中国科学院大学
《计算机组成原理(研讨课)》实验报告

姓名 朱夏楠 学号 2022K8009929031 专业 计算机科学与技术

实验项目编号 2 实验名称 简单功能型处理器设计

注 1:	撰写此 Word 格式实验报告后以 PDF 格式保存 SERVE CloudIDE 的 `/home/serve-ide/cod-lab/reports` 目录下 (注意: reports 全部小写)。文件命名规则: `prjN.pdf`, 其中 `prj` 和后缀名 `pdf` 为小写, `N` 为 1 至 4 的阿拉伯数字。例如: `prj1.pdf`。PDF 文件大小应控制在 5MB 以内。此外, 实验项目 5 包含多个选做内容, 每个选做实验应提交各自的实验报告文件, 文件命名规则: `prj5-projectname.pdf`, 其中 "-" 为英文标点符号的短横线。文件命名举例: `prj5-dma.pdf`。具体要求详见实验项目 5 讲义。
注 2:	使用 `git add` 及 `git commit` 命令将实验报告 `PDF` 文件添加到本地仓库 master 分支, 并通过 `git push` 推送到实验课 SERVE GitLab 远程仓库 master 分支(具体命令详见实验报告)。
注 3:	实验报告模板下列条目仅供参考, 可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

一、 逻辑电路结构与仿真波形的截图及说明(比如 Verilog HDL 关键代码段 {包含注释} 及其对应的逻辑电路结构图 {自行画图, 推荐用 PPT 画逻辑结构框图后保存为 PDF, 再插入到 LaTeX. 中}、相应信号的仿真波形和信号变化的说明等)

• 单周期处理器设计。

本次单周期处理器设计实验使用的指令集为 MIPS, 字长为 32, 数据采用小端存储, 内存存取地址为 4 字节对齐。

将单周期处理器处理指令分为五个阶段, 取指、译码、执行、访存、写回。其中取指阶段为根据 PC 寄存器的值, 从内存中取出新的指令; 译码阶段为对取出的指令进行拆分, 并生成大量的控制信号, 同时根据指令中的寄存器编号读出寄存器中的值, 用于后续的三个阶段的数据选择、操作和写入; 执行阶段通过 ALU 和 Shifter 对上一个阶段得到的数据和指令本身所指示的立即数进行运算; 访存阶段根据之前阶段得到的地址从内存中获取数据或是将数据写入内存; 写回阶段根据指令的需要将数据写入寄存器堆中; 最后还需更新 PC 的值。单周期处理器实际上除了更新 PC 值以外所有过程都是在一个时钟周期内由组合逻辑完成的, 因此将指令分为五个阶段实际上的意义为便于理清思路, 从时序的角度讲模型本身并没有严格的体现在设计中。

本次实验设计中最关键的部分为控制信号的选择逻辑, 以及根据指令对控制信号进行赋值。由于单周期处理器全部操作都在一个时钟周期中完成, 设计中存在大量组合逻辑, 需要使用大量控制信号对输入的数据进行选择, 以此实现多条指令共用一套关键部件。可以说, 将控制信号赋值和选择逻辑完成后, 只需用硬件描述语言将相应的组合电路实现, 处理器的绝大部分就已经完成了。

因此, 在实际编写代码之前, 需要根据 MIPS 指令集手册查阅 45 条指令。可根据指令的操作码格式划分出四种类型的指令 R-Type、I-Type、REGIMM、J-Type, 其中 R-Type 又可以细分为运算 (Rc)、移位 (Rs)、跳转 (Rj) 和 mov(Rm) 四类指令, I-Type 又可以分为计算 (Ic)、分支跳转 (Ib)、内存读 (Il) 和内存写 (Is) 四类指令。同时可以据此初步对控制信号进行分类, 一类选择下一个时钟周期要更新的 PC 值; 一类选择是否向寄存器中存入数据以及存入的位置; 一类选择参与 ALU 和移位器运算的数据; 一类选择 ALU 和移位器进行的运算类型; 一类选择存入寄存器中的数据来源; 一类选择是否向内存中写入数据, 以及对存入内存和从内存中取出的数据进行处理。下面将结合指令对这六类控制信号进行详细的说明。

