中国科学院大学
《计算机组成原理(研讨课)》实验报告

姓名 韩初晓 学号 2023K8009908002 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 (基于 MIPS 32 位指令集)

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。

1 单周期 CPU 电路设计及说明

1.1 逻辑电路结构与译码表

我设计的 CPU 的主要参照了讲义上所给出的结构。但是, 我发现讲义上的结构并不能解决所有的问题, 比如说和 jump 有关的几个指令。因此, 我对总体结构进行了调整, 使之满足所有指令的功能。我的设计方案对应的电路结构图如下图所示:

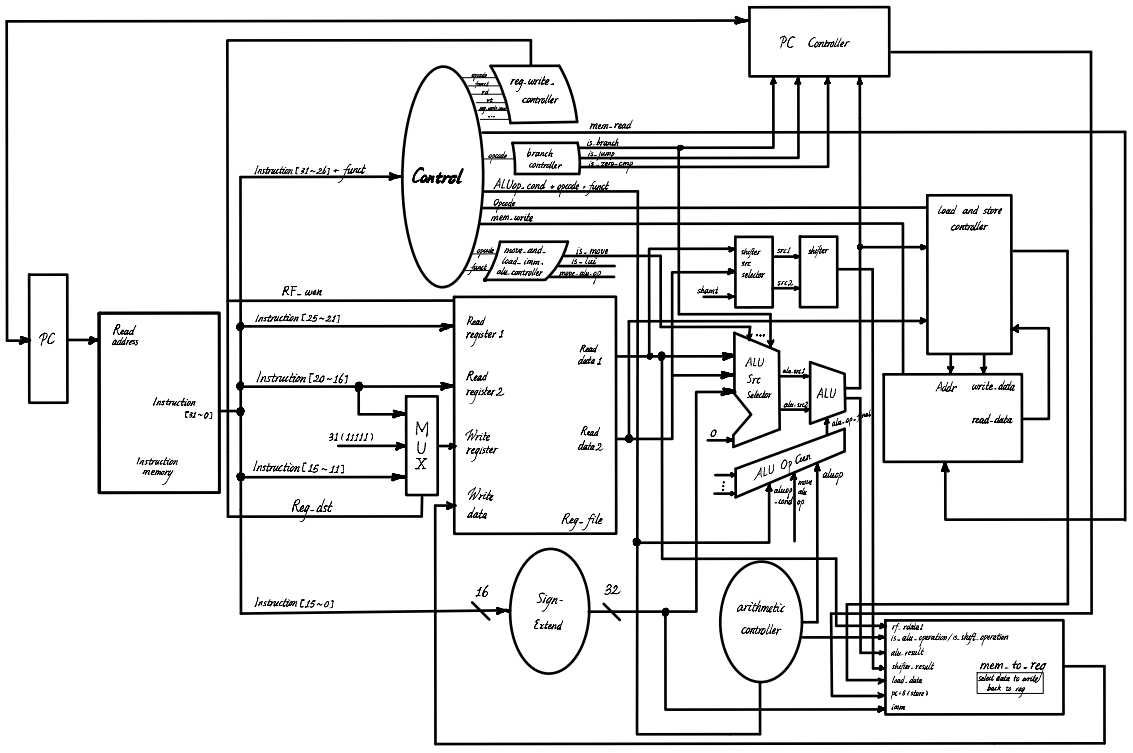

图 1: 简单功能型处理器逻辑电路图

在设计过程中,我为了实现更清晰的译码逻辑,我整理了一个译码表。译码表如下图所示:

Instruction Category	Mnemonic	Format	Description	Opcode	Funct	rs Usage	rt Usage	rd Usage	shamt Usage	imm Usage	target Usage	RegReadAddr1	RegReadAddr2	ALUSrcA	ALUSrcB
R-Type Arithmetic	add	R	Signed Add (Trap on Overflow)	000000	100000	Src1 Reg (rs)	Src2 Reg (rt)	Dst Reg (rd)	N/A	N/A	N/A	rs	rt	RegData1	RegData2
R-Type Arithmetic	addu		Unsigned Add		100001										
R-Type Arithmetic	sub		Signed Subtract (Trap on Overflow)		100010										
R-Type Arithmetic	subu		Unsigned Subtract		100011										
R-Type Logical	and		Bitwise AND		100100										
R-Type Logical	or		Bitwise OR		100101										
R-Type Logical	xor		Bitwise XOR		100110										
R-Type Logical	nor		Bitwise NOR		100111										
R-Type Compare	slt		Set on Less Than (Signed)		101010										
R-Type Compare	sltu		Set on Less Than (Unsigned)		101011										
R-Type Shift	sll		Shift Left Logical (by shamt)		000000										
R-Type Shift	srl		Shift Right Logical (by shamt)		000010	N/A	Src Reg (rt)		Shift Amount			N/A		N/A	
R-Type Shift	sra		Shift Right Arithmetic (by shamt)		000011										
R-Type Shift	sllv		Shift Left Logical Variable (by rs)		000100	Shift Amt Reg (rs)								RegData1(low 5 bits)	
R-Type Shift	srlv		Shift Right Logical Variable (by rs)		000110										
R-Type Shift	srav		Shift Right Arithmetic Variable (by rs)		000111										
R-Type Jump	jr		Jump Register		001000	Target Addr Reg (rs)	N/A	N/A					N/A	N/A	N/A
R-Type Jump	jalr		Jump and Link Register		001001			Dst Reg (rd default 31)							
R-Type Move Conditional	movn		Move Conditional on Not Zero		001011		Check Reg (rt)	Dst Reg (rd)				rs	rt		RegData2
R-Type Move Conditional	movz		Move Conditional on Zero		001010										
I-Type Arithmetic	addi	I	Signed Add Immediate (Trap on Overflow)	001000		Src Reg (rs)	Dst Reg (rt)	N/A	N/A	Signed Immediate	N/A		N/A	RegData1	ImmSE
I-Type Arithmetic	addiu		Unsigned Add Immediate	001001						ZeroExt Immediate					
I-Type Logical	andi		Bitwise AND Immediate	001100						Immediate		N/A		N/A	Imm << 16
I-Type Logical	ori		Bitwise OR Immediate	001101		N/A				Signed Immediate					
I-Type Logical	xori		Bitwise XOR Immediate	001110		Src Reg (rs)						rs	N/A	RegData1	ImmSE
I-Type Logical	lui		Load Upper Immediate	001111						Signed Offset					
I-Type Compare	slti		Set on Less Than Immediate (Signed)	001010		Base Addr Reg (rs)	Dst Reg (rt modified)						rt		
I-Type Compare	sltiu		Set on Less Than Immediate (Unsigned)	001011			Src Reg (rt)								
I-Type Load	lb		Load Byte (Signed)	100000											
I-Type Load	lbu		Load Byte (Unsigned)	100100											
I-Type Load	lh		Load Halfword (Signed)	100001											
I-Type Load	lhu		Load Halfword (Unsigned)	100101											
I-Type Load	lw		Load Word	100011											
I-Type Load	lwl		Load Word Left	100010											
I-Type Load	lwr		Load Word Right	100110											
I-Type Store	sb		Store Byte	101000											
I-Type Store	sh		Store Halfword	101001											
I-Type Store	sw		Store Word	101011											
I-Type Store	swl		Store Word Left	101010											
I-Type Store	swr		Store Word Right	101110											
I-Type Branch	beq		Branch on Equal	000100		Compare Reg1 (rs)	Compare Reg2 (rt)								RegData2
I-Type Branch	bne		Branch on Not Equal	000101											
I-Type Branch	blez		Branch on Less Than or Equal to Zero	000110		Compare Reg (rs)	N/A								Zero
I-Type Branch	bgtz		Branch on Greater Than Zero	000111											
I-Type Branch	bltz		Branch on Less Than Zero	000001	rt=00000										
I-Type Branch	bgez		Branch on Greater Than or Equal to Zero	000001	rt=00001										
J-Type Jump	j	J	Jump	000010	N/A	N/A				N/A	Target Address	N/A		N/A	N/A
J-Type Jump	jal		Jump and Link	000011											

图 2: MIPS 指令译码表 (第一部分)

