计算机系统结构实验报告：实验五

bugenzhao@sjtu.edu.cn

July 2, 2020

目 录

1 实验概述

实验名称 类 MIPS 单周期处理器的设计与实现

实验目的

1. 完成单周期的类 MIPS 处理器
2. 支持至少 16 条 MIPS 指令

实验成果

1. 实现了支持 **45 条指令**的单周期 MIPS 处理器
2. 重新设计了控制单元与信号，使用独立的控制模块，极大提高了代码的可读性
3. 配置了汇编器，允许直接将 MIPS 汇编转换为 Verilog `mem` 文件，并基于此编写了全面的功能测试，验证了 CPU 设计的正确性

2 实验设计

在本次实验中，我设计了一个支持 45 条指令的单周期 MIPS 处理器，如下表所示。其中灰色指令为编译器支持的伪指令，没有计入。

逻辑	AND	OR	XOR	NOR	ANDI	XORI	LUI	ORI	LI	
移位	SLL	SRL	SRA	SLLV	SRLV	SRAV	NOP			
算术	ADD	SUB	SLT	ADDI	SLTI	ADDU	SUBU	SLTU	ADDIU	SLTIU
跳转	JR	JALR	J	JAL						
分支	BEQ	B	BGTZ	BLEZ	BNE	BLTZ	BLTZAL	BGEZ	BGEZAL	BAL
内存	LB	LBU	LH	LHU	LW	SB	SH	SW		

相比于 *Computer Organization and Design* 中对于单周期处理器的电路设计，由于我的处理器支持更多的指令，我对部分模块做了一定的修改或重新设计。修改的部分主要有：

- 将 Control Unit 拆分成多个小模块，分别负责一部分控制信号。这样做方便对大量指令进行归纳处理，可提高代码设计的清晰度。
- 处理器支持的算术逻辑运算已经超过 16 个，需要使用 5 位字编码 ALU 控制信号。基于此，我对 ALU 控制信号做了重新设计，充分利用了 SPECIAL 指令原有的 `funct` 字段。
- 为支持立即数移位指令，我修改了 ALU 操作数的电路逻辑。
- 为支持寄存器跳转、寄存器分支指令，我对原电路中计算新 PC 的部分进行了重新设计，整合为一个新模块 `NewPC`。同时，我对 `WriteData` 和 `WriteReg` 模块进行了修改，以支持 Link 型的跳转和分支指令。

- 为支持 1、2、4 字节的内存读写，我重新设计了 DataMemory 和 MemControl 模块，增加了 memMode 信号控制读写字大小。

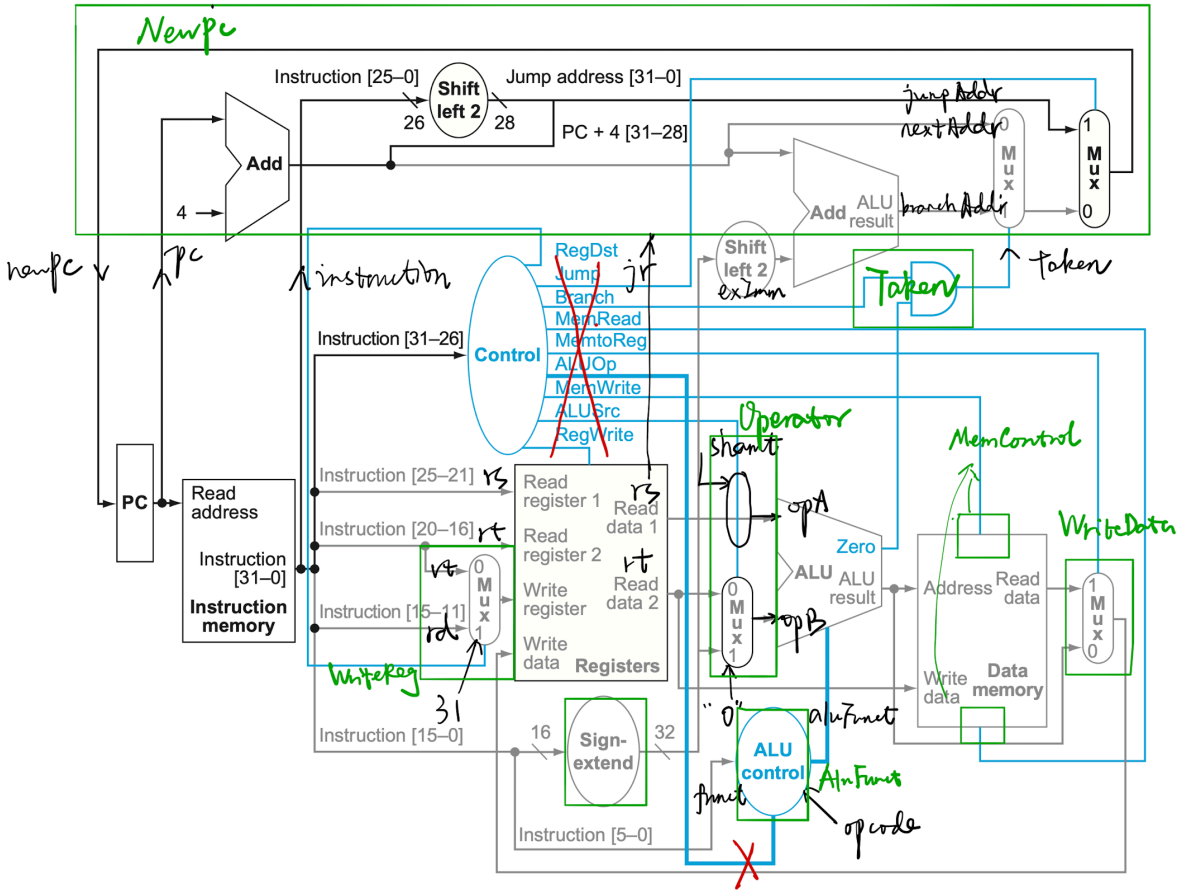

Figure 1: 单周期处理器的电路设计草图

总体而言，本次实验我的电路设计草图如图 1 所示，其中绿色框为 Control Unit 拆分得到的各个小控制单元，红色叉号代表不再需要这条控制线路。下面我将对部分重要的模块进行解读。