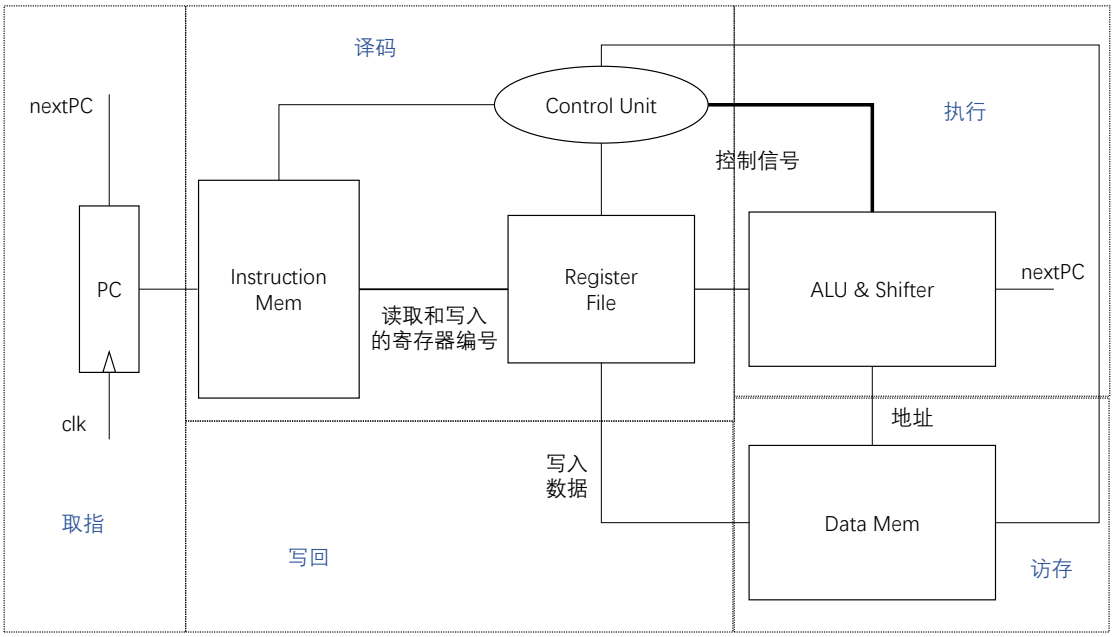

图 1: 单周期处理器简略示意图

对无跳转指令的单周期处理器来说，下一个时钟周期的指令永远是内存中排在当前指令的下一条指令，PC 值为当前 PC 值加 4，即 nextPC←PC+4。当加入跳转指令后，下一个时钟周期的 PC 值可能变为根据跳转指令计算出的值，因此需要加入控制信号来对下一个时钟周期的 PC 值进行选择。跳转指令有如下几种：Rj、Ib、J-Type、REGIMM，其中 Rj 和 J-Type 为直接跳转，Ib 和 REGIMM 为条件跳转，跳转类型为条件跳转时，需要先根据 ALU 运算的结果，决定是否需要跳转，若需要跳转，则将额外的 pc 加法器计算出的 PC 值当作 nextPC。

```
    //eq_Zero = ~(|rdata1)
assign BranchCond = (isbeq & Zero) |
                    (isbne & ~Zero) |
                    (isbgtz & ~eq_Zero & Zero) |
                    (isblez & (eq_Zero | ~Zero)) |
                    (isREGIMM & REG[0] & Zero) |
                    (isREGIMM & ~REG[0] & ~Zero);

assign nextpc = ({32{isRjump}} & rdata1)                        //将寄存器中的值作为PC值
           | ({32{isJtype}} & {pc_add_4[31:28],taddr,2'b0})
           | ({32{Branch & BranchCond}} & pc_branch)       //Branch = isREGIMM | isIb
           | ({32{~isRjump & ~isJtype & ~Branch}} & pc_add_4); //其他情况下, PC<=PC+4
```

寄存器堆根据 Instruction 中的 rs、rt 段选择读取的两个数据，需要控制信号 RegDst 决定写入的寄存器为 rt 还是 rd，还需要 RF_wen 信号决定是否向寄存器中写入数据。

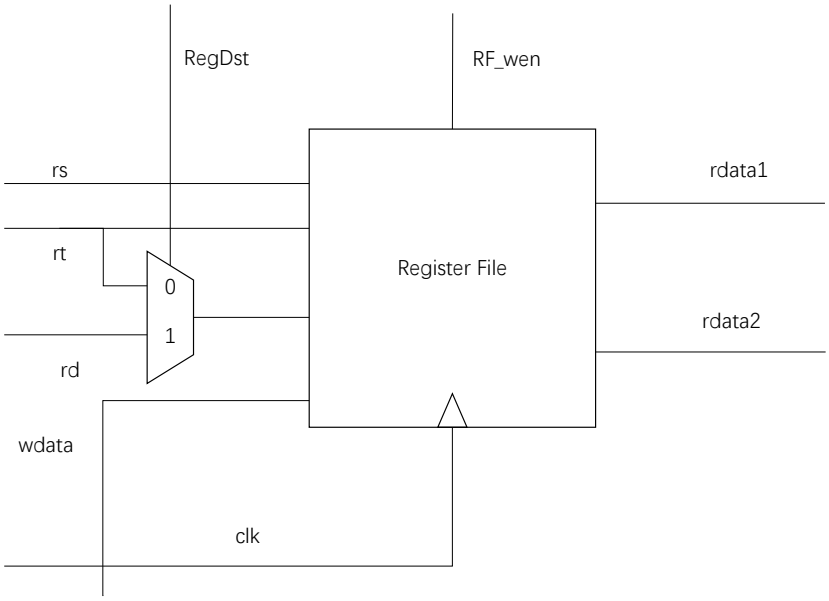

图 2: 寄存器堆简图

当指令类型为 R-Type、Ic、Il 中的一种时，需要将寄存器的写使能拉高。另需注意的是，对 R-Type 的指令，当类型为 Rm 时，需要根据 ALU 判断的结果决定是否将写使能拉高，当类型为 Rj 中的 jr 时，可将写使能拉低，或者使其将数据存入零号寄存器中，当指令为 J-Type 中的 jal 时，也需要将写使能拉高。在需要向寄存器中写入数据的指令中，当指令类型为 Ic 或 Il 时，写入的寄存器为 rt，对其他的指令则为 rd。

```
assign RegWrite = isRcalculate | isRshift | isIcalculate | isIload;
assign RF_wen = RegWrite | isjalr | isjal | islui | ((ismovn & ~Zero) | (ismovz & Zero));

assign RegDst = isIload | isIcalculate | 0;
```

ALU 和移位器运算的操作数均为两个，需要根据指令决定操作数的来源，在单周期处理器中，移位器仅需处理 R-Type 类型中的移位指令，MIPS 的移位指令要求将 rt 寄存器中的数据进行移位，因此操作数 A 的值为 rdata2，而操作数 B 的值由 func[2] 的值决定，为 rs 寄存器的后三位或者 shamt 段的值。

```
assign shifterA = rdata2;
assign shifterB = (rdata1[4:0] & {5{func[2]}}) | (shamt & {5{~func[2]}});
```