Instruction Category	Mnemonic	Format	Description	ALUControl	ALU Operation Meaning	ShiftAmountSource	MemRead	MemWrite	MemAddressSource	MemWriteDataSource	MemWriteStrb	BranchCondition	NextPCSrc	RegWriteEnable	RegWriteAddrSource	RegWriteDataSource
R-Type Arithmetic	add	R	Signed Add (Trap on Overflow)	010	Add	N/A	0	0	N/A	N/A	N/A	N/A	PC+4	1	rd	ALUResult
R-Type Arithmetic	addu		Unsigned Add													
R-Type Arithmetic	sub		Signed Subtract (Trap on Overflow)	110	Subtract											
R-Type Arithmetic	subu		Unsigned Subtract													
R-Type Logical	and		Bitwise AND	000	AND											
R-Type Logical	or		Bitwise OR	001	OR											
R-Type Logical	xor		Bitwise XOR	100	XOR											
R-Type Logical	nor		Bitwise NOR	101	NOR											
R-Type Compare	slt		Set on Less Than (Signed)	111	Set Less Than (Signed)											
R-Type Compare	sltu		Set on Less Than (Unsigned)	011	Set Less Than (Unsigned)											
R-Type Shift	sll		Shift Left Logical (by shamt)	00	Shift Left Logical	shamt										ShifterResult
R-Type Shift	srl		Shift Right Logical (by shamt)	10	Shift Right Logical											
R-Type Shift	sra		Shift Right Arithmetic (by shamt)	11	Shift Right Arithmetic											
R-Type Shift	sllv		Shift Left Logical Variable (by rs)	00	Shift Left Logical	rs(low 5 bits)										
R-Type Shift	srlv		Shift Right Logical Variable (by rs)	10	Shift Right Logical											
R-Type Shift	srav		Shift Right Arithmetic Variable (by rs)	11	Shift Right Arithmetic											
R-Type Jump	jr		Jump Register	N/A	N/A							Always Jump	JumpRegisterTarget(rs)	0	N/A	N/A
R-Type Jump	jalr		Jump and Link Register											1	PC+8	PC+8
R-Type Move Conditional	movn		Move Conditional on Not Zero									!Zero		!zero	rd	RegData1
R-Type Move Conditional	movz		Move Conditional on Zero									Zero		Zero		
I-Type Arithmetic	addi	I	Signed Add Immediate (Trap on Overflow)	010	Add	N/A	1	0	ALUResult	SignExt(MemData[7:0])	N/A	N/A	PC+4	1	rt	ALUResult
I-Type Arithmetic	addiu		Unsigned Add Immediate							ZeroExt(MemData[7:0])						SignExt(MemReadData[7:0])
I-Type Logical	andi		Bitwise AND Immediate	000	AND					SignExt(MemData[15:0])						ZeroExt(MemReadData[15:0])
I-Type Logical	ori		Bitwise OR Immediate	001	OR					ZeroExt(MemData[15:0])						MemReadData
I-Type Logical	xori		Bitwise XOR Immediate	100	XOR					MemData[31:0]						Merge(RegData2 MemReadData)
I-Type Logical	lui		Load Upper Immediate	N/A	N/A					Merge Left (details complex)						Merge(RegData2 MemReadData)
I-Type Compare	slti		Set on Less Than Immediate (Signed)	111	Set Less Than (Signed)					Merge Right (details complex)						
I-Type Compare	sltiu		Set on Less Than Immediate (Unsigned)	011	Set Less Than (Unsigned)					Byte mask based on Addr[1:0]						
I-Type Load	lb	I	Load Byte (Signed)	010	Add (for address calc)	N/A	1	0	ALUResult	Halfword mask based on Addr[1]						
I-Type Load	lbu		Load Byte (Unsigned)							1111 (All bytes)						
I-Type Load	lh		Load Halfword (Signed)							Left byte mask (details complex)						
I-Type Load	lhu		Load Halfword (Unsigned)							Right byte mask (details complex)						
I-Type Load	lw		Load Word													
I-Type Load	lwl		Load Word Left													
I-Type Load	lwr		Load Word Right													
I-Type Store	sb		Store Byte				0	1	RegData2					0	N/A	N/A
I-Type Store	sh		Store Halfword													
I-Type Store	sw		Store Word													
I-Type Store	swl		Store Word Left													
I-Type Store	swr		Store Word Right													
I-Type Branch	beq	I	Branch on Equal	110	Subtract (for compare)	N/A	0	0	N/A	N/A	N/A	Zero == 1	BranchTarget			
I-Type Branch	bne		Branch on Not Equal									Zero == 0				
I-Type Branch	blez		Branch on Less Than or Equal to Zero									Sign==1 OR Zero==1				
I-Type Branch	bgtz		Branch on Greater Than Zero									Sign==0 AND Zero==0				
I-Type Branch	bltz		Branch on Less Than Zero	111	Set Less Than (Signed vs 0)							Result == 1				
I-Type Branch	bgez		Branch on Greater Than or Equal to Zero									Result == 0				
J-Type Jump	j	J	Jump	N/A	N/A							Always Jump	JumpTargetAddr	1	GPR[31]	PC+8
J-Type Jump	jal		Jump and Link													

图 3: MIPS 指令译码表 (第二部分)

1.2 关键模块结构设计说明

在 CPU 的设计过程中,ALU,移位器,PC 以及寄存器堆是四个关键模块。它们分别负责 CPU 的算术逻辑运算,数据的移位,程序计数器的更新和寄存器的读写操作。在设计 CPU 的过程中,我们对 ALU 进行了改动,并新加入了移位器和 PC 两个模块。因此,下面将首先对这三个模块进行设计说明。

1.2.1 ALU 设计

为了实现 CPU 的完整功能,ALU 需要实现 `slt` 和 `sltu` 两个功能。对于 `sltu` 指令,我们只需要设置 ALU 的工作模式为减法,同时查看 `carryout` 信号的值。当存在借位信号时,表示不够减。因此 `sltu` 指令的结果为 1 。否则,表示够减,因此 `sltu` 指令的结果为 0 。对于 `slt` 指令,我们可分三种情况,当 A 为正,B 为负时,结果一定为 0;当 A 为负,B 为正时,结果一定为 1。如果都不满足这两种情况,直接对 `overflow` 和 `sub_result` 的最高位进行异或,即可得到结果。事实上,直接使用异或而无需分类讨论即可得出正确的结果。但是如果能够直接根据操作数得到结果而不经过加法器运算,可能会提高性能。以下是我的 Verilog 代码设计。

```
assign slt_result =
    (A[`DATA_WIDTH-1] & !B[`DATA_WIDTH-1]) ? 1'b1 :
    (!A[`DATA_WIDTH-1] & B[`DATA_WIDTH-1]) ? 1'b0 :
    (sub_overflow ^ sub_result[`DATA_WIDTH-1]);
```

```
assign Result =
    (ALUop == 3'b000) ? (A & B) :
    (ALUop == 3'b001) ? (A | B) :
    (ALUop == 3'b010) ? (A + B) :
    (ALUop == 3'b100) ? (A ^ B) :
    (ALUop == 3'b101) ? ~(A | B) :
    (ALUop == 3'b110) ? sub_result[`DATA_WIDTH-1:0] :
    (ALUop == 3'b011) ? {{(`DATA_WIDTH-1){1'b0}}, sub_carryout} :
    (ALUop == 3'b111) ? {{(`DATA_WIDTH-1){1'b0}}, slt_result} :
    {`DATA_WIDTH{1'bx}};
```

1.2.2 移位器设计

在移位器的设计过程中,逻辑左右移的实现自然是比较简单的。需要作特殊考虑的是算术右移操作。我本打算使用 Verilog 内置功能 `$signed(A) >>> B` 来实现该功能。但是,在本地仿真的过程中出现了如下的错误:

```
================================================================
ERROR: at               2430ns.
Yours:    RF_waddr = 04, (RF_wdata & 0xffffffff) = 0x04c3b2a1
Reference: RF_waddr = 04, (RF_wdata & 0xffffffff) = 0xfcc3b2a1
================================================================
```

我发现,是本应当做算术右移的时候却做了逻辑右移。因此,最终我选择采用了逻辑右移 + 符号掩码的方式来实现算术右移。具体的实现方式如下:

```
wire sign_bit = A[`DATA_WIDTH - 1];
wire [`DATA_WIDTH-1:0] logical_shifted_A = A >> B;
wire [`DATA_WIDTH-1:0] sign_extension_mask = ({`DATA_WIDTH{sign_bit}} << (`DATA_WIDTH - B));
wire [`DATA_WIDTH-1:0] manual_sra_result = logical_shifted_A | sign_extension_mask;
```

1.2.3 PC 及其更新模块

为了最大化简化时序逻辑模块,我采用了 PC + PC 控制器的设计方式。PC 控制器掌管和 PC 更新有关的全部信息 (包含分支跳转和顺序执行),并输出 `next_pc` 值。PC 模块接受 `next_pc` 值,并仅负责在时钟上升沿更新 PC 的值。PC 控制器的设计如下:

是否进行 branch 跳转 ?

```
wire branch_condition_satisfied = (!opcode[2] && (funct_branch[0] ^ alu_result)) ||
                                  (opcode[1] && (opcode[0] ^ (alu_result[31] || alu_zero))) ||
                                  ((opcode[2] ^ opcode[1]) && (opcode[0] ^ alu_zero));
wire [31:0] next_pc_branch = (branch_condition_satisfied) ?
                             (current_pc + 4) + {imm_B_ext[29:0], 2'b00} :
                             current_pc + 4;
```

branch 类型操作共有六个,六个指令各自有自己满足跳转条件并跳转的规则。这对应着以下的逻辑表达式:

$$
\begin{aligned}
branch\_condition\_satisfied = &(beq \wedge Zero) \vee \\
&(bne \wedge \neg Zero) \vee \\
&(blez \wedge (Sign \vee Zero)) \vee \\
&(bgtz \wedge (\neg Sign \wedge \neg Zero)) \vee \\
&(bltz \wedge (alu\_result = 1)) \vee \\
&(bgez \wedge (alu\_result = 0))
\end{aligned}
$$

但是，直接使用上述逻辑表达式需要对 opcode 的每一位进行暴力匹配，使得电路复杂度过高。因此，我将跳转的规则直接和 opcode 的每一位建立了逻辑关系，得到 branch_condition_satisfied 信号。这样，能够降低电路的复杂程度，利于降低芯片面积。

是否进行 jump 跳转？

```
assign is_jump = (~|opcode && funct_jump[3] == 1'b1 && funct_jump[1] == 1'b0) ||
                 (~|opcode[5:2] && opcode[1] == 1'b1);
wire [31:0] next_pc_jump = (|opcode) ? {current_pc[31:28], imm_J, 2'b00} : rs;
```