2.1 准备工作：ISA.v 文件

设计 由于需要实现的指令繁多，在开始本次实验前，我首先花半天时间调研了 MIPS I、MIPS 32 等多个指令集变体，挑选了较为常用的 45 条指令，对它们的指令编码进行了归纳总结。ISA.v 文件中包含指令 opcode、funct 与 REGIMM 类型指令的 rt 字段的宏，还定义了一些常用操作、常用总线宽度、内存访问模式信号和一些常用寄存器序号。

其中 opcode 字段为 6'b000000 的，统一归纳为 SPECIAL 指令，由 funct 字段进行区分；opcode 字段为 6'b000001 的分支指令，统一归纳为 REGIMM 指令，由 rt 字段进行区分。

实现 文件中的定义超过一百行，限于篇幅，这里只列举其中的一部分内容。之后，我们可以在 Vivado 里设置该文件为 Global Include，或者注明 `` `include "ISA.v" ``，即可在其他模块中引用这些常量。

```
// OPCODE ------------------------------------------

// R-type
`define OPC_SPECIAL  6'b000000 // ALU, JR, NOP(SLL)
// I-type
`define OPC_ADDI     6'b001000
`define OPC_ADDIU    6'b001001
`define OPC_ANDI     6'b001100
`define OPC_ORI      6'b001101
`define OPC_XORI     6'b001110

...
// FUNCT -------------------------------------------

// Arithmetic
`define FUN_ADD      6'b100000
`define FUN_ADDU     6'b100001
`define FUN_SUB      6'b100010
`define FUN_SUBU     6'b100011

...
// Length ------------------------------------------

`define OPC      5:0
`define REG      4:0
`define SHA      4:0
`define FUN      5:0
`define WORD    31:0
`define MMD      1:0

...
// Memory mode -------------------------------------

`define MEM_BYTE        2'b00
`define MEM_HALF        2'b01
`define MEM_WORD        2'b11
```

2.2 操作数选择：Operand 模块

设计 支持更多指令后，要充分利用好 ALU，首先要对 Operand 进行修改。

- 对于第一个操作数 opA，三个立即数移位指令需要用到 shamt 字段，其它指令一律使用 rs 寄存器读出的值。
- 对于第二个操作数 opB，要分多钟情况讨论：
  - 对于与 0 比较的分支指令（包括 REGIMM 类型指令），一律设置为 0。
  - 对于其它分支指令和 SPECIAL 指令，设置为 rt 寄存器读出的值。
  - 对于立即数运算指令，要特别注意 ISA 的规定，采用无符号扩展的立即数或有符号扩展的立即数。
  - 对于内存访问指令，一律设置为有符号扩展的立即数，最终在 ALU 中使用加法算出偏移后的地址。
  - 对于跳转指令，可以忽略。

实现 该模块的关键代码如下所示。

```
// opA:
always @(*) begin
    case (opcode)
        `OPC_SPECIAL: begin
            case (funct)
                `FUN_SLL, `FUN_SRL, `FUN_SRA:
                    opA = {{27{1'b0}}, shamt};
                default:
                    opA = rsData;
            endcase
        end
        default:
            opA = rsData;
    endcase
end

// opB:
always @(*) begin
    case (opcode)
        `OPC_REGIMM, `OPC_BGTZ, `OPC_BLEZ: // slt, sle
            opB = 0;
        `OPC_SPECIAL, `OPC_BEQ, `OPC_BNE: // alu
            opB = rtData;
        `OPC_ADDI, `OPC_ADDIU, `OPC_SLTI, `OPC_SLTIU: // imm
```

```
            opB = extendedImm;
        `OPC_ANDI, `OPC_ORI, `OPC_XORI, `OPC_LUI: // zero-ext imm
            opB = zeroExtendedImm;
        `OPC_LB, `OPC_LBU, `OPC_LH, `OPC_LHU, `OPC_LW, `OPC_SB, `OPC_SH,
    `OPC_SW: // l, s
            opB = extendedImm;
        default: // j, jal
            opB = 32'hxxxxxxxx;
    endcase
end
```

2.3 算术逻辑功能：ALUFunct 模块

设计 上文已经提到，在选择支持 45 条指令后，ALU 的功能已经超过 16 种。为方便今后进一步扩展，我将 ALU 功能控制信号扩展到 5 位，也因此很好地利用起了 SPECIAL 类型指令的 `funct` 字段。具体而言：

- 对于 SPECIAL 类型的指令，直接使用其 `funct` 字段作为 ALU 功能信号。其中的 JR 和 JALR 指令，可在 ALU 中做忽略处理。
- 对于基本的立即数运算指令，直接将它们映射到对应的 SPECIAL 类型中的寄存器运算指令的 `funct` 字段上。对于 LUI（加载立即数到高位）指令，单独设计一个新的信号 `FUN_LUI`。
- 对于分支指令，要分多钟情况讨论：
  - 对于寄存器间比较的分支指令，设置为 `FUN_SUB`。
  - 对于 REGIMM 类型的分支指令，注意到只存在 $\geq$ 和 $<$ 两种类型，可统一设置为 `FUN_SLT` 做小于比较，最终由 `Taken` 模块再做解析。
  - 对于其它分支指令，注意到只存在 $>$ 和 $\leq$ 两种类型，可统一设置为 `FUN_SLE` 做小于等于比较，最终由 `Taken` 模块再做解析。
- 对于内存访问指令，设置为加法以计算偏移后的地址。
- 对于跳转指令，可以忽略。