在指令类型不为 Rm 时，ALU 的操作数 A 值为 rs 寄存器中的值，即为 rdata1，这是由于 Rm 指令需要将 rt 中的值与零比较后，将 rs 中的值存入 rd 寄存器，因此 ALU 的两个操作数为 rt 的值和 32'b0。ALU 的操作数 B 有三种可能，当指令类型为 Ic 中的逻辑运算时，值为进行 32 位零拓展的立即数，当指令类型位 Il、Is，或是 Ic 中的非逻辑运算时，值为进行了 32 位有符号拓展的立即数，其余情况下，B 的值为 rt 寄存器的值，即为 rdata2。

```
assign A = (rdata1 & {32{~isRmov}}) | (32'b0 & {32{isRmov}});//if instr type is R-mov, then A = 0;
assign B = (rdata2 & {32{~ALUSrc}})
        | (signed_imm32 & {32{ALUSrc & ~isLogicalOpi}})
        | (unsigned_imm32 & {32{isLogicalOpi}});
```

移位器的操作分为左移、无符号数右移以及有符号数右移，可根据指令 func 段决定。而 ALU 的操作类型在指令类型为 Rc 时，可由 func 决定，在指令类型为 Ic 时，可由 opcode 段决定，但除此之外，还需要对一部分指令

额外指定操作类型，包括条件跳转，内存读数和存数，以及 Rm 类指令。

```
assign Shiftop = func[1:0];

assign ALUop =  ({func[1],2'b10} & {3{isAddSub}})
    | ({func[1],1'b0,func[0]} & {3{isLogicalOp}})
    | ({~func[0],2'b11} & {3{isComp}})
    | ({opcode[1],2'b10} & {3{isAddSubi}})
    | ({opcode[1],1'b0,opcode[0]} & {3{isLogicalOpi}})
    | ({~opcode[0],2'b11} & {3{isCompi}})
    | (3'b110 & {3{isbeq | isbne}})        //ALU操作为减法
    | (3'b111 & {3{isblez | isbgtz}})      //ALU操作为有符号数比较
    | (3'b111 & {3{isREGIMM}}              //ALU操作为有符号数比较
    | (3'b110 & {3{isRmov}}))              //ALU操作为减法
    | (3'b010 & {3{isIload | isIstore}}); //ALU操作为加法
```

在单周期处理器的设计中，存入寄存器的数据有五种来源，分别为从内存中取得的数据，ALU 运算的数据，移位器运算的数据，Rm 类指令写入，PC 返回地址写入。其中，PC 返回地址是因为 jal 和 jalr 指令跳转时，需要存入当前 PC 再加 8 的值，Rm 类指令存入的数据为 rs 寄存器的值。

```
assign memdatawrite = Load_data & {32{MemtoReg}};
    //lui的数据单独计算，但该指令的计算结果也视为aluwrite
assign aluwrite = (aluResult & {32{isRcalculate | isIcalculate}} & {32{~islui}}) |
                  (luiwrite & {32{islui}});
assign shiftwrite = shifterResult & {32{isRshift}};
assign movwrite = rdata1 & {32{isRmov}};
assign address_write = (return_address) & {32{isjalr | isjal}};        //return_address = PC + 4
assign RF_wdata = memdatawrite | aluwrite | shiftwrite | movwrite | address_write;
```

内存写使能只有在指令类型为 Is 时才会拉高。从内存中取数时，由于本次实验要求地址为 4 字节对齐，需要根据指令和计算得到取数地址对取出的数据进行再次处理后存入寄存器堆中。由于计算得出的地址不一定是四字节对齐的，因此需要做对齐处理，并将对齐抹去的后两位作为偏移量提前保存下来。

```
wire [31:0]lb,lh,lw,lbu,lhu,lwl,lwr;
wire offset_0,offset_1,offset_2,offset_3;


assign offset_0 = ~offset[1] & ~offset[0];
assign offset_1 = ~offset[1] & offset[0];
assign offset_2 = offset[1] & ~offset[0];
assign offset_3 = offset[1] & offset[0];

//set lb,lh,lw...
assign lb =  ({{24{Read_data[7]}},Read_data[7:0]} & {32{offset_0}})
     | ({{24{Read_data[15]}},Read_data[15:8]} & {32{offset_1}})
     | ({{24{Read_data[23]}},Read_data[23:16]} & {32{offset_2}})
     | ({{24{Read_data[31]}},Read_data[31:24]} & {32{offset_3}});

assign lh =  ({{16{Read_data[15]}},Read_data[15:0]} & {32{offset_0}})
         | ({{16{Read_data[31]}},Read_data[31:16]} & {32{offset_2}});
```

```
assign lw = Read_data;

assign lbu =  ({{24'b0},Read_data[7:0]} & {32{offset_0}})
     | ({{24'b0},Read_data[15:8]} & {32{offset_1}})
     | ({{24'b0},Read_data[23:16]} & {32{offset_2}})
     | ({{24'b0},Read_data[31:24]} & {32{offset_3}});

assign lhu =  ({{16'b0},Read_data[15:0]} & {32{offset_0}})
          | ({{16'b0},Read_data[31:16]} & {32{offset_2}});

assign lwl =  ({Read_data[7:0],rtdata[23:0]} & {32{offset_0}}) |
     ({Read_data[15:0],rtdata[15:0]} & {32{offset_1}}) |
     ({Read_data[23:0],rtdata[7:0]} & {32{offset_2}}) |
     (Read_data & {32{offset_3}});
assign lwr =  ({rtdata[31:8],Read_data[31:24]} & {32{offset_3}}) |
     ({rtdata[31:16],Read_data[31:16]} & {32{offset_2}}) |
     ({rtdata[31:24],Read_data[31:8]} & {32{offset_1}}) |
     (Read_data & {32{offset_0}});

//select the result
assign Load_data = (lb & {32{~opcode[2] & ~opcode[1] & ~opcode[0]}}) |
        (lh & {32{~opcode[2] & ~opcode[1] & opcode[0]}}) |
        (lw & {32{~opcode[2] & opcode[1] & opcode[0]}}) |
        (lbu & {32{opcode[2] & ~opcode[1] & ~opcode[0]}}) |
        (lhu & {32{opcode[2] & ~opcode[1] & opcode[0]}}) |
        (lwl & {32{~opcode[2] & opcode[1] & ~opcode[0]}}) |
        (lwr & {32{opcode[2] & opcode[1] & ~opcode[0]}});
```