以上代码判定了当前指令是否为 jump ，以及 jump 类型指令的跳转地址。

最终 next_pc 信号选择

```
assign next_pc = (is_jump) ? next_pc_jump :
                 (is_branch) ? next_pc_branch :
                               current_pc + 4;
```

当指令为 jump 时，next_pc 信号的值为 next_pc_jump；当指令为 branch 时，next_pc 信号的值为 next_pc_branch；否则，next_pc 信号的值为 current_pc + 4。

是否向寄存器堆写入 PC + 8

```
assign pc_store = (is_jump) ? current_pc + 8 : 0;
```

pc_store 信号的值为 current_pc + 8，当指令为 jump 时。否则，pc_store 信号的值为 0 。注意，事实上 jump 类指令中只有两个需要向寄存器堆写入 current_pc + 8，但是在这里，我们对所有 jump 指令都给出了这个信号，其余的工作由寄存器堆写管理器 (reg_write_controller) 来完成，这一模块将在后面详细阐述。

1.3 其他重要模块结构设计说明

除了以上介绍的三个关键模块，我设计的 CPU 中还包含了其他的重要模块，我将在下面对它们进行详细介绍：

1.3.1 控制单元

在设计控制单元时，我原先想象中的设计思路是，像讲义中所展示的那样，向控制单元中输入 opcode，从控制单元中输出 RegDst、Branch、MemRead、MemtoReg、ALUOp、MemWrite、ALUSrc、RegWrite 八种控制信号。但是，当我在观察译码表时，我发现这是不可能完成的。因为许多控制信号的产生不仅依赖于操作码，还依赖于功能码等一些其他的信息。因此，我最终采取了“双层”设计的控制单元。第一层是主控制单元 (control_unit)，仅接受操作码输入，产生所有**仅通过操作码**就能够产生的控制信号，同时产生中间信号给到第二层控制模块。第二层控制模块则比较专业化，每个模块负责一类指令，比如寄存器堆写控制器 (reg_write_controller) 用来产生 reg_dst 和 RF_wen 信号；分支控制器 (branch_controller) 判定是否是分支、跳转类指令以及判断分支指

令的 ALU 操作数中是否含有零;数据移动及立即数加载控制器 (move_and_load_imm_controller) 判定是否为 move 和 lui 指令, 并产生相应的 ALUop; 算术控制器 (arithmetic_controller) 掌管各种立即数运算, 并为 ALU 和移位器传递控制信号。下面对这几个模块进行详细介绍:

主控制单元

直接能根据操作码产生的信号有: mem_write, mem_read, 立即数算术运算的 ALU 操作码, 分支类指令的 ALU 操作码,以及加载和存数指令的 ALU 操作码。这几个信号的产生逻辑如下:

```
//------------------------------------------------------------------------------
// Memory Read Control (mem_read)
//------------------------------------------------------------------------------
// Assert mem_read for Load instructions (opcode 100xxx)
assign mem_read = (opcode[5:3] == 3'b100);

//------------------------------------------------------------------------------
// Memory Write Control (mem_write)
//------------------------------------------------------------------------------
// Assert mem_write for Store instructions (opcode 101xxx)
assign mem_write = (opcode[5:3] == 3'b101);
```

立即数运算的 ALUop 的产生逻辑如下:

```
// Decode I-type arithmetic operation type to generate ALU hint (arith_type)
wire [2:0] arith_type;
assign arith_type[0] = (~opcode[2] & opcode[1]) | (opcode[2] & opcode[0]);
assign arith_type[1] = ~opcode[2];
assign arith_type[2] = opcode[1] & ~opcode[0];
```

事实上,上述代码也是通过对下面的原始代码进行真值表化简得到的:

```
// Decode I-type arithmetic operations for alu_op_cond
wire addiu = (opcode == 6'b001001);
wire slti = (opcode == 6'b001010);
wire sltiu = (opcode == 6'b001011);
wire andi = (opcode == 6'b001100);
wire ori  = (opcode == 6'b001101);
wire xori = (opcode == 6'b001110);

wire [2:0] arith_type =
                    addiu ? 3'b010 : // ADD
                    slti ? 3'b111 : // SLT
                    sltiu ? 3'b011 : // SLTU
                    andi ? 3'b000 : // AND
                    ori  ? 3'b001 : // OR
                    xori ? 3'b100 : // XOR
                    3'bxxx;         // Should not happen for imm_arithmetic
```

分支类指令的 ALU 操作码的逻辑则比较简单,对于需要执行 slt 操作的分支指令,其 ALU 操作码为 111; 对于需要执行减法操作的分支指令,其 ALU 操作码为 110。加载和存数类指令的 ALU 操作码则一直为 010. 因此,最终在主控制单元内产生的 ALUop 的选择逻辑如下:

```
// Determine alu_op_cond based on instruction type:
// - BLTZ/BGEZ: SLT (for sign bit check)
// - BEQ/BNE/BLEZ/BGTZ: SUB (for zero/sign check)
// - Load/Store: ADD (for address calculation)
// - I-type Arith: Based on arith_type (ADD, SLT, SLTU, AND, OR, XOR)
assign alu_op_cond =
                  is_branch_slt ? 3'b111 :  // SLT for BLTZ/BGEZ
                  is_branch_sub ? 3'b110 :  // SUB for BEQ/BNE/BLEZ/BGTZ
                  is_load_store ? 3'b010 :  // ADD for address calculation (Load/Store)
                  imm_arithmetic ? arith_type : // Arith type for I-type arithmetic
                  3'bxxx;                   // Default (R-type or others, not used)
```

最终，在主控制单元中产生的 ALUop 毕竟不能涵盖所有的情况。因此我们需要定义一个信号来表示什么时候这个 ALUop 是可用的。这个信号如下所示：

```
// alu_op_ok: Flag indicating alu_op_cond is valid for ALUOp Generator
assign alu_op_ok = (imm_arithmetic || is_branch || is_load_store);
```

这个信号将会在最终输入给 ALUop 生成器，用于生成最终的 ALUop。

寄存器堆写控制器 reg_write_controller

在主控制单元模块中，我们通过 opcode[5] ^ opcode[3] 对寄存器的写地址进行了一步粗略的筛选。这个信号输入给寄存器写控制器，进一步生成最终的寄存器写地址。最终的寄存器写地址的生成逻辑如下：

```
//------------------------------------------------------------------------------
// Final Destination Register Address Selection (reg_dst)
//------------------------------------------------------------------------------
// Determine the final register destination address based on instruction type
// and control signals:
// - If reg_dst_input is asserted:
//   - For I-type and Load instructions (where reg_dst_input selects 'rt'): use 'rt'.
// - For JAL instruction (opcode[0] is asserted, and reg_dst_input selects 'ra'): use $ra
    (register 31).
// - Default (for instructions that do not write or don't care): use 'rd' (though it won't be
    written).
assign reg_dst =
               (reg_dst_input) ? rt :     // Use 'rt' if reg_dst_input is asserted (I-type, Loads)
               (opcode[0])   ? 5'b11111 : // Use $ra (31) for JAL
                               rd;        // Default or R-type (using 'rd' when reg_dst_input is
                                   asserted for R-type)
```

在主控制模块中，我们对寄存器堆写信号做出了初步筛选，在本模块中，我们对寄存器堆写信号进行了进一步的筛选。最终的寄存器堆写信号的生成逻辑如下：

```
wire intermediate_reg_write_cond = (input_reg_write_cond && !is_j && !is_jr);
assign wen =
           (intermediate_reg_write_cond && !is_move) || // Normal write condition (not MOVZ/N)
           (is_move && (funct[0] ^ is_zero));           // MOVZ/MOVN condition is satisfied
```

分支控制器 branch_controller

分支控制器的主要工作除了判断是否为分支指令之外，其另一个重要工作是判断 ALU 操作数是否含有零。

因为 BLEZ,BGTZ 和 REGIMM 这三类指令都需要 ALU 的其中一个操作数是零。因此,可以由 branch_controller 来判断这件事情,并将该信号输入给 ALU 操作数选择器。该信号的产生规则如下:

```
// is_zero_cmp: True for BLEZ, BGTZ (opcode check) and BLTZ, BGEZ (REGIMM opcode check)
assign is_zero_cmp = is_branch &&
                  (opcode[2:0] == 3'b110 || // BLEZ
                   opcode[2:0] == 3'b111 || // BGTZ
                   opcode[2:0] == 3'b001);  // REGIMM (includes BLTZ, BGEZ)
```

数据移动和立即数加载控制器 move_and_load_imm_controller

由于 move 和 lui 是两个比较特殊的指令,因此需要一个模块来判断当前指令是否为这两个指令。同时,move 指令需要 ALUop 取 001,这也是这个模块的职责之一。事实上,由于这个模块比较简单,后续我发现这个模块是完全可以省略的。不过,在这里我还是希望展现我一开始设计它的想法。这个模块的主要功能包含下面的三句话:

```
//------------------------------------------------------------------------------
// Conditional Move Instruction Detection (MOVZ, MOVN)
//------------------------------------------------------------------------------
// --- Optimized Implementation (direct logic formula based on funct bits) ---
// Formula Breakdown:
// - ~|opcode: Ensures it's an R-type instruction (opcode is all zeros).
// - funct[1]: Funct bit 1 is common to both MOVZ and MOVN (funct[1] == 1).
// - funct[3]: Funct bit 3 is common to both MOVZ and MOVN (funct[3] == 1).
// - !funct[5]: Funct bit 5 is 0 for both MOVZ and MOVN (funct[5] == 0).
assign is_move = (~|opcode && funct[1] && funct[3] && !funct[5]);
assign move_alu_op = 3'b001; // OR operation

//------------------------------------------------------------------------------
// Load Upper Immediate Instruction Detection (LUI)
//------------------------------------------------------------------------------
// LUI opcode is 001111 (binary), which means bits [3:0] are all 1s.
assign is_lui = (&opcode[3:0]); // Check if opcode[3:0] are all '1's (AND reduction)
```