实现 该模块的关键代码如下所示。

```
always @(*) begin
    case (opcode)
        `OPC_SPECIAL:
            aluFunct = funct; // JR, JALR: NO
        `OPC_ADDI:
```

```
6               aluFunct = `FUN_ADD;
7           `OPC_ADDIU:
8               aluFunct = `FUN_ADDU;
9           `OPC_ANDI:
10              aluFunct = `FUN_AND;
11          `OPC_ORI:
12              aluFunct = `FUN_OR;
13          `OPC_XORI:
14              aluFunct = `FUN_XOR;
15          `OPC_LUI:
16              aluFunct = `FUN_LUI;
17          `OPC_SLTI:
18              aluFunct = `FUN_SLT;
19          `OPC_SLTIU:
20              aluFunct = `FUN_SLTU;
21          `OPC_REGIMM: // BGEZ, BGEZAL, BLTZ, BLTZAL, (BAL)
22              aluFunct = `FUN_SLT; // opB will be 0
23          `OPC_BGTZ, `OPC_BLEZ:
24              aluFunct = `FUN_SLE; // opB will be 0
25          `OPC_BEQ, `OPC_BNE:
26              aluFunct = `FUN_SUB;
27          `OPC_LB, `OPC_LBU, `OPC_LH, `OPC_LHU, `OPC_LW, `OPC_SB, `OPC_SH,
    `OPC_SW:
28              aluFunct = `FUN_ADD;
29          `OPC_J, `OPC_JAL:
30              aluFunct = `FUN_NO;
31
32          default: begin
33              $warning("%m: opcode not recognized: %06b", opcode);
34              aluFunct = `FUN_NO;
35          end
36      endcase
37  end
```

2.4 算术逻辑运算：ALU 模块

设计 经过前面两个模块的准备，我们可以根据两个操作数和 `aluFunct` 信号设计 ALU 进行运算，若计算结果为 0，则要将 `zero` 输出为高电平。MIPS ISA 规定 `ADD` 等不是以 `U` 为后缀的指令在运算溢出时应触发异常，因为我的设计中没有实现异常，因此这里选择当作普通的 `ADDU` 加法进行处理。

实现 该模块的关键代码如下所示。特别要注意的是，Verilog 的数据类型默认是无符号的，在进行有符号的比较、移位时，要先使用 `$signed` 进行转换。另外，注意区分逻辑移位和算术移位的运算符：`<<` 和 `<<<`。

```
always @(*) begin
    case (aluFunct)
        `FUN_ADD:
            out = opA + opB;
        `FUN_ADDU:
            out = opA + opB;
        `FUN_SUB:
            out = opA - opB;
        `FUN_SUBU:
            out = opA - opB;
        `FUN_SLT:
            out = $signed(opA) < $signed(opB) ? 1 : 0;
        `FUN_SLTU:
            out = opA < opB ? 1 : 0;
        `FUN_AND:
            out = opA & opB;
        `FUN_OR:
            out = opA | opB;
        `FUN_XOR:
            out = opA ^ opB;
        `FUN_NOR:
            out = ~(opA | opB);
        `FUN_SLL:
            out = opB << opA; // rt << sa(opA)
        `FUN_SLLV:
            out = opB << (opA[`SHA]); // rt << rs
        `FUN_SRL:
            out = opB >> opA;
        `FUN_SRLV:
            out = opB >> (opA[`SHA]);
        `FUN_SRA:
            out = $signed(opB) >>> opA;
        `FUN_SRAV:
            out = $signed(opB) >>> (opA[`SHA]);
        `FUN_LUI:
            out = opB << 16;
        `FUN_SLE:
            out = $signed(opA) <= $signed(opB) ? 1 : 0;
        `FUN_JR, `FUN_JALR, `FUN_NO:
```

```
            out = 32'hxxxxxxxx;

        default: begin
            if (aluFunct != 6'bxxxxxx) $warning("%m: aluFunct not recognized:
    %06b", aluFunct);
            out = 0;
        end
    endcase

    zero = out == 0 ? 1 : 0;
end
```

2.5 内存控制：MemControl 模块和 DataMemory 模块

设计 MIPS 的内存读写指令相当丰富，支持 Byte、Half、Word 三个大小，还有有符号、无符号的区分。为了让数据内存接口设计尽可能简单，我们在内存读写时暂不考虑符号的区别，而最终写入寄存器前，由 `WriteReg` 模块统一处理符号，进行扩展。因此，我设计了一个 2 位的 `mode` 信号，用于指示三种读写大小，其定义在 `ISA.v` 中已经涉及。

数据内存已改写为字节为寻址单元，在接收到内存读写信号和模式信号后，要跟据不同大小输出不同内容。这里我遵循了 MIPS 中流行的大端格式，即地址的低位存储数据的高位。

实现 `MemControl` 模块的关键代码如下所示，比较直观。

```
always @(*) begin
    memRead  = 0;
    memWrite = 0;
    case (opcode)
        `OPC_LB, `OPC_LBU: begin
            memRead = 1;
            mode    = `MEM_BYTE;
        end
        `OPC_LH, `OPC_LHU: begin
            memRead = 1;
            mode    = `MEM_HALF;
        end
        `OPC_LW: begin
            memRead = 1;
            mode    = `MEM_WORD;
        end
        `OPC_SB: begin
```

```
            memWrite = 1;
            mode     = `MEM_BYTE;
        end
        `OPC_SH: begin
            memWrite = 1;
            mode     = `MEM_HALF;
        end
        `OPC_SW: begin
            memWrite = 1;
            mode     = `MEM_WORD;
        end
    endcase
end
```

DataMemory 模块的关键代码如下所示。读写时都要特别注意大端格式，避免出错。这里可以运用 Verilog 的拼接语法简化代码。

```
always @(negedge clk) begin
    if (memWrite) begin
        case (mode)
            `MEM_BYTE:
                memFile[address + 0] = writeData[ 7: 0];
            `MEM_HALF: begin
                memFile[address + 0] = writeData[15: 8];
                memFile[address + 1] = writeData[ 7: 0];
            end
            `MEM_WORD: begin
                memFile[address + 0] = writeData[31:24];
                memFile[address + 1] = writeData[23:16];
                memFile[address + 2] = writeData[15: 8];
                memFile[address + 3] = writeData[ 7: 0];
            end
        endcase
    end
end

always @(address, mode, memRead) begin
    if (memRead) begin
        case (mode)
            `MEM_BYTE:
                readData = {{24{1'b0}}, memFile[address]};
            `MEM_HALF:
                readData = {{16{1'b0}}, memFile[address], memFile[address +
```

```
    1]};
27              `MEM_WORD:
28                  readData = {memFile[address], memFile[address + 1], memFile[
    address + 2], memFile[address + 3]};
29          endcase
30      end
31      else begin
32          readData = 32'hxxxxxxxx;
33      end
34  end
```

