

MIPS 单周期处理器实验报告

张作柏

17300240035

2019 年 3 月 23 日

目录

	4	讨论	9
	4.1	显示寄存器选择	9
	4.2	时钟频率调节	9
	4.3	显示模块 Display	9
	5	注意事项	9
1	2	MIPS 指令集	
1.1	2	指令集分析	
1.1.1	2	R 型指令	
1.1.2	2	I 型指令	
1.1.3	2	J 型指令	
1.2	3	MIPS 汇编器实现	
2	3	部件分析	
2.1	3	多路复用器 MUX2	
2.2	3	加法器 Adder	
2.3	3	移位器 SL2	
2.4	3	位扩展 SignExtend	
2.5	3	触发器 flopr	
2.6	4	算术逻辑单元 ALU	
2.7	4	指令存储器 Imem	
2.8	4	数据存储器 Dmem	
2.9	4	寄存器文件 RegFile	
2.10	5	数据通路 Datapath	
2.11	6	控制模块 Controller	
3	7	测试样例与结果	
3.1	7	全部指令 all.in	
3.2	7	位运算测试 test_sign.in	
3.3	8	条件跳转 condition_beq.in	
3.4	8	switch 语句 switch.in	
3.5	8	while 循环 while_loop.in	
3.6	8	for 循环 for_loop.in	
3.7	8	快速乘法 quick_multiply.in	
3.8	8	递归测试 factorial.in	

1 MIPS 指令集

本节主要介绍 MIPS 指令集架构, 讨论 MIPS 指令的功能, 描述 MIPS 汇编器的实现细节。

1.1 指令集分析

本小节将简单介绍 MIPS 指令集的基本体系与实验中涉及的指令, 并给出相应的功能说明。具体的指令编码可见 2.11 节。

1.1.1 R 型指令

R 类型是寄存器类型的缩写。R 类型指令有 3 个寄存器操作数: 2 个为源操作数, 1 个为目的操作数。下图给出了 R 类型机器指令格式:

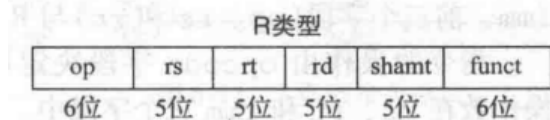


图 6-5 R 类型机器指令格式

32 位指令分为 6 个字段: op、rs、rt、rd、shamt 和 funct。指令的操作编码由 op 和 funct 决定, R 类型指令的 op 操作码都是 0, 故特定 R 类型操作由 funct 字段决定。指令的操作数编码包括 3 个字段: rs、rt 和 rd。前两个寄存器 rs 和 rt 是源寄存器, rd 是目的寄存器。shamt 字段仅仅用于移位操作, 其中数值表示移位的位数, 对于其他 R 类型指令, shamt 为 0。

在本次实验中, 我们实现的 R 类型指令包括:

- **nop**: 空指令, 不做任何操作。其 op 字段编码为全 0, 因此被归类于 R 类型指令, 等价于指令 `sll $r0, $r0, $r0`。
- **add, sub, and, or, slt**: 运算指令, 将寄存器 rs 与 rt 中的值运算后, 写入 rd 寄存器。五种运算分别为: $[rs] + [rt]$, $[rs] - [rt]$, $[rs] \& [rt]$, $[rs] | [rt]$, $([rs] < [rt]) ? 1 : 0$ 。
- **sll, sra, srl**: 移位指令, 将寄存器 rt 中的值 (左/右) 移 shamt 位后, 写入 rd 寄存器。默认 rs 字段为 0。三种移位分别为: 左移 (补 0), 算术右移 (补符号位), 逻辑右移 (补 0)。
- **jr**: 跳转寄存器指令, 跳至寄存器 rs 中的值所在的地址。默认 rt、rd 字段为 0。因为涉及寄存器 PC 的更改, 所以被归类于 R 类型指令。

后四条指令为补充指令!