注:move 指令的判断也使用了与操作码和功能码直接相联系的逻辑表达式,在这里不再赘述。

算术控制器 arithmetic_controller

我们知道,和立即数算术运算有关的 ALUop 能够直接由主控制单元通过指令操作码得出。但是,如果是寄存器算术运算,其 ALUop 和 Shifterop 就将由此模块给出。同时,该模块还给出了何时移位器需要选择 shamt 作为操作数的信息。具体的实现如下:

```
//------------------------------------------------------------------------------
// ALU Operation Decoding (alu_op)
//------------------------------------------------------------------------------
// r_type_arith_op: Decodes the funct field for R-type arithmetic/logic operations
wire [2:0] r_type_arith_op;
assign r_type_arith_op[0] = (funct[3] & funct[1]) | (~funct[3] & funct[2] & funct[0]);
assign r_type_arith_op[1] = funct[3] | ~funct[2];
assign r_type_arith_op[2] = (funct[3] & ~funct[0]) | (~funct[3] & funct[1]);

//------------------------------------------------------------------------------
// Shifter Operation Decoding (shifter_op) and shamt Selection (is_shamt)
//------------------------------------------------------------------------------
```

```
// shift_type: Decodes the funct field for R-type shift operations.
wire [1:0] shift_type;
assign shift_type[0] = funct[1] & funct[0];
assign shift_type[1] = funct[1];

// is_shamt: Shamt field usage indicator.
// Indicates if the shift amount for the Shifter should come from the `shamt` field of the
    instruction.
// This is true for SLL, SRL, SRA (R-type shifts with immediate shift amount).
assign is_shamt = is_shift_operation && ~funct[2]; // shamt used when funct[2] is 0 in shift
    instructions
```

加载和存数控制器 load_and_store_controller

在四十五条指令之中，最复杂的指令无疑是加载和存数类指令。为此，我在这里引入了专门对这类指令进行处理的控制器。

对于加载类指令，这个模块对从内存读来的数据进行位扩展；同时，对于非对齐类加载指令，这个模块会把从内存中读取的数据和寄存器中原先的数据进行裁切和拼合，再重新写回到寄存器中，来实现非对齐加载只将数据加载到寄存器高位或低位的目的。具体的代码实现如下：

```
// -- Load Data Processing (load_data) --
assign load_data = (lb | lbu) ?
    (addr_0 ? {{24{(lb & mem_data [ 7])}}, mem_data [ 7: 0]} :
     addr_1 ? {{24{(lb & mem_data [15])}}, mem_data [15: 8]} :
     addr_2 ? {{24{(lb & mem_data [23])}}, mem_data [23:16]} :
              {{24{(lb & mem_data [31])}}, mem_data [31:24]} ) :

    (lh | lhu) ?
     (addr_0 ? {{16{(lh & mem_data [15])}}, mem_data [15: 0]} :
     addr_2 ? {{16{(lh & mem_data [31])}}, mem_data [31:16]} :
              32'b0 ) :

    lw ? mem_data :
     lwl ?
     (addr_0 ? {mem_data [ 7: 0], rf_rdata2 [23: 0]} :
     addr_1 ? {mem_data [15: 0], rf_rdata2 [15: 0]} :
     addr_2 ? {mem_data [23: 0], rf_rdata2 [ 7: 0]} :
              mem_data ) :
    lwr ?
     (addr_0 ? mem_data :
     addr_1 ? {rf_rdata2 [31:24], mem_data [31: 8]} :
     addr_2 ? {rf_rdata2 [31:16], mem_data [31:16]} :
              {rf_rdata2 [31: 8], mem_data [31:24]} ) :
    32'b0;
```

对于存数类指令，这个模块会将寄存器中的数据和内存中原先的数据进行裁切和拼合，再重新写回到内存中，来实现非对齐存数只将数据存入内存高位或低位的目的。具体的代码实现如下：

```
// Memory address output is always word-aligned
assign mem_addr_o = {mem_addr[31:2], 2'b00};
```

```
// -- Byte Strobe Calculation (mem_strb) --
// Determine which byte lanes are active for a write operation
assign mem_strb =
     ({4{sb}} & ( // Store Byte
        (addr_0 ? 4'b0001 : 0) | (addr_1 ? 4'b0010 : 0) |
        (addr_2 ? 4'b0100 : 0) | (addr_3 ? 4'b1000 : 0)
     )) |
     ({4{sh}} & ( // Store Halfword
        (addr_0 ? 4'b0011 : 0) | (addr_2 ? 4'b1100 : 0)
     )) |
     ({4{sw}} & ( // Store Word (aligned)
        addr_0 ? 4'b1111 : 4'b0000
     )) |
     ({4{swl}} & ( // Store Word Left
        (addr_0 ? 4'b0001 : 0) | (addr_1 ? 4'b0011 : 0) |
        (addr_2 ? 4'b0111 : 0) | (addr_3 ? 4'b1111 : 0)
     )) |
     ({4{swr}} & ( // Store Word Right
        (addr_0 ? 4'b1111 : 0) | (addr_1 ? 4'b1110 : 0) |
        (addr_2 ? 4'b1100 : 0) | (addr_3 ? 4'b1000 : 0)
     ));

// -- Memory Write Data Formatting (mem_wdata) --
// Prepare the 32-bit data word to be written based on the store type and alignment
assign mem_wdata =
     ({32{sb}} & ( // Store Byte: Replicate byte across all lanes (strobe selects)
        {4{rf_rdata2[7:0]}}
     )) |
     ({32{sh}} & ( // Store Halfword: Replicate halfword across lanes
        {2{rf_rdata2[15:0]}}
     )) |
     ({32{sw}} & ( // Store Word: Use full register data
        rf_rdata2
     )) |
     ({32{swl}} & ( // Store Word Left: Shift data based on address offset
        (addr_0 ? {24'b0, rf_rdata2[31:24]} :    // Write MSB
         addr_1 ? {16'b0, rf_rdata2[31:16]} :    // Write upper 2 bytes
         addr_2 ? { 8'b0, rf_rdata2[31: 8]} :    // Write upper 3 bytes
                  rf_rdata2)                     // Write all 4 bytes
     )) |
     ({32{swr}} & ( // Store Word Right: Shift data based on address offset
        (addr_0 ? rf_rdata2 :                    // Write all 4 bytes
         addr_1 ? {rf_rdata2[23:0], 8'b0} :      // Write lower 3 bytes
         addr_2 ? {rf_rdata2[15:0], 16'b0} :     // Write lower 2 bytes
                  {rf_rdata2[ 7:0], 24'b0})      // Write LSB
     ));
```

对存储字节指令，将原寄存器低八位数据复制四次，拼成 32 位数据，并配合只含有一个 1 的掩码使用；对存储半字指令，将原寄存器低十六位数据复制两次，拼成 32 位数据，并配合含两个 1 的掩码使用；对存储字指令，直

接向内存发送原寄存器中的数据，并配合 1111 掩码。非对齐存储字的规则比较复杂，需要根据 ALU 计算出的地址的后两位的不同而产生不同的数据和掩码。具体的规则已经在上面的代码中给出。

1.3.2 ALU 附属模块

以上的控制单元产生了许多控制信号。例如，不止一个模块产生了 ALUop。将这些纷繁复杂的控制信号直接输入到 ALU 中是行不通的。因此，在将控制信号输入到 ALU 之前，我们需要对它们进行筛选，得到在当前时刻真正有效的那一个，这就是 ALU 附属模块的主要功能。

ALU 操作数选择器 (alu_src_selector)

这个模块用于选择输入 ALU 的操作数 A 和 B。它们分别对应 alu_src1 和 alu_src2。具体的选择逻辑如下：

```
//------------------------------------------------------------------------
// ALU Source 1 (Operand A) Selection Logic
//------------------------------------------------------------------------
// Selects the first operand for the ALU (alu_src1) based on instruction type:
// Priority is checked in the order listed below.
assign alu_src1 =
               (is_move) ? 32'b0 :
                rs;

//------------------------------------------------------------------------
// ALU Source 2 (Operand B) Selection Logic
//------------------------------------------------------------------------
// Selects the second operand for the ALU (alu_src2) based on instruction type:
// Priority is checked in the order listed below.
assign alu_src2 =
               (is_branch_zero_cmp) ? 32'b0 :
               (alu_src_imm_input) ?
               (use_zero_extend ? imm_0E : imm_SE) :
                rt;
```

第一个操作数仅会在检测到 move 指令时被置为零，其他情况下均为寄存器 rs 的值。第二个操作数的选择则比较复杂，首先判断是否为与零比大小的分支指令，如果是，则将其置为零；如果不是，则判断是否为立即数运算，如果是，则根据 use_zero_extend 信号的值来决定使用零扩展还是符号扩展；如果都不是，则直接使用寄存器 rt 的值。

ALU 操作码生成器 (alu_op_generator)