2.6 写回：WriteData 模块和 WriteReg 模块

设计 WriteData 模块负责根据指令类型决定写回寄存器的数据，大体而言就是从 ALU 运算结果和数据内存读取结果中进行选择。由于我们设计了不同大小、不同符号的内存操作指令，这里需要根据指令类型，对不足 32 位字的读取指令做符号扩展或零扩展。另外，由于实现了 Link 指令，其行为类似 x86 中的 CALL，需要将下一个 PC 写回寄存器中，我们还需特别处理这种情况。

WriteReg 模块决定数据写回的目标寄存器。在原先的设计中，我们只需要从 `rt` 和 `rd` 中选择一个即可。在实现了 Link 后，要注意它的写回设计：部分指令固定写入到 31 号寄存器 `$ra` 中，另一部分写入到 `rt` 中。

上次实验在实现寄存器时，我提到了可以“把 0 号寄存器当作‘无寄存器’”的寄存器设计，这里我延续了该设计。即当不需要写入寄存器时，可直接将目标设置为 0，从而不必另外维护 `regWrite` 信号。

实现 WriteData 模块的关键代码如下所示。

```
1  always @(*) begin
2      case (opcode)
3          `OPC_LB:
4              writeData = {{24{memoryOut[7]}}, memoryOut[7:0]};
5          `OPC_LBU:
6              writeData = {{24{1'b0}}, memoryOut[7:0]};
7          `OPC_LH:
8              writeData = {{16{memoryOut[15]}}, memoryOut[15:0]};
9          `OPC_LHU:
10             writeData = {{16{1'b0}}, memoryOut[15:0]};
11         `OPC_LW:
12             writeData = memoryOut;
13         `OPC_REGIMM: begin
14             case (rt)
15                 `RT_BGEZAL, `RT_BLTZAL: writeData = pc + 4; // link
```

```
                default: writeData = aluOut; // actually no data
            endcase
        end
        `OPC_JAL: // link
            writeData = pc + 4;
        `OPC_SPECIAL: begin
            case (funct)
                `FUN_JALR: writeData = pc + 4; // link
                default:   writeData = aluOut;
            endcase
        end
        default:
            writeData = aluOut;
    endcase
end
```

WriteReg 模块的关键代码如下所示。

```
always @(*) begin
    case (opcode)
        `OPC_SPECIAL: begin
            case (funct)
                `FUN_JR:   writeReg = 0;
                `FUN_JALR: writeReg = rd; // link, 31 implied
                default:   writeReg = rd;
            endcase
        end
        `OPC_ADDI, `OPC_ADDIU, `OPC_ANDI, `OPC_ORI, `OPC_XORI, `OPC_LUI,
    `OPC_SLTI, `OPC_SLTIU:
            writeReg = rt;
        `OPC_REGIMM: begin
            case (rt)
                `RT_BGEZAL, `RT_BLTZAL: writeReg = 31; // link
                default: writeReg = 0;
            endcase
        end
        `OPC_BGTZ, `OPC_BLEZ, `OPC_BEQ, `OPC_BNE:
            writeReg = 0;
        `OPC_LB, `OPC_LBU, `OPC_LH, `OPC_LHU, `OPC_LW:
            writeReg = rt;
        `OPC_SB, `OPC_SH, `OPC_SW:
            writeReg = 0;
        `OPC_J:
```

```
            writeReg = 0;
        `OPC_JAL: // link
            writeReg = 31;

        default: begin
            $warning("%m: opcode not recognized: %06b", opcode);
            writeReg = 5'bxxxxx;
        end
    endcase
end
```

2.7 下一条指令：Taken 模块和 NewPC 模块

设计 扩展了原先的指令集后，现在新 PC 的来源共有 4 种，它们分别是：

- nextAddr：紧接着的下一条指令，即 PC + 4
- branchAddr：分支目标
- jumpAddr：J 和 JAL 指令的跳转目标
- jumpRegAddr：JR 和 JALR 指令的跳转目标，从寄存器读出、经过 ALU 做 +0 计算后得到。

选择哪一个来源既取决于指令类型，也取决于分支指令是否 taken，这一部分的实现在 Taken 模块中。回忆 ALU 功能设计中，我们把不同的分支指令都统一用 FUN_SLT 和 FUN_SLE 归纳，这里需要根据 ALU 输出的 zero 信号，再次联合分支指令类型做进一步判断：对于大于和大于等于的分支指令，可能要进行取反操作。

实现 NewPC 模块的关键代码如下所示，这里的 taken 信号是 Taken 模块判断后输出的。

```
wire [`WORD] nextAddr   = pc + 4;
wire [`WORD] branchAddr = (extendedImm << 2) + nextAddr;
wire [`WORD] jumpAddr   = {nextAddr[31:28], (jumpIndex << 2)};

always @(*) begin
    case (opcode)
        `OPC_REGIMM, `OPC_BGTZ, `OPC_BLEZ, `OPC_BEQ, `OPC_BNE:
            newPC = taken ? branchAddr : nextAddr;
        `OPC_J, `OPC_JAL:
            newPC = jumpAddr;
        `OPC_SPECIAL: begin
            case (funct)
```

```
                `FUN_JR, `FUN_JALR:
                    newPC = jumpRegAddr;
                default:
                    newPC = nextAddr;
            endcase
        end
        default:
            newPC = nextAddr;
    endcase
end
```

Taken 模块的关键代码如下所示。

```
wire set = ~aluZero; // SLT, SLE

always @(*) begin
    case (opcode)
        `OPC_REGIMM: begin
            case (rt)
                `RT_BLTZ, `RT_BLTZAL:
                    taken = set;
                `RT_BGEZ, `RT_BGEZAL:
                    taken = ~set;
                default: begin
                    $warning("%m: rt not recognized: %05b", rt);
                    taken = 0;
                end
            endcase
        end
        `OPC_BGTZ:
            taken = ~set;
        `OPC_BLEZ:
            taken = set;
        `OPC_BEQ:
            taken = aluZero;
        `OPC_BNE:
            taken = ~aluZero;
        default:
            taken = 'bx;
    endcase
end
```

