecpu.v 在多周期 CPU 中，也采用预设置的方法，从外部文件读入内存和寄存器文件。同样地，设置时钟周期为 200ns，时钟初始化为 0。

```
module MultipleCycleCPU_tb;
    reg clk;

    MultipleCycleCPU cpu_inst (
        .clk(clk)
    );

    always #100 clk = ~clk;

    /***
        in memory, we assume 0~39bytes are instructions, 40~79bytes are data
        AC030040 sw $3, 64($1)
        8C240002 lw $4, 2($1)
        00a63820 add $7, $5, $6
        1109FFFF beq $8, $9, -1
        08000002 j 2
    ***/

    initial begin
        clk <= 1'b0;

        $readmemh("./TestData/lw_RF.txt",cpu_inst.regFile.registers);
        $readmemh("./TestData/lw_Mem.txt",cpu_inst.dmem.memory);
        #10000
        $finish;
    end

endmodule
```

3.2 测试用代码

这是寄存器的内容，从上到下，每一行表示一个 32 位数，第一行是 0 号寄存器的值，以此类推，我们在设计时添加了某些寄存器的值用来测试，但是寄存器的值个数不能超过 32 个。

```
00000000
```

```
FFFFFFFF
00000002
AAAAAAAA
00000000
00111011
00F000FF
DC01001E
F0000000
F0000001
FFFFFFF7
00000005
FFFFFFFC
00000000
00000000
00000008
```

这是内存的初始化文件，里面装入我们的指令格式的一些二进制代码，每条指令同样是 32 位，用 8 位 16 进制数表示，以下是某几条测试用的指令：

```
04410004
0C000016
AC030040
00410004
00010080
0022001A
0022001B
00220018
00220019
00A63826
00A63827
8C240002
00a63820
1109FFFF
08000002
0022001a
0c000000
00000008
00010082
00410007
00a6382a
1509FFFF
08000002
E0FD0132
01E00008
```

我们对第 0、2、6 条指令进行解释：
第 0 条指令：
16 进制：04410004

2 进制：000001 00010 00001 00000 00000 000100
翻译过来是：bgtz (2), 4;
第 2 条指令：
16 进制：AC030040
2 进制：101011 00000 00011 00000 00001 000000
翻译过来是：sw (3), 64(0);
第 6 条指令：
16 进制：0022001B
2 进制：000000 00001 00010 00000 00000 011011
翻译过来是：div (1), (2);

4 实验结果

4.1 添加指令集

我们添加了某些指令，这些指令可以被大致分为：
R 型指令：
1	add
2	sub
3.4	srl(v)
5.6	sra(v)
7.8	sll(v)
9	and
10	or
11	xor
12	sltu
13	mult
14	div
15	slt
16	addu
17	divu
18	multu
19	nor
20	subu
有条件跳转指令（B 型指令）：
21	beq
22	bne
23	blez
24	bgtz
25	bltz
26	bgez
无条件跳转指令（J 型指令）：

27	j
28	jal
29	jr
存数指令（S 型指令）：
30	sw
取数指令（L 型指令）：
31	lw
其中，我们的所有指令严格按照了 MIPS 的指令格式。可以在 MIPS 指令手册中查到所有使用的指令格式。
在基本的框架和 10 条指令集的基础上，我们添加了自己的指令，有些指令需要添加寄存器（如乘法除法的四条指令，均需将结果传入到 HI、LO 两个寄存器中）；有些指令需要在多路选择器上添加立即数，如 jal 指令，为了实现调用另一函数前保存断点的目的，将当前状态下的下一条指令地址 PC+4 放在 31 号寄存器中；有些指令需要添加新的部件，如 B 型指令中的 Branch 信号，可以据此判断不同的 Branch 指令类型，由 MCU 发出，在原来的框架里，是需要将 Z 标志位与 PCWrCond 相与，现在改为更加复杂的逻辑判断，将所有的与或门加入到 Branchornot 的逻辑判断模块中，接收 Branch 信号和 PSW 状态字，以此来判断是否需要跳转。
下面，我们简要列出所有指令的实现过程：