以 lbu 为例，需要读取计算所得地址中存储的一个字节，并将这个字节存入寄存器的最低 8 位，可利用偏移量从内存中读到的 32 位数据中选出所需的 8 位。

Is 类型的指令类似，但除了写入的数据外，还多出了字节有效信号，因此还需要根据指令和偏移量对其进行赋值。

```
wire sb,sh,sw,swl,swr;
wire offset_0,offset_1,offset_2,offset_3;
wire is0001,is0010,is0100,is0011,is0111,is1111,is1110,is1100,is1000;
wire issb1,issb2,issb3,issb4,isshhigh,isshlow;
wire swleft1,swleft2,swleft3,swleft4,swright1,swright2,swright3,swright4;

assign sb = ~opcode[2] & ~opcode[1] & ~opcode[0];
assign sh = ~opcode[2] & ~opcode[1] & opcode[0];
assign sw = ~opcode[2] & opcode[1] & opcode[0];
assign swl = ~opcode[2] & opcode[1] & ~opcode[0];
assign swr = opcode[2] & opcode[1] & ~opcode[0];

assign offset_0 = ~offset[1] & ~offset[0];
assign offset_1 = ~offset[1] & offset[0];
assign offset_2 = offset[1] & ~offset[0];
assign offset_3 = offset[1] & offset[0];
```

```
//store the left/right n bytes of rtdata
assign swleft1 = swl & offset_0;
assign swleft2 = swl & offset_1;
assign swleft3 = swl & offset_2;
assign swleft4 = swl & offset_3;
assign swright4 = swr & offset_0;
assign swright3 = swr & offset_1;
assign swright2 = swr & offset_2;
assign swright1 = swr & offset_3;

//store n-th byte of rtdata
assign issb1 = sb & offset_0;
assign issb2 = sb & offset_1;
assign issb3 = sb & offset_2;
assign issb4 = sb & offset_3;

assign isshlow = sh & offset_0;   //store lower 2 bytes
assign isshhigh = sh & offset_2;  //store higher 2 bytes

//set Write_strb
assign is0001 = issb1 | swleft1;
assign is0010 = issb2;
assign is0100 = issb3;
assign is0011 = isshlow | swleft2;
assign is0111 = swleft3;
assign is1111 = sw | swleft4 | swright4;
assign is1110 = swright3;
assign is1100 = swright2 | isshhigh;
assign is1000 = swright1 | issb4;

assign Write_data = ({24'b0,rtdata[31:24]} & {32{swleft1}})
       | ({16'b0,rtdata[31:16]} & {32{swleft2}})
       | ({8'b0,rtdata[31:8]} & {32{swleft3}})
       | (rtdata & {32{swleft4 | swright4 | issb1 | isshlow | sw}})
       | ({rtdata[23:0],8'b0} & {32{swright3 | issb2}})
       | ({rtdata[15:0],16'b0} & {32{swright2 | issb3 | isshhigh}})
       | ({rtdata[7:0],24'b0} & {32{swright1 | issb4}});
assign Write_strb = (4'b0001 & {4{is0001}})
       | (4'b0010 & {4{is0010}})
       | (4'b0100 & {4{is0100}})
       | (4'b0011 & {4{is0011}})
       | (4'b0111 & {4{is0111}})
       | (4'b1111 & {4{is1111}})
       | (4'b1110 & {4{is1110}})
       | (4'b1100 & {4{is1100}})
       | (4'b1000 & {4{is1000}});
```

- **多周期处理器设计**。

与单周期处理器相比，多周期处理器将一条指令在多个周期内完成，此时，需要根据取指、译码、执行、访存、写回五个阶段，将一条指令的完成拆分为多个阶段。为此，需要使用有限状态机为不同阶段的控制信号赋值，在控制单元内部，通过时序逻辑，每一个时钟周期跳转到下一个状态，根据当前指令的 opcode 段，为控制信号赋值，而针对 opcode 全为零的 R-Type，则需要在控制单元之外利用组合逻辑，根据 opcode 和 func 段设置一些额外的控制信号。我针对不同指令类别设置了有 13 个状态的有限状态机，状态转移图如下：