2.8 其它模块简述

2.8.1 RegisterFile 模块

寄存器文件模块继承了上一次实验中的设计，在写入时“一视同仁”，而在读取时保证 0 号永远输出 0。这样做的另一个好处是，我们可以把 0 号寄存器当作“无寄存器”，CPU 其他模块决定不写入寄存器时，可以直接将目的地址设置为 0，而不必额外维护 regWrite 信号。

注 在这种设计下，波形中显示的 regFile[0] 的值没有参考价值，读取逻辑保证 0 号寄存器的值永远为 0。

2.8.2 InstMemory 模块

指令内存模块运行相当简单，因为每条指令均为 4 字节，所以可以直接将其寻址单元设置为 4 字节，使用 PC 值右移两位后的结果进行寻址。

2.8.3 SignExt 模块

符号扩展模块也直接继承了上一次实验中的设计。

2.9 最终整合：CPU 模块和 System 模块

考虑一个真实的计算机系统，它的 CPU 内部并不包含指令内存和数据内存，而是通过 CPU 引出控制总线、地址总线、数据总线的方式与主存储器进行交流。因此，我首先构建了一个 CPU 模块，其中不带任何内存装置，而通过如下输入输出信号与主存交流。这样做更加符合真实计算机结构，同时也使得在 CPU 和主存之间加入 Cache 变得比较方便——不需要修改已有的 CPU 内部设计。

```
module CPU(
           input clk,
           output wire [`WORD] pc,
           input  wire [`WORD] inst,
           output wire [`WORD] dataAddress, writeMemData,
           output wire         memRead, memWrite,
           output wire [ `MMD] memMode,
           input  wire [`WORD] readMemData
       );
```

由于我们的设计严格遵循模块化的要求，整个 CPU 模块中只出现了三种语句，即 wire 声明语句、assign 赋值语句、模块实例化语句，而不包含任何逻辑代码。这使得模块设计极其清晰、简洁。由于篇幅所限，这部分代码不再列出，请参见 CPU.v。

构建好了 CPU 模块后，我们可以最终将主存储器和 CPU 整合起来，构建顶层 System 模块。同样地，System 模块的代码十分简洁，它不接受输入、输出，但接受两

个参数，指定指令内存的转储文件路径和时钟频率。

```
module System #(parameter textDump = "path/to/text/dump",
                parameter PERIOD   = 10);
// --- Clock ---
reg clk;
always #(PERIOD) clk = !clk;
initial clk = 1;

// --- Memory ---
wire [`WORD] pc;
wire [`WORD] instruction;
InstMemory #(textDump) u_InstMemory(
    .pc          (pc          ),
    .instruction (instruction )
);

wire [`WORD] dataAddress;
wire [`WORD] writeData, readData;
wire [ `MMD] memMode;
wire         memRead, memWrite;
DataMemory u_DataMemory(
    .clk       (clk         ),
    .address   (dataAddress ),
    .writeData (writeData   ),
    .mode      (memMode     ),
    .memRead   (memRead     ),
    .memWrite  (memWrite    ),
    .readData  (readData    )
);


// --- MIPS CPU ---
CPU u_CPU(
    .clk          (clk          ),
    .pc           (pc           ),
    .inst         (instruction  ),
    .dataAddress  (dataAddress  ),
    .writeMemData (writeData    ),
    .memRead      (memRead      ),
    .memWrite     (memWrite     ),
    .memMode      (memMode      ),
    .readMemData  (readData     )
```

```
42 );
43
44 endmodule // System
```

3 仿真测试

3.1 准备工作：汇编器

实验手册中建议我们编写简单的 MIPS 汇编程序来进行测试，然而手动将汇编语言翻译为机器码费时费力，且极其容易出现错误。因此，我在测试工作开展之前，先配置了 MIPS 目标的 gcc 交叉编译器和 binutils 二进制工具，其中就包含 MIPS 汇编器。

由于我这项工作在 macOS 上开展，因此我自行配置了一个 Homebrew 软件包，可自动从 GNU 网站获得 gcc 和 binutils 源码，设置 mips32-elf 目标后进行编译安装。这部分的内容和使用方法我已公开在 GitHub 仓库上：https://github.com/BugenZhao/homebrew-mips32。配置、安装完成后，在终端输入 `mips32-elf-as` 即可调用 MIPS 汇编器，如下图所示。

```
❯ mips32-elf-as --version
GNU assembler (GNU Binutils) 2.34
Copyright (C) 2020 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or later.
This program has absolutely no warranty.
This assembler was configured for a target of `mips32-elf'.
```

Figure 2: MIPS 汇编器

汇编器编译得到的结果是二进制文件，且包含一些无用的 section，因此需要进一步处理、转换成 Verilog 要求的格式。为此，我编写了一个 Makefile 文件（位于 `Resources/Makefile`）来自动化这一过程，其大致步骤为：

1. 汇编 MIPS 代码。
2. 将汇编结果再次反汇编得到 `lst` 文件，其中详细展示了每条指令和它的地址、二进制码。
3. 提取二进制中的 `text` 段，即程序指令段。
4. 反转上一步结果的字节顺序，适应大端系统。
5. 使用 UNIX 的 `od`、`tr`、`sed` 程序将上一步结果转换为 Verilog 格式。

```
1 AS      = mips32-elf-as
2 OBJDUMP = mips32-elf-objdump
3 OBJCOPY = mips32-elf-objcopy
```

```
ASFLAGS = -O0 -mips1 -EB
SRCS    = $(wildcard *.mips)
LSTS    = $(SRCS:.mips=.mem)
OUTDIR  = ./Products

all: $(LSTS)

%.mem: %.mips
    mkdir -p $(OUTDIR)
    $(AS) -o a.out $< $(ASFLAGS)
    $(OBJDUMP) -d a.out > $(OUTDIR)/$(@:.mem=.lst)
    $(OBJCOPY) --dump-section .text=a.bin a.out
    $(OBJCOPY) -I binary -O binary --reverse-bytes=4 a.bin r.bin
    od -An -v -t x4 r.bin | tr ' ' '\n' | sed '/^$$/d' > $(OUTDIR)/$@
    rm a.bin a.out r.bin
```