1.1.2 I 型指令

I 类型是立即数类型的缩写。I 类型指令有两个寄存器操作数和一个立即数操作数。下图给出了 I 类型机器指令格式:

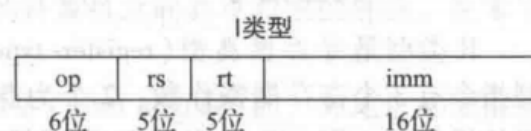


图 6-8 I 类型指令格式

一条 32 位指令有 4 个字段: op、rs、rt 和 imm。前三个字段与 R 类型指令一样。imm 字段代表一个 16 位立即数。指令的操作由 opcode 字段决定。操作数在 rs、rt 和 imm 三个字段中。rs 和 imm 常用于源操作数。rt 既可用于目的操作数, 也可用于源操作数。

在本次实验中, 我们实现的 I 类型指令包括:

- **addi, andi, ori, slti**: 运算指令, 将寄存器 rs 中的值与立即数 imm 运算后, 写入 rt 寄存器。四种运算分别为: $[rs] + imm$, $[rs] \& imm$, $[rs] | imm$, $([rs] < imm) ? 1 : 0$ 。
- **sw, lw**: 存取指令, 从内存某地址处读取 (向内存某地址处写入) 寄存器 rt 中的值。内存地址的计算方式为: $[rs] + imm$ 。
- **beq, bne**: 条件转移指令。当寄存器 rs 与 rt 中的值相等 (不等) 时, 跳至某地址处。跳转地址的计算方式为: $PC + 4 + (imm \ll 2)$ 。

1.1.3 J 型指令

J 类型是跳转类型的缩写。这种格式仅用于跳转指令。下图给出了 J 类型机器指令格式:

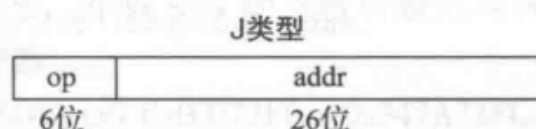


图 6-11 J 类型指令格式

这种指令格式有一个 26 位地址操作数 addr。与其他格式一样, J 类型指令由一个 6 位 opcode 开始, 决定操作类型。剩下的位用于指定地址 addr。

- **j**: 跳转指令，直接跳至某地址处。地址的计算方式为: $\{PC + 4_{31..28}, \text{addr}, 00\}$ 。

- **jal**: 跳转写入指令，跳至某地址处，并将当前 PC 值写入 \$ra 寄存器。地址的计算方式为: $\{PC + 4_{31..28}, \text{addr}, 00\}$ 。

jal 指令为补充指令！

1.2 MIPS 汇编器实现

为了更方便的书写测试代码，我特地使用 Python 写了一个 MIPS 指令的汇编器，将汇编代码翻译为机器代码。

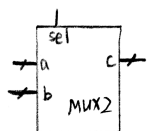
汇编器的设计思路如下：

1. 扫描整个文件，处理出 label 对应的地址。
2. 依次处理每条指令，因为指令长度固定，所以每次可以让 PC 值直接加四。
3. 将指令分为三种类型，三种类型分别对应三种译码方式。
4. 若为 R 类型或 I 类型指令，则利用字符串处理从中分离出寄存器，译码后放入对应位置。
5. 若为 I 类型或 J 类型指令，则从指令中提取立即数，注意立即数可以以 label 的形式给出。
6. 在计算地址时，需注意 I 类型指令为相对寻址（相对于 PC+4），J 类型指令则为直接寻址（高四位与 PC+4 相同，低两位为 0）。

最终的输出格式有两种：一种是.out 文件，便于阅读；另一种是.txt 文件，可用于直接写入 lmem。

2 部件分析

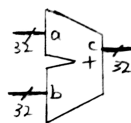
2.1 多路复用器 MUX2



功能说明：操作数长度可选的多路复用器。当 $\text{sel} = 0$ 时， $c = a$ ；当 $\text{sel} = 1$ 时， $c = b$ 。

实现细节：在模块时引入参数 parameter 定义操作数长度，利用 assign 语句搭配 $(\text{sel} == 0)?a:b$ 做选择。

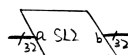
2.2 加法器 Adder



功能说明：32 位加法器， $c = a + b$ ，用于计算 PC 值与跳转地址。

实现细节：通过 assign 语句和 Verilog 自带的加法器实现。

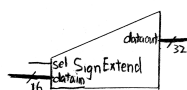
2.3 移位器 SL2



功能说明：将 32 位数左移 2 位， $b = a \ll 2$ ，用于立即数的字对齐。

实现细节：通过 assign 语句和拼接操作实现。

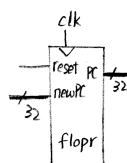
2.4 位扩展 SignExtend



功能说明：将 16 位数扩展为 32 位数。当 $\text{sel} = 0$ 时，进行符号扩展；当 $\text{sel} = 1$ 时，进行 0 扩展。用于扩展指令中的 16 位立即数。