R 型指令 R 型指令的实现过程大致相同，主要有几种截然不同的指令格式：
1. 乘法除法指令格式：
special rs rt 0.A.0 func（0.A.0 表示有 10 个 0，16 进制表示）
需要将 rs、rt 寄存器中的值传入 ALU，ALU 计算完毕后，将结果传入 HI、LO 寄存器中。
我们在 CU 中加入了一个新的信号 HILOWr，用于控制将 ALU 的输出在下一时钟周期上升沿写入到 HI、LO 寄存器中。
2. 不可变移位指令格式：
special 0.5.0 rt rd s func
需要将 s 通过一个 5 到 32 位的符号扩展器扩展为 32 位，然后将 rt 寄存器中的值传入 ALU，ALU 计算完毕后，将结果传入 rd 寄存器中。
我们加入了一个新的符号扩展器，用于将 5 位的 s 扩展为 32 位，在 ALU 的左操作数 ALUSrcA 的多路选择器上加入被扩展后的数字，并将多路选择器的控制位数适当的增加。
3. 可变移位指令格式、其余所有 R 型指令格式：
special rs rt rd 0.5.0 func
需要将 rs、rt 寄存器中的值传入 ALU，ALU 计算完毕后，将结果传入 rd 寄存器中。

B 型指令 B 型指令的实现过程中，我查阅了 CSAPP 的内容，根据 ALU 返回的结果逐一判断是否符合现有的跳转条件，该指令分为两种格式：
1. 双寄存器比较指令格式：
opcode rs rt offset
需要将 rs、rt 寄存器中的值传入 ALU，ALU 计算完毕后，将结果传入到 Branchornot 模块中，由 Branchornot 模块判断是否需要跳转。
2. 单寄存器比较（大于 0、小于等于 0）指令格式：
opcode rs 0.5.0 offset

在 ALU 的右操作数 ALUSrcB 的多路选择器上加入 0，适当扩展多路选择器的控制位数，将 ALU 的输出传入到 Branchornot 模块中，由 Branchornot 模块判断是否需要跳转。
3. 单寄存器比较（大于等于 0、小于 0）指令格式：
opcode rs bCode offset
该指令格式中，两条指令的 opcode 相同，具体的指令类型需要从 bCode 中判断，bCode 的值为 00000 或 00001，00001 表示大于等于 0，00000 表示小于 0。
将 bCode 传入到 MCU 模块中，用于具体判断 B 指令类型，其余的做法与单寄存器比较（大于 0、小于等于 0）指令格式较为一致。

J 型指令 J 型指令的实现过程较为简单，仅有两种格式：
1. jal 指令格式：
opcode *instr_index*
该指令与 j 指令有以下区别：需要将 PC+4 的值传入到 31 号寄存器中，用于保存断点。单周期的 CPU 添加该指令较简单，只需要添加这条指令的类型，设置寄存器写信号和写的目标——31 寄存器号；在多周期 CPU 中，我们添加了当前状态，为 jal 添加一族额外的状态，用于在不同的时钟周期执行发出相应的控制信号，即设置寄存器写信号和写的目标。
2. jr 指令格式：
special rs 0.15.0 func
严格意义上，这是一个 R 型指令，需要将 rs 寄存器中的值传入到 PC 中，用于跳转返回。
我们将 R_data1 的输出线引到 Jump 信号控制的多路选择器中，在该指令的周期中，将要写的 PC 值从 RF 中读出来，在多路选择器的选择下，在下一时钟周期上升沿写入到 PC 中。

4.2 缺陷分析

在本次实验中，我们设计的 CPU 依然存在一些缺陷。

第一个执行周期的问题 在验收代码时，我们和老师讨论过这个问题。即我们的 CPU 在执行代码时，由于 clk 初始为 0，此时寄存器和内存中数据已经初始化完成，故直接执行第一条指令。但这就导致第一个指令实际上只用了半个周期的时间就执行完成。后续，我们了解到需要在增加 Reset 用于寄存器的初始化可以解决这个问题。

CU 设计 在多周期的 CU 设计时，由于各种 Branch 指令需要额外使用多种状态才能跳转，但是我们只为 Branch 指令设计了 state8，所以各种不同 branch 所用到不同的微命令只能在这个状态中判断，这也就导致了在 PLAd 的上半部分需要使用一定的或逻辑才能做到，这破坏了 PLA 原本的结构。

5 写在最后

本实验的全部代码，测试用数据，实验报告均已经上传 **Github**[Zhang and Ni, 2024]。所有指令均通过测试，可以自行下载，根据测试数据的指引进行测试验证。

参考文献

Matt GodBolt. Compiler explorer, 2024. URL https://godbolt.org.

Huanzhao Wang and Kewang Zhang. 计算机组成与设计. 清华大学出版社, 2021.

Hangyu Zhang and Luyang Ni. Verilog cpu design, 2024. URL https://github.com/Otaku-hy/XJTU_VerilogCPUDesign.