本模块对控制单元生成的众多 ALU 操作码进行筛选，最终输出给 ALU。具体的实现如下：

```
// Determine the final ALU operation with priority:
// 1. MOVZ/MOVN: Use the specific move_alu_op.
// 2. R-type/I-type Arithmetic: Use alu_op from Arithmetic Controller.
// 3. Load/Store/Branch: Use alu_op_cond hint from Control Unit.
// 4. Default: ADD (e.g., if no other condition applies, though usually one should).
assign alu_op_final = (is_move)          ? move_alu_op :
                      (is_alu_operation) ? alu_op :
                      (alu_op_ok)        ? alu_op_cond :
                      3'b010;
```

正如注释所写的那样，当是 move 指令时，ALU 操作码为从数据移动控制器中输出的 move_alu_op；当是 R 型指令或者立即数运算时，ALU 操作码为从算术控制器中输出的 alu_op；当是加载、存数或者分支指令时，ALU 操作码为主控制单元中输出的 alu_op_cond；否则，默认状态下 ALU 操作码为 010。

1.3.3 移位器附属模块

和 ALU 附属模块一样，移位器附属模块也是对控制单元输出的众多控制信号进行筛选，最终输出给移位器。事实上，移位器附属模块仅需要一个，那就是移位器操作数选择器 (shifter_src_selector)。这个模块的主要功能是选择移位器的操作数 A 和 B。它们分别对应 shifter_src1 和 shifter_src2。具体的选择逻辑如下：

```
// Shifter Source 1 is always the data from register rt
assign shifter_src1 = rt;

// Shifter Source 2 is the shift amount:
// If is_shamt is true (SLL, SRL, SRA), use the 5-bit shamt field.
// Otherwise (SLLV, SRLV, SRAV), use the lower 5 bits of register rs.
assign shifter_src2 = (is_shamt) ? {27'b0, shamt} : {27'b0, rs[4:0]};
```

1.3.4 写回阶段控制模块

写回阶段的控制模块和讲义上所描述的有很大的差异。讲义上仅仅通过一个 MUX 对来自内存的数据和来自 ALU 的计算结果进行了一次选择并写回到了寄存器中。但是在实际设计时，我发现，写入寄存器堆的数据可能有六种。因此，我引入了写回数据源选择器 (mem_to_reg_controller)。事实上，一开始我对这个信号的英文名字 (mem_to_reg) 很是疑惑，但我求证之后发现，这确实是写回数据源选择信号的名称。这个选择器在六个输入信号中选择一个写回寄存器中，具体逻辑如下：

```
// Select the data source for register writeback based on instruction type priority.
// The conditions are evaluated sequentially using the following priority:
// Priority 1: LUI result
// Priority 2: Data loaded from memory
// Priority 3: ALU result (non-move arithmetic/logic)
// Priority 4: Shifter result
// Priority 5: rs data (Conditional Moves)
// Priority 6: Link address (JAL/JALR), PC + 8
assign write_data =
               (is_lui)             ? lui_result :
               (memread)            ? mem_data :
               (is_alu_operation && !is_move) ? alu_result :
               (is_shift_operation) ? shift_result :
               (is_move)            ? rs :
               (is_jump)            ? pc_store :
               32'b0;
```

1.4 仿真波形图

下面是仿真过程中的波形图，这些波形图能够展示上述模块工作的过程。

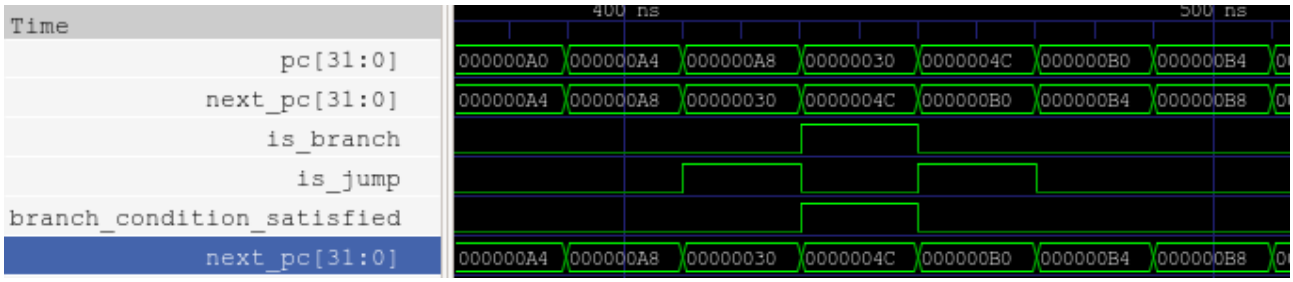

图 4: PC 及分支控制器仿真波形图

图 4 为分支控制器仿真波形图。可以看出，当分支条件满足以及检测到 `jump` 类型指令时，程序计数器发生跳转。

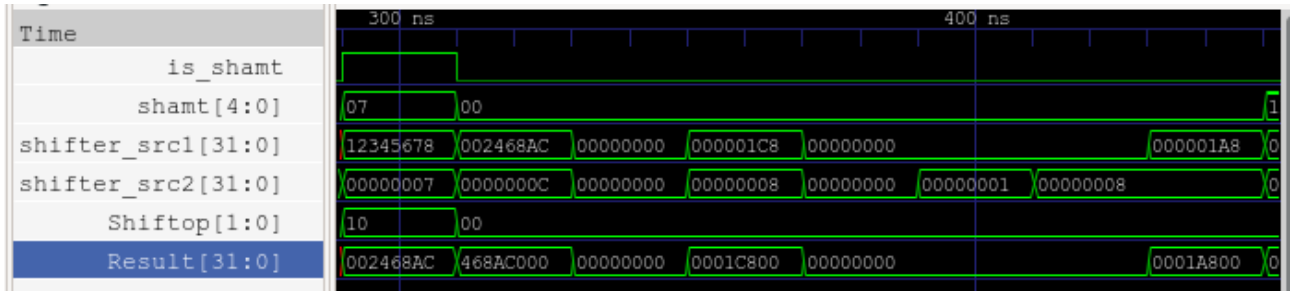

图 5: 移位器及其控制器仿真波形图

图 5 为移位器及其控制器的仿真波形图。可以看出，当 `is_shamt` 信号被拉高时，移位器将 `shamt` 作为其中一个操作数，并正确的完成移位操作。

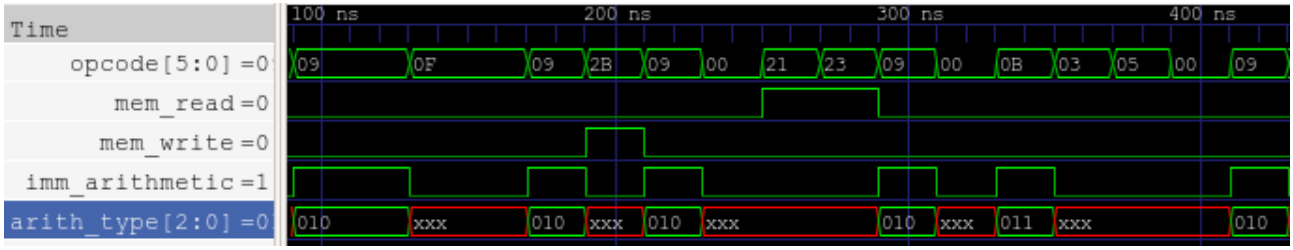

图 6: 主控制单元仿真波形图

图 6 为主控制单元的仿真波形图。可以看出，主控制单元能正常根据操作码判断出内存读写信号，同时也能够正常的给出立即数算术运算的 ALUop。

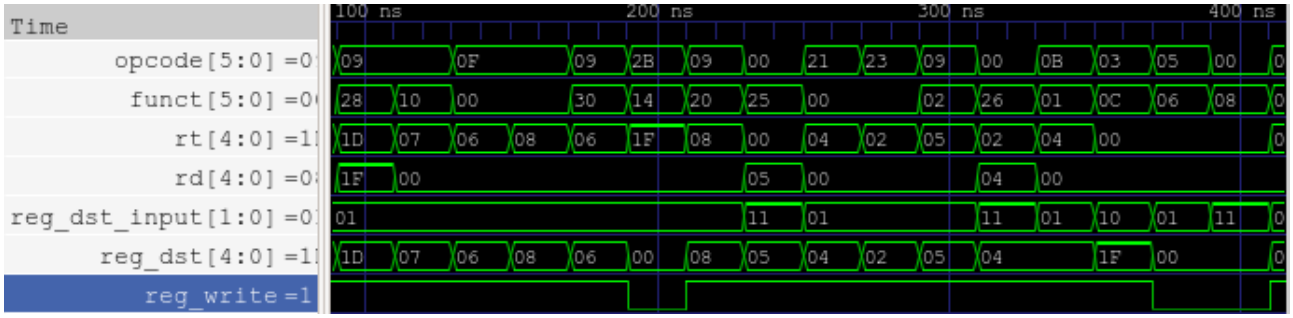

图 7: 寄存器堆写控制器仿真波形图

图 7 为寄存器堆写控制器的仿真波形图。可以看出，寄存器堆写控制器能够正确根据控制信号对 `rd`，`rt` 和 `1F`(三十一号寄存器) 进行选择，并且能够适时开启寄存器堆写使能 (`reg_write`) 信号。