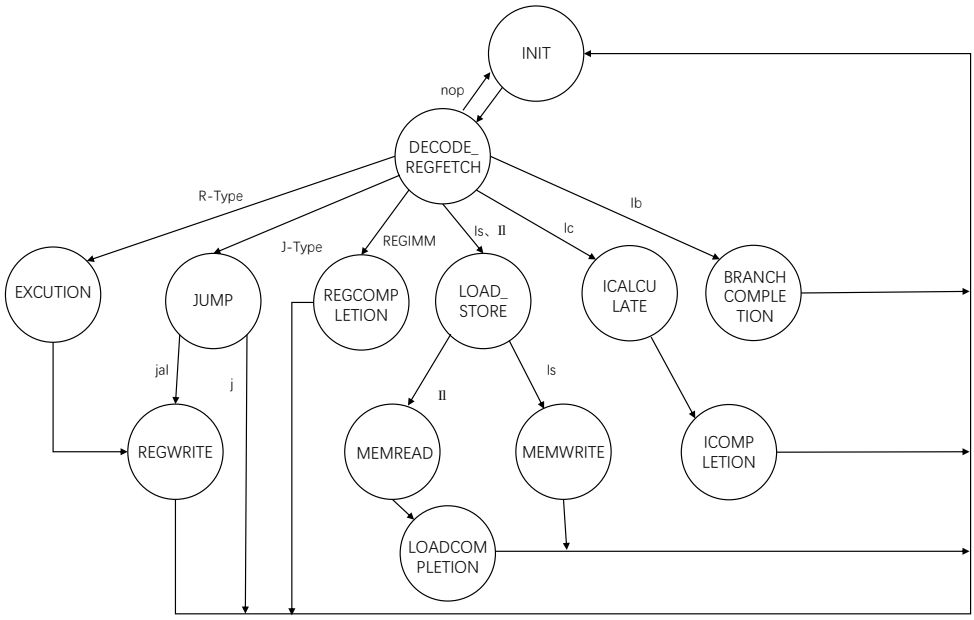

图 3: 多周期处理器状态转移图

在多周期处理器中控制逻辑变得更为复杂，需要考虑完成一条指令的过程中，同一个控制信号可能出现不同取值，相较于单周期处理器，对控制逻辑进行了一些改动。在控制单元中根据状态机状态赋值的控制信号如下。

	PCsrc	ALUOp	ALUsrcA	ALUsrcB	Shiftersrc	IorD	RegDst	RegWrite	MemRead	MemWrite	MemtoReg	IRWrite	PCWrite	PCWriteCond
INIT	00	000	0	010	01(x)	0	0(x)	0	1	0	0(x)	1	1	0(x)
DECODE_REGFETCH	01	000	0	001	00	0(x)	0(x)	0	0	0	0(x)	0	0	0
EXCUTION	11	100	1	000	01	0(x)	0(x)	0	0	0	0(x)	0	0	1
REGCOMPLETION	01	010	1	011	01(x)	0(x)	0(x)	0	0	0	0(x)	0	0	1
JUMP	10	000	0	010	01(x)	0(x)	0(x)	0	0	0	0(x)	0	1	0(x)
BRANCHCOMPLETION	01	010	1	000	01(x)	0(x)	0(x)	0	0	0	0(x)	0	0	1
ICALCULATE	00(xx)	101	1	100	10	0(x)	0(x)	0	0	0	0(x)	0	0	0
LOAD_STORE	00(xx)	000	1	100	01(x)	0(x)	0(x)	0	0	0	0(x)	0	0	0
REGWRITE	00(xx)	000(xxx)	0(x)	000(xxx)	01(x)	0(x)	1	1	0	0	0	0	0	0
ICOMPLETION	00(xx)	000(xxx)	0(x)	000(xxx)	01(x)	0(x)	0	1	0	0	0	0	0	0
MEMREAD	00(xx)	000(xxx)	0(x)	000(xxx)	01(x)	1	1(x)	0	1	0	0(x)	0	0	0
MEMWRITE	00(xx)	000(xxx)	0(x)	000(xxx)	01(x)	1	1(x)	0	0	1	0(x)	0	0	0
LOADCOMPLETION	00(xx)	000(xxx)	0(x)	000(xxx)	01(x)	0(x)	0	1	0	0	1	0	0	0

图 4: 多周期处理器控制单元信号

其中，PCsrc 用于选择下一条指令对应的 PC 值，ALUOp 用于选择 ALU 执行的运算种类，ALUsrcA 和 ALUsrcB 用于选择参与 ALU 运算的操作数，可以使用同一个 ALU 完成 PC+4、PC+offset 和常规的两操作数运算，同理，Shiftersrc 可用一个移位器完成移位操作，分支指令 offset 左移两位和 lui 指令。

- **ALU 设计**。

相较于 project1，单周期处理器中的 ALU 新增了异或，同或和无符号数比较操作，与 project1 中的 ALU 相比，3 位 ALUop 被占用的位置变多，需要相应的修改结果选择逻辑。

```
assign comp = ((sum[31]^Overflow) & ALUop[2]) | (CarryOut & ~ALUop[2]);
assign result_comp = {31'b0,comp};
//select the final result, using one-hot encoding
assign isand = ~(|ALUop);
```

```
assign isor = ALUop[0] & ~ALUop[1] & ~ALUop[2];
assign isxor = ~ALUop[0] & ~ALUop[1] & ALUop[2];
assign isnor = ALUop[0] & ~ALUop[1] & ALUop[2];
assign iscomp = ALUop[0] & ALUop[1];
assign issum = ~ALUop[0] & ALUop[1];

assign Result = ({32{isand}} & result_and)
     | ({32{isor}} & result_or)
     | ({32{isxor}} & result_xor)
     | ({32{isnor}} & result_nor)
     | ({32{issum}} & sum)
     | ({32{iscomp}} & result_comp);
assign Zero = ~(|Result);
```