为了防止汇编器自动在某些指令冒险之间添加 NOP，我们需要在汇编程序的第一行注明.set noreorder，这样做可以为下次实验中测试流水线冒险的处理情况打下基础。

不妨进行一些简单的试验。将我自己编写内存指令测试程序 LoadStore.mips 进行操作，可得到 Resources/Products/下两个文件：

- LoadStore.lst 详细展示了每个指令和它的地址、二进制码，非常方便调试。

```
00000000 <main>:
   0:   3c09ffff    lui t1,0xffff
   4:   35298000    ori t1,t1,0x8000
   8:   ac090000    sw  t1,0(zero)
   c:   840a0002    lh  t2,2(zero)
  10:   940b0002    lhu t3,2(zero)
  14:   800c0000    lb  t4,0(zero)
  18:   900d0000    lbu t5,0(zero)
  1c:   24090090    li  t1,144
  20:   a0090003    sb  t1,3(zero)
  24:   8c0e0000    lw  t6,0(zero)
```

- LoadStore.mem 可直接导入 Verilog 进行测试。

```
3c09ffff
35298000
ac090000
840a0002
940b0002
800c0000
```

```
900d0000
24090090
a0090003
8c0e0000
```

基于此，我首先在 Example.mips 中编写了一系列各种类型的示例指令，将它们编译后获得了对应的二进制，方便在下面的模块仿真测试中直接使用真正的指令进行测试。这部分内容使用宏的方式定义在 Debug.v 中。

3.2 模块仿真测试

对于一些较为独立的模块，我编写了专门的测试激励文件，如 MemControl、WriteData 等。其中我都使用上一节提到的示例指令进行测试，并使用自己撰写的 `assert 宏与预期结果进行比较。运行时，可直接观察控制台输出，如无“ASSERTION FAILED”警告，则代表测试通过。