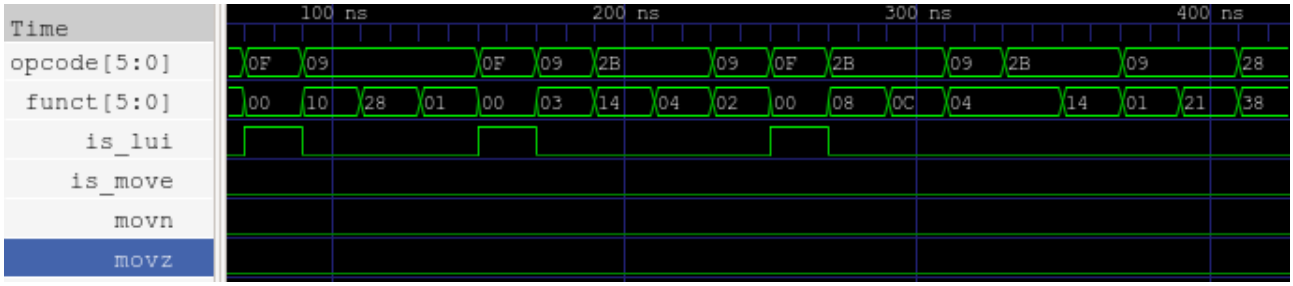

图 8: 数据移动和立即数加载控制器仿真波形图 (LUI 指令)

图 8 为立即数加载控制器的仿真波形图。可以看出，控制器能够正确检测 `lui` 指令。令人震惊的是，这个名为 `movsx` 的程序竟没有一条 `move` 指令。因此，`move` 指令的正确性在这张图片中无法展现。

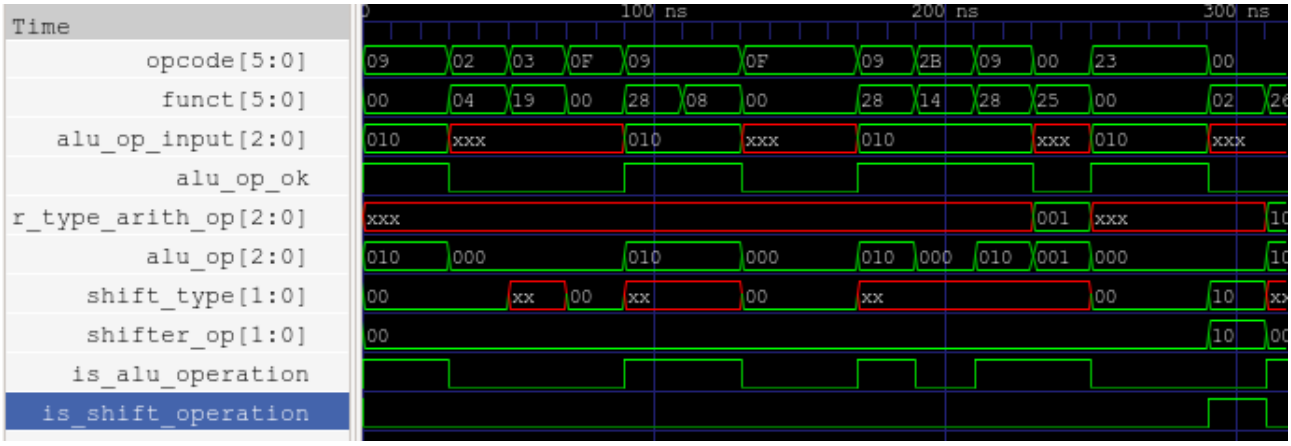

图 9: 算术控制器仿真波形图

图 9 为算术控制器的仿真波形图。可以看出，算术控制器能够正确的根据操作码和功能码来判断 ALUop 和 Shifterop，并且能够正确的给出当前为 ALU 运算还是 Shifter 运算的信号。同时，在来自主控制模块的 ALUop 可用时，模块将会直接将来自主控制模块的 ALUop 作为输出 ALUop。

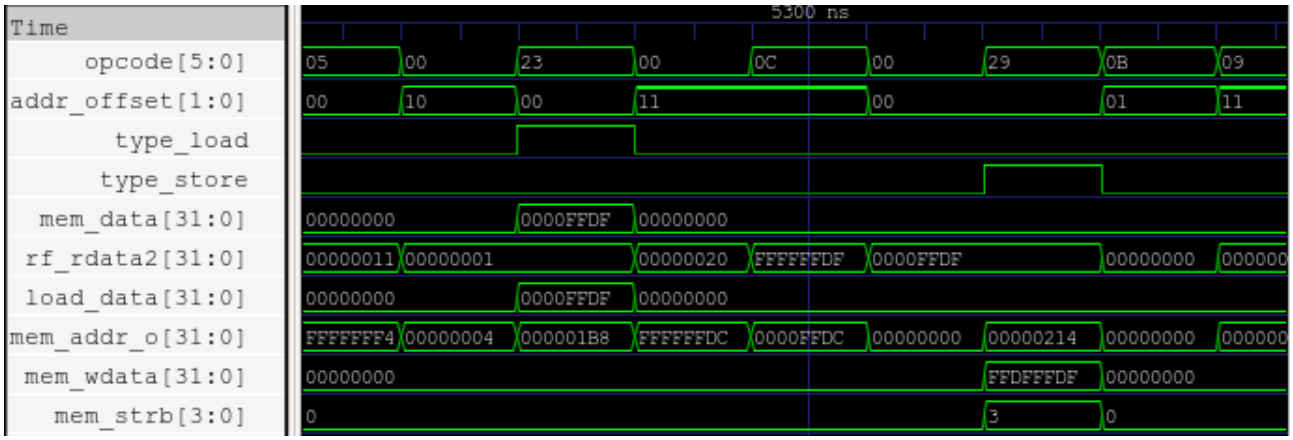

图 10: 加载和存数控制器仿真波形图

图 10 为加载和存数控制器的仿真波形图。可以看出，加载和存数控制器能够正确的根据 ALU 计算出的地址来判断当前是对齐还是非对齐加载/存储，并且能够正确的给出加载/存储的数据。我们可以观察到图中 `type_store` 被拉高的时刻。可以看出这是一条加载半字指令：从寄存器堆中读入的数据 `0000FFDF` 的低两个字节被复制了两次，变成 `FFDFFFDF`，作为内存写数据，同时，控制器正确的产生了掩码 `0011`，即图中的 3。因此，此模块的实现是正确的。

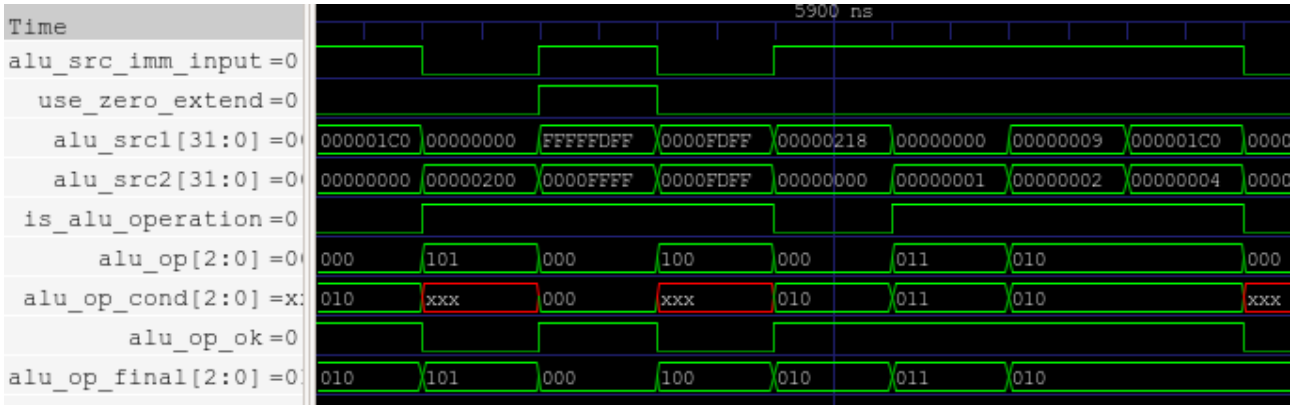

图 11: ALU 附属模块仿真波形图

图 11 为 ALU 附属模块的仿真波形图。可以看出，ALU 操作码生成器能够根据输入的 ALUop 正确的生成 ALU 操作码。ALU 操作数选择器能够根据输入的 ALU 操作数选择信号正确的选择 ALU 的操作数。