- **Shifter 设计**。

  移位器有三种操作,分别是算数右移、逻辑右移和逻辑左移。

```
wire [`DATA_WIDTH-1:0]sll_result,sra_result,srl_result;
wire [1:0]sel;
assign sel = Shiftop;
assign sll_result = A << B;
assign sra_result = $signed(A) >>> B;
assign srl_result = A >> B;
assign Result = ({32{~sel[0] & ~sel[1]}} & sll_result)
     | ({32{sel[0] & sel[1]}} & sra_result)
     | ({32{~sel[0] & sel[1]}} & srl_result);
```

需要注意的是,Verilog 默认移位运算的数为无符号数,因此在不对变量额外声明的情况下,算数右移的实际效果是逻辑右移,为此,需要"$signed()"将变量声明为有符号数。

```
assign sra_result = $signed(A) >>> B;
```

二、 实验过程中遇到的问题、对问题的思考过程及解决方法(比如 RTL 代码中出现的逻辑 bug,逻辑仿真和 FPGA 调试过程中的难点等)

- **单周期处理器设计问题**。

在设计访存类指令的数据通路时,我遇到了较大的阻碍。MIPS 手册对于访问内存地址的要求和实验的要求不同,实验中要求内存读写地址对齐,而 MIPS 手册在存取字节和半字时未要求内存访问地址是四字节对齐的,因此在设计此类指令时不能完全按照手册上写。此外,在 MIPS 中,lwl、lwr、swl、swr 这四条指令本身的操作比较复杂,且手册上着重解释的是大端的情况,比较难实现。经过分析之后,我选择统一使用有效地址加偏移量的形式对写入或读出的数据进行处理,由 GPR[base]+offset 计算出实际的地址,再统一进行四字节对齐得到有效地址,并且把地址的最低两位作为一个小的偏移量记录下来。在加载数据时,可以根据有效地址读出 32 位数据,再根据偏移量和指令类型从中选出要存入寄存器的数据;在写入数据时,根据偏移量和指令类型从 rt 寄存器中选出需要的字节,然后安排在相应的位置,再根据 Write_strb 选择写入的字节,下面以 lb 和 sb 为例。