限于篇幅，本节只引用仿真测试的波形图，具体代码请您参考对应的 tb 文件。

3.2.1 ALU 仿真测试

仿真测试波形图如图 3 所示，运行过程中没有“ASSERTION FAILED”警告。

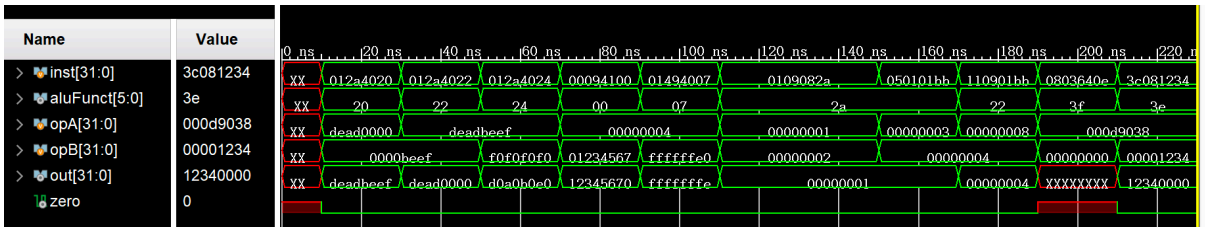

Figure 3: ALU 仿真测试

3.2.2 DataMemory 仿真测试

仿真测试波形图如图 4 所示，运行过程中没有“ASSERTION FAILED”警告。

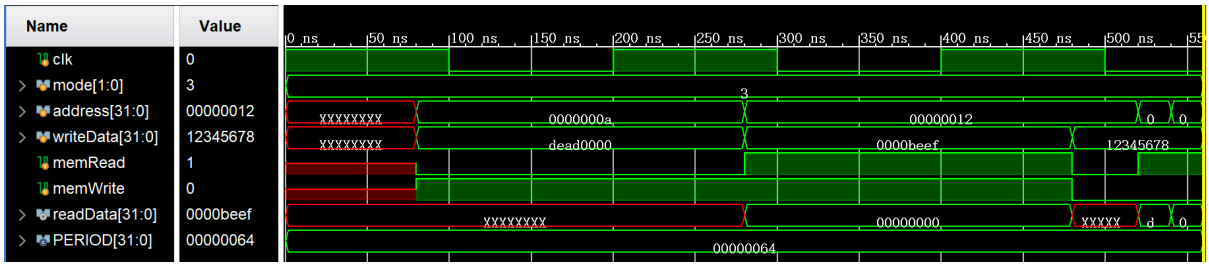

Figure 4: DataMemory 仿真测试

3.2.3 InstMemory 仿真测试

仿真测试波形图如图 5 所示，运行过程中没有“ASSERTION FAILED”警告。

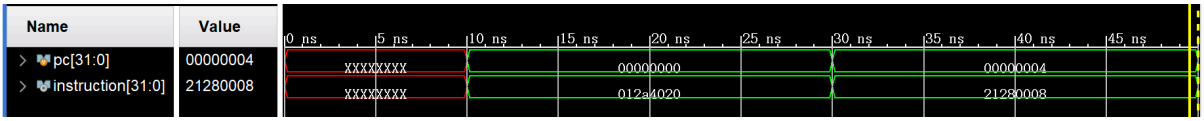

Figure 5: InstMemory 仿真测试

3.2.4 MemControl 仿真测试

仿真测试波形图如图 6 所示，运行过程中没有“ASSERTION FAILED”警告。

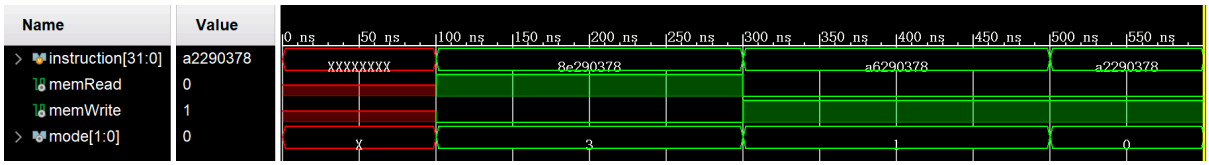

Figure 6: MemControl 仿真测试

3.2.5 RegisterFile 仿真测试

仿真测试波形图如图 7 所示，运行过程中没有“ASSERTION FAILED”警告。

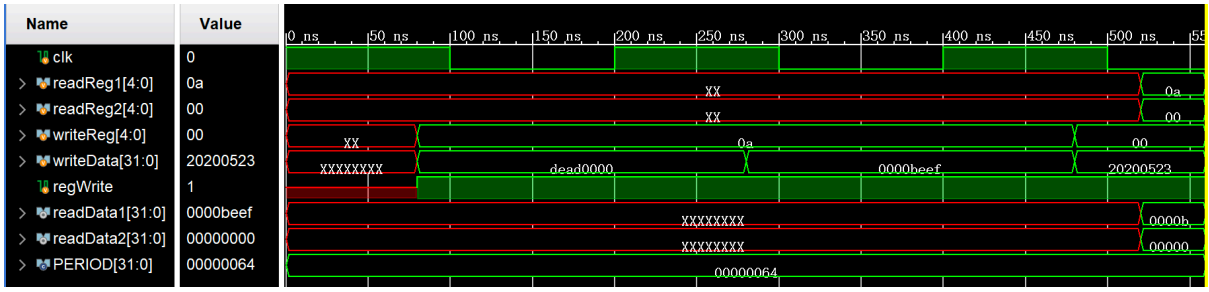

Figure 7: RegisterFile 仿真测试

3.2.6 WriteData 仿真测试

仿真测试波形图如图 8 所示，运行过程中没有“ASSERTION FAILED”警告。

3.2.7 WriteReg 仿真测试

仿真测试波形图如图 9 所示，运行过程中没有“ASSERTION FAILED”警告。

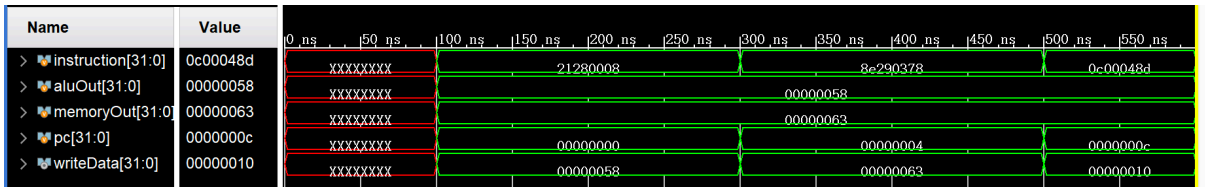

Figure 8: WriteData 仿真测试

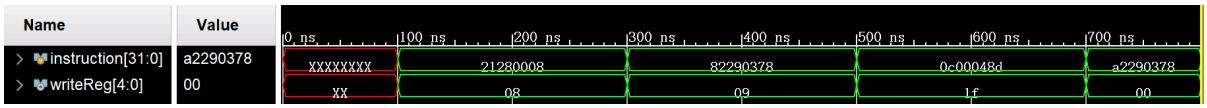

Figure 9: WriteReg 仿真测试

3.3 System 仿真测试

本实验中，我编写了四个 MIPS 小程序，汇编后导入了模拟器中进行仿真。以 S 开头的三个 tb 激励模块分别测试了算术逻辑运算指令、分支跳转指令、内存访问指令。System_tb.v 则加载了 Accumulation.mips 这一综合型的小程序，同时涉及算术、跳转、分支、内存指令，能够成功计算 $\sum_{i=1}^{n} i$ 的值。

3.3.1 算术逻辑运算仿真测试

汇编代码 位于 Resources/ArithLogic.mips，如下所示：

```
    .set noreorder

main:
    li      $t1, 0x88       # $t1 = 0x88
    addi    $t2, $t1, 0x11  # $t2 = $t1 + 0x11 = 0x99
    add     $t2, $t1, $t2   # $t2 = $t1 + $t2 = 0x121
    srl     $t2, $t2, 4     # $t2 = 0x12
    li      $t3, 0x2        # $t3 = 0x2
    sub     $t2, $t2, $t3   # $t2 = $t2 - $t3 = 0x10
    lui     $t7, 0x10       # $t7 = 0x00100000
    add     $t2, $t2, $t7   # $t2 = 0x00100010
    not     $t2, $t2        # $t2 = 0xffefffef

    andi    $t2, 0x00ff     # $t2 = 0xef
    li      $t3, 0x10       # $t3 = 0x10
    or      $t4, $t2, $t3   # $t4 = 0xff
    xor     $t5, $t4, $t2   # $t5 = 0x10

    li      $t6, 8          # $t6 = 8
    sllv    $t6, $t5, $t6   # $t6 = 0x1000
```