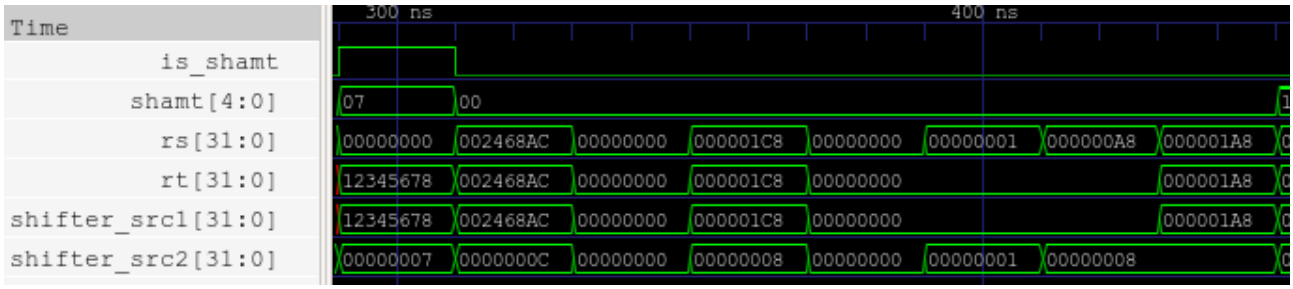

图 12: 移位器操作数选择器仿真波形图

图 12 为移位器操作数选择器的仿真波形图。可以看出，移位器操作数选择器能够根据输入的移位器操作数选择信号正确的选择移位器的操作数。

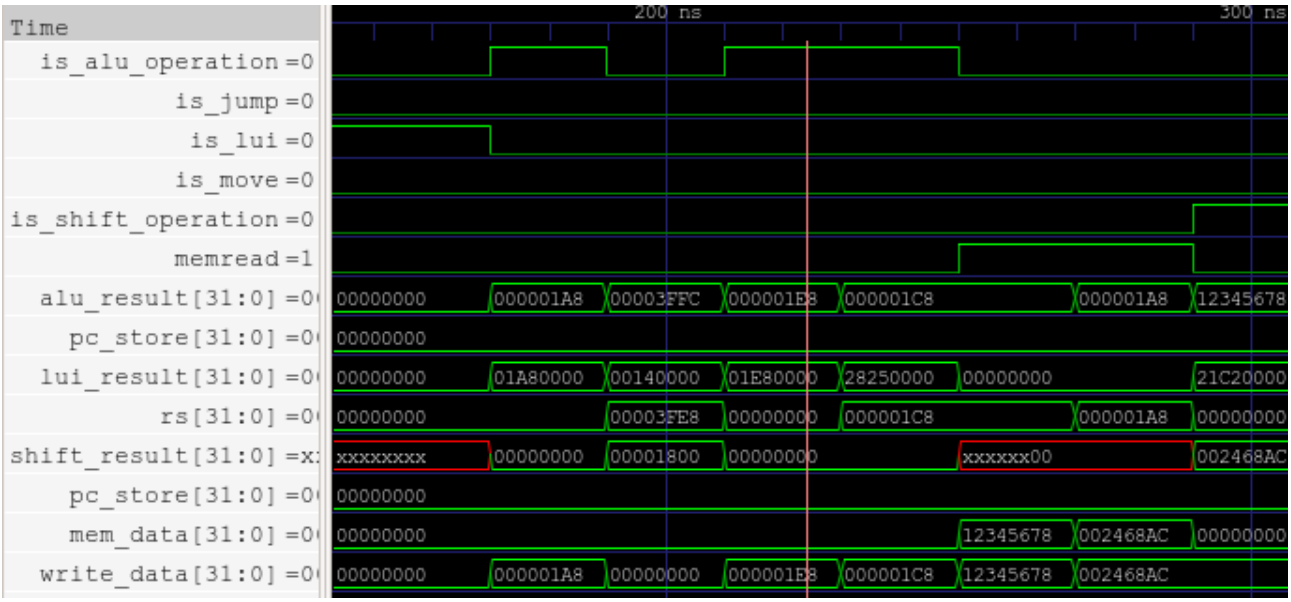

图 13: 写回数据源选择器仿真波形图

图 13 为写回数据源选择器的仿真波形图。图中,前六个信号是选择信号。谁被拉高,最终的写入数据就会选择谁对应的信号。可以看出,图中数据源选择器的工作是完全正常的。

2 多周期 CPU 电路设计及说明

2.1 RTL 设计及说明

在设计多周期 CPU 的过程中,我主要参照了讲义中所给出的 FSM 的状态图:

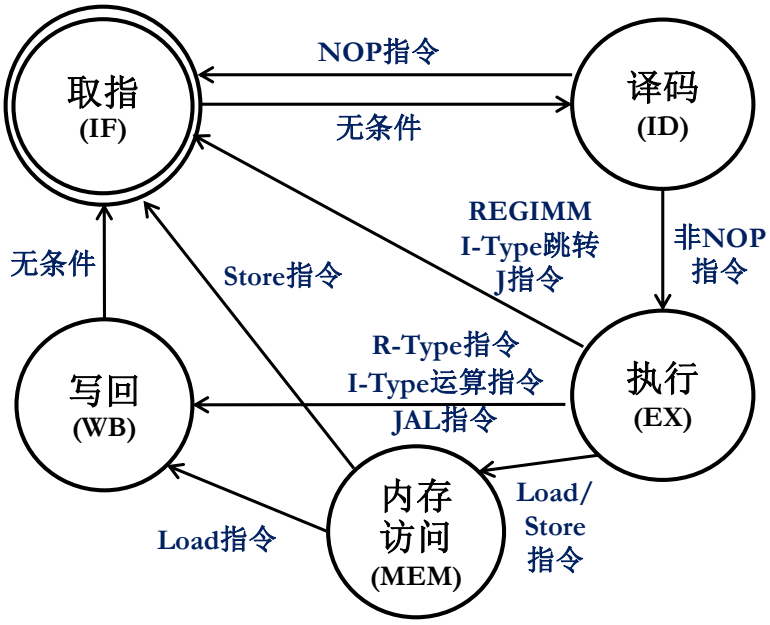

图 14: FSM 状态转移图

图 14 为有限状态自动机的状态转移图。据此,我设计了下面的状态机模块代码:

```
//=============================================================================
// PIPELINE STAGE 0: FSM Implementation
//=============================================================================

// -- FSM State Definitions --
localparam IF = 5'b00001,
          ID = 5'b00010,
          EX = 5'b00100,
          MEM = 5'b01000,
          WB = 5'b10000;

// -- FSM State Register --
always @(posedge clk) begin
  if (rst) begin
    current_state <= IF;     // Reset state
  end else begin
    current_state <= next_state; // Update state based on FSM logic
  end
end

// -- FSM Next State Logic --
always @(*) begin
  case (current_state)
    IF : next_state = ID;
    ID : begin
         if (NOP) begin
           next_state = IF;
         end
         else begin
           next_state = EX;
         end
    end
    EX : begin
      if (is_j_instr || is_REGIMM || is_branch) begin
        next_state = IF;
      end
      else if (is_load_store) begin
        next_state = MEM;
      end
      else begin
        next_state = WB;
      end
    end
    MEM : begin
      if (opcode[3]) begin
        next_state = IF;
      end
      else begin
        next_state = WB;
```

```
      end
    end
    WB : next_state = IF;
    default: next_state = IF;
endcase
end
```

在多周期 CPU 中，许多当前状态下产生的信号需要到下个周期使用，因此我们需要将许多单周期 CPU 中的信号转化为寄存器类型，并让它们在时钟的上升沿更新。

首先，ALU 和移位器的计算结果需要 MEM 阶段或 WB 阶段使用。因此需要将 ALU 和移位器的计算逻辑转变为如下的形式：

```
//-- ALU Execution --
wire [31:0] alu_out;           // ALU computation result

alu instance_alu (
    .A         (alu_src1),      // Input A from Src Sel
    .B         (alu_src2),      // Input B from Src Sel
    .ALUop     (alu_op_final),  // Input Opcode from Op Gen
    .Overflow  (),
    .CarryOut  (),
    .Zero      (alu_zero),      // Output: Zero flag -> To EX (PC Ctrl), WB
    .Result    (alu_out)        // Output: ALU computation result -> To EX (PC Ctrl), MEM, WB
);

always @(posedge clk) begin
    alu_result <= alu_out;      // Store ALU result for MEM stage
end

//-- Shifter Execution --
wire [31:0] shifter_out;          // Shifter computation result

shifter instance_shifter (
    .A         (shifter_src1),      // Input Data from Src Sel
    .B         (shifter_src2[4:0]), // Input Shift Amount from Src Sel
    .Shiftop   (shifter_op),        // Input Opcode from ID
    .Result    (shifter_out)        // Output: Shifter result -> To WB
);

always @(posedge clk) begin
    shift_result <= shifter_out;   // Store Shifter result for WB stage
end
```

其次，指令寄存器需要在新指令到来之时更新，若无新指令，则保持其原来的值。这个信号由 IRWrite 来控制：

```
assign IRWrite = current_state[0];
// -- Instruction Register Logic --
always @(posedge clk) begin
  if (IRWrite) begin
```

```
    IR <= Instruction;
  end else begin
    IR <= IR;
  end
end
```

同时，PC 的更新逻辑也需要调整。PC 不再是每个周期更新一次，而是在控制信号 pc_write_enable 存在时才更新。

```
wire pc_write_enable = (current_state == IF) ||
                       ((current_state == EX) && is_jump) ||
                       ((current_state == EX) && is_branch && branch_condition_satisfied);

//================================================================================
// Module: pc
// Description: Program Counter register. Stores the address of the next
//              instruction to be fetched.
//================================================================================
module pc (
  input       clk,
  input       rst,
  input       pc_write_enable,
  input [31:0] next_pc,
  output reg [31:0] pc
);
always @(posedge clk or posedge rst) begin
  if (rst) begin
    pc <= 32'h00000000;
  end else if (pc_write_enable) begin
    pc <= next_pc;
  end
end
endmodule
```

最后，写回的逻辑也需要转化为时序逻辑。因为写回的信号来自 EX 和 MEM 阶段。修改后，写回数据源选择器的实现如下：

```
//-- Write Back Data Selection (Uses ID signals, EX results, MEM results) --
wire [31:0] _RF_wdata;                                          // Write data to RegFile
    (processed from ID, EX, MEM)
assign     _RF_wdata       = (is_lui)          ? lui_result :   // P1: LUI (from ID)
                             (mem_read_internal) ? load_data :   // P2: Load result from MEM
                             (is_move)         ? RF_rdata1 :     // P3: MOVZ/MOVN result
                             (is_alu_operation) ? alu_result :   // P4: Other ALU result from EX
                             (is_shift_operation) ? shift_result : // P5: Shifter result from EX
                             (is_link_jump)    ? pc_store :      // P6: PC+8 from EX
                             32'b0;                              // Default: 0 (No writeback)
                                                                 // -> To RegFile instance
always @(posedge clk) begin
    RF_wdata <= _RF_wdata;                                       // Store write data for RegFile
```

```
end
```