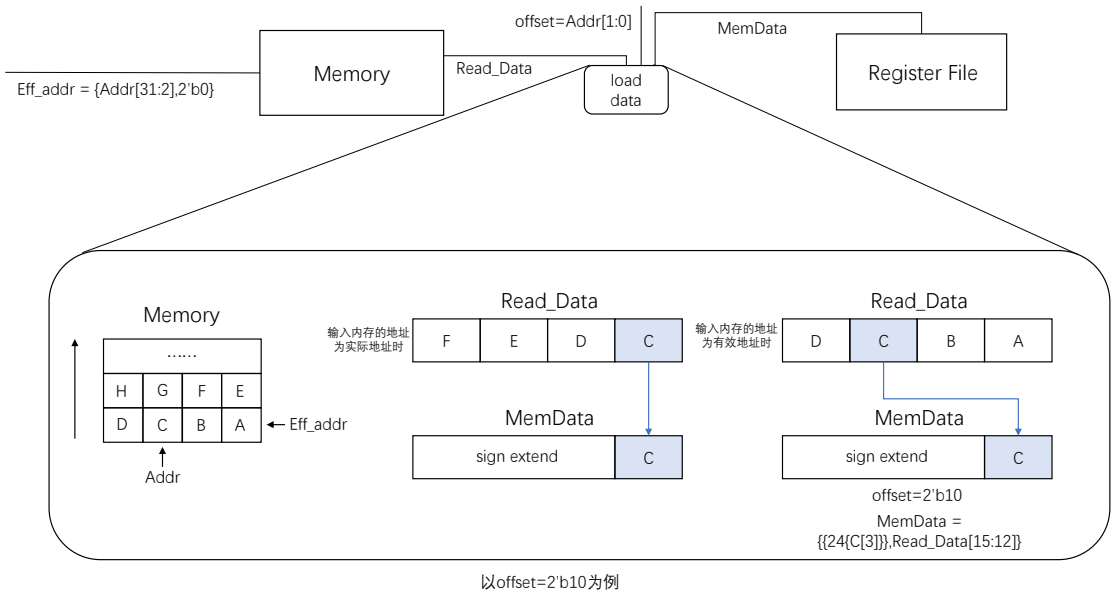

图 5: lb 示意图

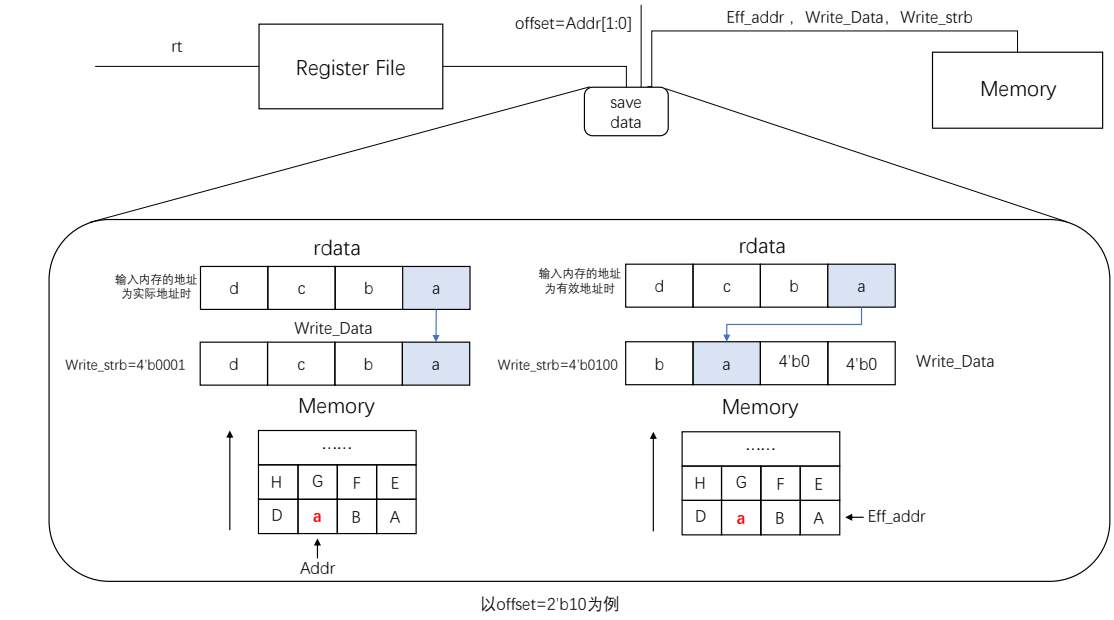

图 6: sb 示意图

lwl、lwr、swl、swr 这四条指令用这种方法也可以准确的计算出来

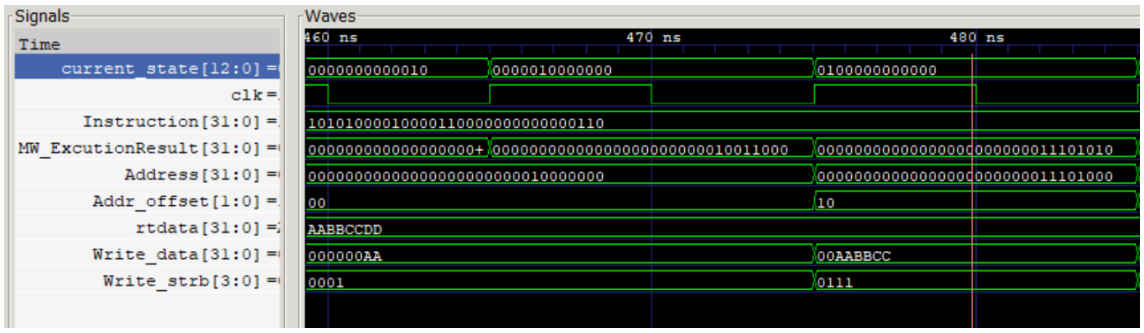

图 7: swl 波形图

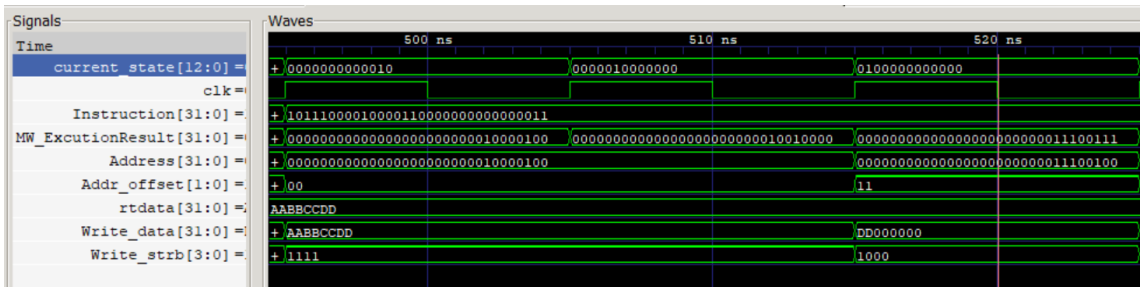

图 8: swr 波形图

可以看到在 current_state 为 13'b0100000000000 时，对 Write_data 和 Write_strb 的赋值有效，此时可以看到，待处理数据 rtdata 为 32'hAABBCCDD，对于 swl 指令，将最高的三字节赋给 Write_data 的最低三字节，再将 Write_strb 赋值为 4'b0111；对于 swr 指令，将最低的一字节赋给 Write_data 的最高一字节，再将 Write_strb 赋值为 4'b1000 这两条指令可以在内存中拼接成一个不对齐的四字节数据。