测试结果 寄存器波形图如图 10 所示，运行过程中没有“ASSERTION FAILED”警告，观察也可得知，符合程序运行预期结果。

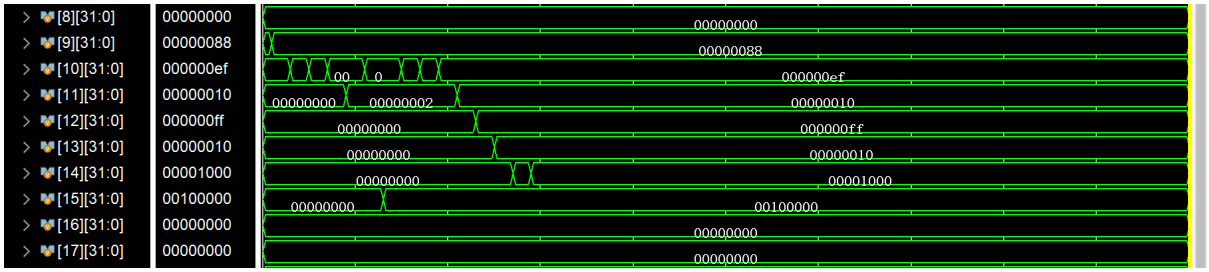

Figure 10: ArithLogic 仿真测试

3.3.2 分支跳转仿真测试

汇编代码 位于 Resources/BranchJump.mips，如下所示：

```
    .set noreorder
    j       main

inc_t5:
    addi    $t5, $t5, 1
    jr      $ra

main:
    move    $t0, $zero      # $t0 = $zero
    li      $t1, 0x88       # $t1 = 0x88
    li      $t2, 0x99       # $t2 = 0x99
    bge     $t2, $t1, l1    # if $t2 >= $t1 then l1
    addi    $t0, $t0, 1     # $t0 = $t0 + 1
l1:
    li      $t1, 0x88
    li      $t2, 0x88
    beq     $t2, $t1, l2
    addi    $t0, $t0, 1
l2:
    b       l3
    addi    $t0, $t0, 1
l3:
    bgt     $t1, $zero, l4
    addi    $t0, $t0, 1
l4:
    li      $t3, -8
    blez    $t3, l5
```

```
28      addi    $t0, $t0, 1
29  l5:
30      bne     $t1, $t3, l6
31      addi    $t0, $t0, 1
32  l6:
33      bltz    $t3, l7
34      addi    $t0, $t0, 1
35  l7:
36      bltzal  $t3, l8
37      addi    $t0, $t0, 1
38  l8:
39      bgez    $zero, l9
40      addi    $t0, $t0, 1
41  l9:
42      bgezal  $zero, l10
43      addi    $t0, $t0, 1
44  l10:
45      la      $t1, inc_t5
46      jalr    $t1
47      jal     inc_t5
48      j       l11
49      addi    $t0, $t0, 1
50  l11:
51      nop
```

测试结果 寄存器波形图如图 11 所示，运行过程中没有“ASSERTION FAILED”警告，观察也可得知，符合程序运行预期结果。

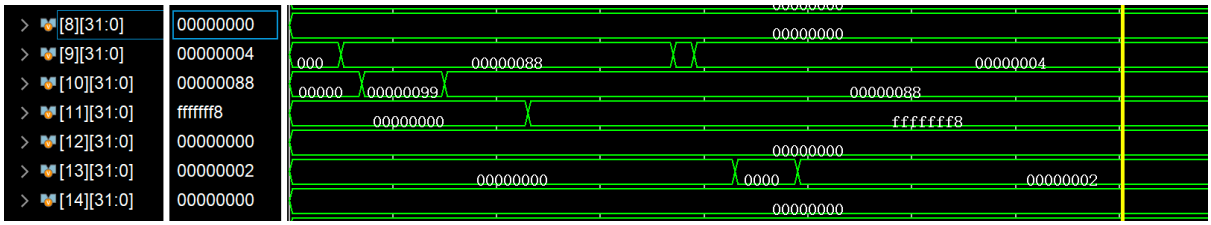

Figure 11: BranchJump 仿真测试

3.3.3 内存操作仿真测试

汇编代码 位于 Resources/LoadStore.mips，如下所示：

```
1       .set noreorder
2   main:
```

```
    lui     $t1, 0xffff
    ori     $t1, $t1, 0x8000
    sw      $t1, 0($zero)
    lh      $t2, 2($zero)       # 0xffff8000
    lhu     $t3, 2($zero)       # 0x00008000
    lb      $t4, 0($zero)       # 0xffffffff
    lbu     $t5, 0($zero)       # 0x000000ff
    li      $t1, 0x90
    sb      $t1, 3($zero)
    lw      $t6, 0($zero)       # 0xffff8090
```

测试结果 寄存器波形图如图 12 所示，运行过程中没有“ASSERTION FAILED”警告，观察也可得知，符合程序运行预期结果。

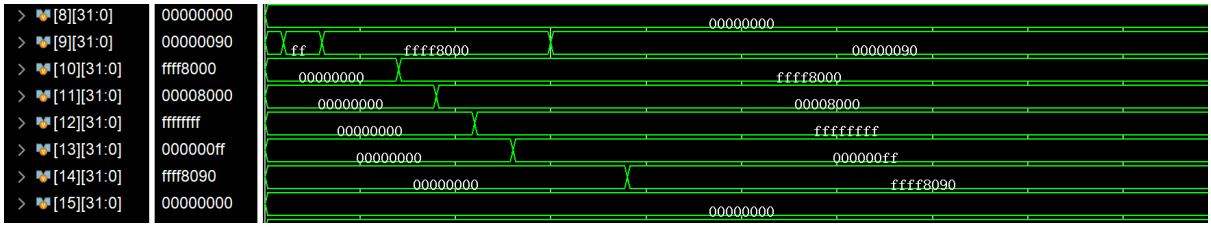

Figure 12: LoadStore 仿真测试

3.3.4 Accumulation 综合仿真测试

汇编代码 位于 Resources/Accumulation.mips，如下所示：

```
    .set noreorder
init:
    addi    $sp, $zero, 0x20    # $sp = 0x20
    j       main                # jump to main

worker:
    lw      $a0, 0($sp)         # pop 10
    addi    $sp, $sp, 4
    move    $v0, $zero          # $v0 = $zero
    move    $t0, $zero          # $t0 = $zero
    j       .test               # jump to .test
.loop:
    addu    $v0, $t0, $v0       # $v0 = $t0 + $v0
    addiu   $t0, $t0, 1         # $t0 = $t0 + 1
.test:
    bne     $t0, $a0, .loop     # if $t0 != $a0 then .loop
```

```
17      jr      $k0                 # jump to $k0
18
19
20  main:
21      li      $t0, 10             # $t0 = 10
22      addi    $sp, $sp, -4
23      sw      $t0, 0($sp)         # push 10
24      la      $t1, worker
25      jalr    $k0, $t1            # call worker
26      sw      $v0, 0($zero)
```

测试结果 激励模块为 System_tb.v。寄存器波形图如图 13 所示，运行过程中没有“ASSERTION FAILED”警告，观察也可得知，符合程序运行预期结果，成功得到结果 $\sum_{i=1}^{9} i = 45 = 2d_{16}$。

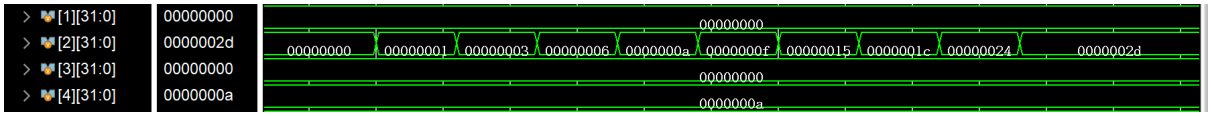

Figure 13: Accumulation 综合仿真测试

4 总结与感想