2.2 仿真波形图

图 15: 多周期 CPU 仿真波形图

图 15 为多周期 CPU 的仿真波形图。从图中可以观察出以下几点特征：首先，该 CPU 确实是多周期的，不同的指令执行的时钟周期数不同。其次，有限状态自动机工作正常，随着时钟周期的变化而更新其状态。此外，在 MEM 状态 (08) 下，`MemRead` 信号被拉高，说明 CPU 正常进入内存读取状态；在 WB 状态 (10) 下，`RF_wen` 被拉高，说明 CPU 正常进入寄存器堆写回状态。另外，ALU 计算结果在每次 EX 阶段结束后都会更新，符合我们定义的时序逻辑更新规则。最后，当检测到分支指令且 EX 阶段 (02) 结束后，CPU 直接进入 IF 阶段，符合我们定义的自动机状态转换规则。

该 CPU 模块通过了全部的仿真测试。

3 实验过程中遇到的问题、对问题的思考过程及解决方法

聪明设计还是蠢设计？

我设计单周期 CPU 的过程中，所参考的材料主要有两个，一是译码表，二是讲义上所给的结构图。在设计过程中，我一直幻想着能同讲义上的结构图一样，以模块化的方式来完成 CPU 的设计。事实上我也是这样做的。因此，在我的最初几版设计中，我的主模块中没有任何的逻辑表达式，只是将各个模块组织并连接起来。但后来，我逐渐意识到这种设计方式的弊端，那就是模块之间无法高效的共享信号——如果希望一个模块所产生的中间信号为其他模块所用，那就必须在二者之间接一条线。同时，大量模块实例化占用了很多的行数，导致我最初的几版设计中，有三分之一的代码是在实例化模块。后来，经过和同学们的讨论，我了解到了另一种设计思路，那就是抛开结构图不谈，仅仅从译码表出发，寻找各个信号和 opcode 等输入信号之间的逻辑关系。通过这种思路设计出来的代码整体看起来就是将每个信号都转化成了一个优雅的合取范式，并且不包含任何的暴力匹配逻辑，十分美观。后来，我尝试对自己代码中的模块化逻辑进行了简化，取消了实例化，直接将每个模块对应的功能写在主模块当中。这种做法使我的代码长度缩短至二百多行，同时也使得模块之间信号共享的问题得以解决。但是我的设计中始终保留着原来模块化设计的影子，不如我前面提到的另一种设计方式来得简洁、美观。不过，虽然从代码结构和美观程度上讲，我的设计看起来比较蠢，但是这种设计反映了讲义中所提到的 CPU 的结构，是比较容易理解和最容易想到的设计方式。

为什么使用 $signed(A) >>> B 仍然会导致未预期行为？

这可能是因为,不同的仿真器处理该语句中的符号扩展可能会出现不同的行为。即便使用 $signed(A) 进行强制转换,某些仿真器 (比如我们本地仿真用到的 iverilog) 可能仍然不会保留符号信息,导致 >>> 仍然表现出逻辑右移的行为。因此,为了使得代码在不同的仿真器中都能有正确的行为,我们最好采取符号掩码 + 逻辑移位的方式来处理算术移位。

单周期 CPU 如何做到在一个周期内实现寄存器堆读写？

寄存器堆的读操作和 ALU 的运算都是组合逻辑,因此可以在时钟上升沿到来之前完成读取数据和运算的操作。寄存器堆写是时序逻辑,会在时钟上升沿到来的时候执行写入操作。

什么样的错误最容易犯？

在自己写代码和与同学交流的过程中,我总结出了几类在代码中很容易出现,但是在第一次 debug 过程中却很难找出的错误:

- **我感觉我的波形图在出错的地方是没问题的**。可能是因为看错了模块,在波形图中把金标准当成了自己写的模块。同时需要注意,波形图往往会停在下一个时钟周期的上升沿,但是不会显示下一个时钟周期的完整数据。我们在 debug 时需要看的并不是这个时钟周期,而是展现完整数据的最后一个时钟周期。
- **我将一个信号赋值给了另一个信号,但是在波形图中它们并不相等**。这种错误一般是因为两个信号位宽不一致导致的。
- **我正确实现了 ALU 中的有符号和无符号比较逻辑,但是 ALU 的计算仍然出现错误**。这种错误可能是由于,忘记让 ALU 在 slt 和 sltu 时执行减法。
- **我将一个值赋给一个信号,但是这个信号在波形图中显示为 xxxxxx**。这时需要检查,代码中是否同时将两个值赋给了同一个信号。

在多周期 CPU 设计中 next_pc 应该是时序逻辑吗？

我的多周期 CPU 是通过单周期 CPU 改造而来的。因此,我在模块设计中保留了 PC 负责更新,PC 控制器负责计算 next_pc 信号的逻辑。但是,我只将 PC 的更新逻辑与 FSM 联系了起来,而 PC 控制器则仍然是一个组合逻辑。这导致了在跳转和分支指令时,next_pc 往往不是当前周期所需要的 next_pc。我认为,在设计多周期 CPU 时,这种分开的设计并不是一个好设计,应当将 PC 的更新逻辑和 next_pc 的计算逻辑放在同一个模块中,统一受 FSM 状态的控制。

4 对讲义中思考题(如有)的理解和回答

ALUop 的编码有什么规律？表格中的 ALUop 编码是否还有优化空间？

ALUop 的值可以通过解码指令的特定位段来生成。

- **R-Type:**
  - func[3:2] 用于区分大的操作类别:00= 加减,01= 逻辑,10= 比较。
  - 在加减类 (func[3:2]==00) 中, func[1] 区分加 (0) 和减 (1)。生成规则: ALUop = {func[1], 2'b10}。

- 在逻辑类 (`func[3:2]==01`) 中,`func[1]` 和 `func[0]` 共同决定具体操作。生成规则: `ALUop = {func[1], 1'b0, func[0]}`。
- 在比较类 (`func[3:2]==10`) 中, `func[0]` 的反码 (`~func[0]`) 区分有符号 (0→1) 和无符号 (1→0)。生成规则: `ALUop = {~func[0], 2'b11}`。

- **I-Type:**
  - `opcode` 的特定位用于区分操作类别,并生成与 R-Type 相同的 `ALUop`。
  - 加法类 (`opcode[2:1]==00`): 直接映射到加法 `ALUop = 010`。
  - 逻辑类 (`opcode[2]==1`): 使用 `opcode[1]` 和 `opcode[0]` 生成 `ALUop`。生成规则: `ALUop = {opcode[1], 1'b0, opcode[0]}`。
  - 比较类 (`opcode[2:1]==01`): 使用 `opcode[0]` 的反码 (`~opcode[0]`) 生成 `ALUop`。生成规则: `ALUop = {~opcode[0], 2'b11}`。

我们不难发现, 除去移位器的操作, `opcode` 全零时就对应着寄存器类型的算术逻辑指令; 而 `opcode[5] == 0 && opcode[3] == 1` 便对应着所有的立即数类型的算术逻辑指令。此时, 寄存器类指令 `funct` 域的低三位和立即数类型的指令 `opcode` 的低三位有着极大程度上的相似性。因此可以共用一套译码逻辑。

5 实验所耗时间

在课后,你花费了大约 50 小时完成此次实验。

刚做完研讨课的第一个实验,随后便开始做这个实验。相较于第一次实验,本次实验的难度陡然上升,颇有一种弹射起步的感觉。经过本次实验,我成功的实现了单周期和多周期 MIPS CPU,并且在此过程中,学习到了如何将一个复杂的系统分解为多个简单的模块。通过对每个模块的设计和实现,我对 MIPS 指令集以及 CPU 的工作原理有了更深入的理解。

一开始,我并不知道从哪里下手。于是便和学长与同学进行讨论,认真地分析讲义上所给出的示例结构图中每个模块和信号的功能。在分析结构图的过程中,我逐渐对如何实现每个模块有了一些初步的想法。阅读手册是设计此 CPU 的重要一环。如果说 CPU 结构图可以让我们对 CPU 的设计有一个宏观的构想,那么指令集手册便将每个指令的实现细节事无巨细的告诉了我们。英文阅读自然是一个难题,但是如果不读手册只问 AI 的话,就会发现十个 AI 对于同一个指令都能给出十个不同的解释。因此对指令最准确的理解必须来自阅读手册。另外,译码表是设计 CPU 的关键抓手。通过观察译码表,可以发现同类指令的共性和细微差别,很有助于我们化简 CPU 设计中一些信号的逻辑表达式。利用好这些工具以后,我最大的感想就是,只要开始写第一行代码,你就离成功不远了!

debug 也是这个实验中重要的一环。我刚写完后的第一版 CPU 在三十纳秒的时候就出错了。在查错的过程中,我发现我最容易犯的就是位宽错误。此外,对指令的理解不准确也会造成错误。因此,在阅读指令集手册的时候就需要万分谨慎,不能有半点理解上的谬误。

总而言之, 亲手设计 CPU 的过程充满了挑战, 但成功之后也带来了很大的成就感。从无从下手到逐步理解结构图、指令集手册和译码表, 再到将思路转变成 RTL 代码, 这不仅是一次技术的实践, 更是一次解决复杂问题的思维训练。这次实验使得我对计算机底层的运作逻辑有了更深入的了解, 不再是纸上谈兵, 也让我对后续更复杂的实验充满了期待与好奇。