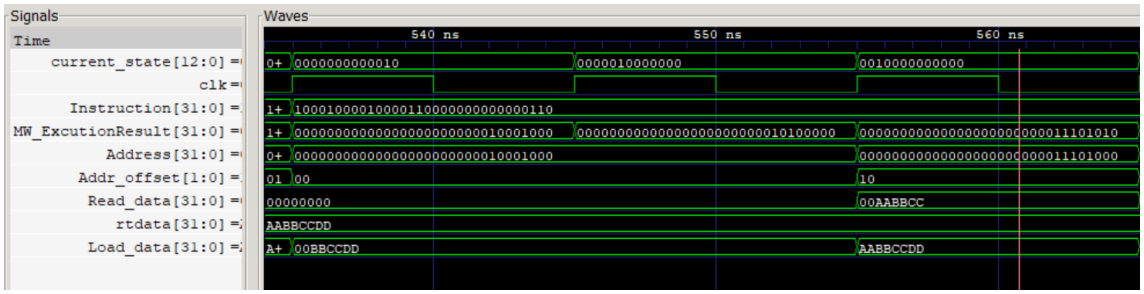

图 9: lwl 波形图

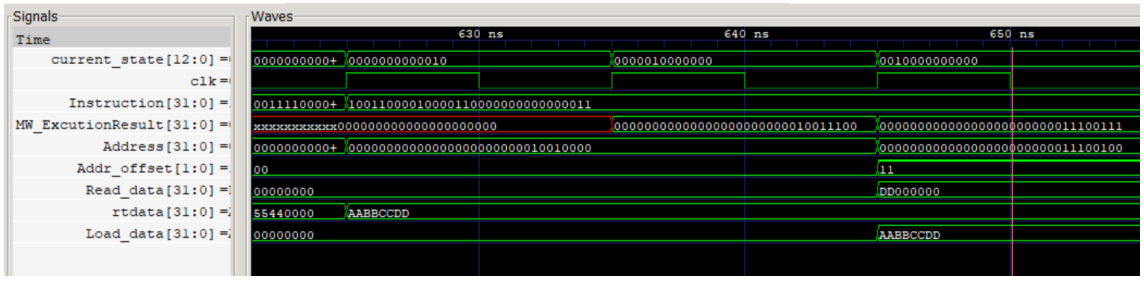

图 10: lwr 波形图

在 current_state 为 13'b0010000000000 时，对于 lwl 指令，寄存器的数据 rtdata 为 32'hAABBCCDD，

Read_data 为 32'b00AABBCC，此时将 Read_data 的最高三字节和 rtdata 的最低一字节拼接，得到 Load_data 为 32'hAABBCCDD；对于 lwr 指令，寄存器的数据 rtdata 为 32'hAABBCCDD，Read_data 为 32'bDD000000，此时将 Read_data 的最高一字节和 rtdata 的最高三字节拼接，得到 Load_data 为 32'hAABBCCDD。这样就向 rt 寄存器内存入了一个内存中不对齐的四字节数据。

- **多周期处理器设计问题**。

在设计多周期的过程中，我主要遇到的问题是 Rm 类指令的实现。首先，Rm 类指令在执行写回操作时并不总是写使能的，而时需要根据 rt 是否为零来判断是否写使能拉高，但在多周期处理器中，指令类型为 R-Type 时，统一对 RegWrite 赋值为 1，因此还需要额外添加判断条件。这就引发了另一个问题，判断 Rm 类指令是否能写回寄存器，需要依据执行阶段 ALU 产生的 Zero 标志位。

但修改后在运行一些 benchmark 时仍出现错误，检查相关 benchmark 的错误波形后发现是写使能错误导致寄存器存入的数据错误，此时的指令为 Rm 类，与 Zero 标志位错误有关，这说明直接使用 Zero 标志位是不行的。因为在写回阶段，由于信号的变化，ALU 直接生成的 Zero 信号会发生变化。因此还需要额外添加一个保存 Zero 值的寄存器，判断是否写回时，使用保存在该寄存器中的值。

- **ALU 设计问题**。在实验项目一中，ALU 的设计要求输出标志位 Zero，当时我的 Zero 是根据最终输出结果得到的，即为

```
assign Zero = ~(|Result);
```

但在对两数进行比较的操作中，Result 在 A<B 的情况下为 1，反之为 0，此时的 Zero 输出的结果和 Result 的结果恰好相反，相当于 Zero 的输出无用。

在处理器设计的过程中，我发现分支指令需要同时判断 A 与 B 的大小关系以及 A 是否等于 B，因此可以对 ALU 进行适当的修改，使得进行比较时输出值 Zero 可被用于判断 A、B 是否相等。

```
assign Zero = (~(|sum) & iscomp) | (~(|Result) & ~iscomp);
```

进行比较时，实际进行的是 A-B 的操作，用两数相减得到的值作为判断 Zero 的依据。不过要注意到的是，此时对 ALU 的修改会导致其无法通过实验项目一的测试。

三、 对讲义中思考题(如有)的理解和回答

- **思考 1:ALUop 编码**。

在本次的实验项目中，未考虑 add、addi、sub 三条指令，因此可以对 ALUop 的编码进一步优化。对 R-Type 类指令，当 func[3] 为 0 时，直接将 func 的后三位赋给 ALUop，当 func[3] 为 1 时，ALUop 赋值为 010 和 000. 对于 I-Type 类指令，由于没有考虑减法指令，可直接将 opcode 的后三位赋给 ALUop。

- **思考 2:控制单元的赋值**。

建议对控制信号赋值时只使用 current_state 的原因可能是为了使当前的状态机一经变化，就能立即为控制信号进行正确的赋值，而不用等到 next_state 变化后再得到正确的赋值。同时以 next_state 对控制信号赋值也会增加相应的判断逻辑的复杂度。

- **思考 3:状态机的状态数**。

在本次多周期处理器实验中，我共使用了 13 个状态，最初的想法视为了尽可能的利用输入控制单元的 opcode 段，以达到将外部控制信号减少的目的，但实际上最后发现并没有减少太多的外部控制信号。主要原因在于 R-type 类指令的 opcode 均为 1，需要靠 func 段进行分类，为了对分支跳转进行选择，仍需要对 Ib 类的每一条指令进行判断，并且为了根据 ALUOp 选择真正的 ALU 运算，也需要在外部对 Ic 类指令进一步划分。因此我认为此后可以对我的写法进一步进行改进，状态机可以只保留取指、译码、执行、访存、写回五个状态。

四、 实验感悟

本次实验项目内容是简单处理器的设计，实验主要关注的内容是简单处理器的数据通路和控制单元的设计。通过此次实验，我对处理器的基本运行过程有了一定的了解，掌握了按取指、译码、执行、访存、写回五个阶段设计处理器的方法；此外本次实验使用的指令集为 MIPS，因此我也对 RISC 指令的设计有了一定的认识；本次实验过程中，我通过 GTKWave 查看仿真波形解决了很多错误，对于硬件设计的这一调试方法也有了一定的心得。

五、 实验所耗时间

在课后，你花费了大约 40 小时完成此次实验。