实现细节：利用 assign 语句和拼接操作。

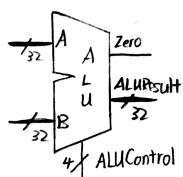
2.5 触发器 flopr



功能说明：32 位触发器。当 clk 到达上升沿时，将 newPC 的值写入 PC；当 $\text{reset} = 1$ 时，将 PC 清零。用于存储 PC。

实现细节：简单的 always 模块，reset 设置为同步清零端。

2.6 算术逻辑单元 ALU



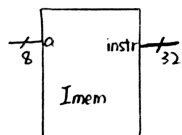
功能说明：根据 ALUControl 信号，进行运算，zero 信号为零标志。运算表如下：

ALUControl	ALUResult	ALUControl	ALUResult
0000	A & B	0101	A ~B
0001	A B	0110	A - B
0010	A + B	0111	(A<B)?1:0
0011	B << A	1000	B >> A (logic)
0100	A & ~B	1001	B >> A (arithmetic)

表 1: ALU 功能表

实现细节：使用 case 语句对 ALUControl 信号进行判断，然后用 Verilog 内置的运算单元进行运算。最后通过 assign 语句判断 ALUResult 是否为 0，设置 zero 信号。

2.7 指令存储器 Imem



功能说明：指令存储器。根据 8 位地址 a 读取指令 instr。

实现细节：定义 256 个长度为 8 的寄存器。¹ 注意定义时二维数组的两维不能搞反，前面一维表示寄存器的长度，后面一维表示寄存器的个数。

```
reg [7:0] RAM[255:0];
```

初始化时，利用 initial 将预先写好的机器编码写入 RAM 寄存器。写的时候，因为实际指令的长度为 32，所以我们需要一次写四个寄存器。

```
initial
begin
```

¹与课本中的写法不同，这里我们按照实际机器中每个寄存器对应一个字节 (8 bits) 的方式开数组

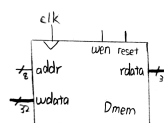
```
{RAM[3], RAM[2], RAM[1], RAM[0]} <= 32'h20080063;
{RAM[7], RAM[6], RAM[5], RAM[4]} <= 32'h20090025;
end
```

用 assign 语句读取 a 地址对应的指令编码，注意一次要读四个寄存器。

```
assign instr = {RAM[a+3], RAM[a+2], RAM[a+1], RAM[a]};
```

读要写成组合逻辑！

2.8 数据存储器 Dmem



功能说明：数据存储器。当 reset = 1 时，将内存中全部数据清零；否则，当 wen = 1 时，向 addr 地址写入 wdata。rdata 始终等于 addr 地址处的值。

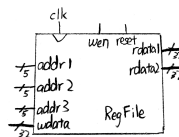
实现细节：原理同 Imem。当 reset 为 1 时，可利用 for 语句对 RAM 进行清零。

```
if (reset)
begin
for (j = 0; j < 32; j = j + 1)
RAM[j] <= 0;
end
```

写入时，同样要同时写四个寄存器。

```
if (wen) RAM[addr+3], RAM[addr+2], RAM[addr+1], RAM[
addr]} <= wdata;
```

2.9 寄存器文件 RegFile



功能说明：寄存器文件。从 addr1 与 addr2 两个寄存器同时读数据，结果放入 rdata1 和 rdata2 中；当 reset=1 时，将所有寄存器中的数据清零；否则，当 wen=1 时，向 addr3 寄存器写入数据 wdata。

实现细节：原理同 Dmem。要注意 0 号寄存器的值始终为 0，所以我们直接判断若 addr=0，则直接返回 0。

```
assign rdata1 = (addr1 != 0) ? rf[addr1] : 0;
assign rdata2 = (addr2 != 0) ? rf[addr2] : 0;
```

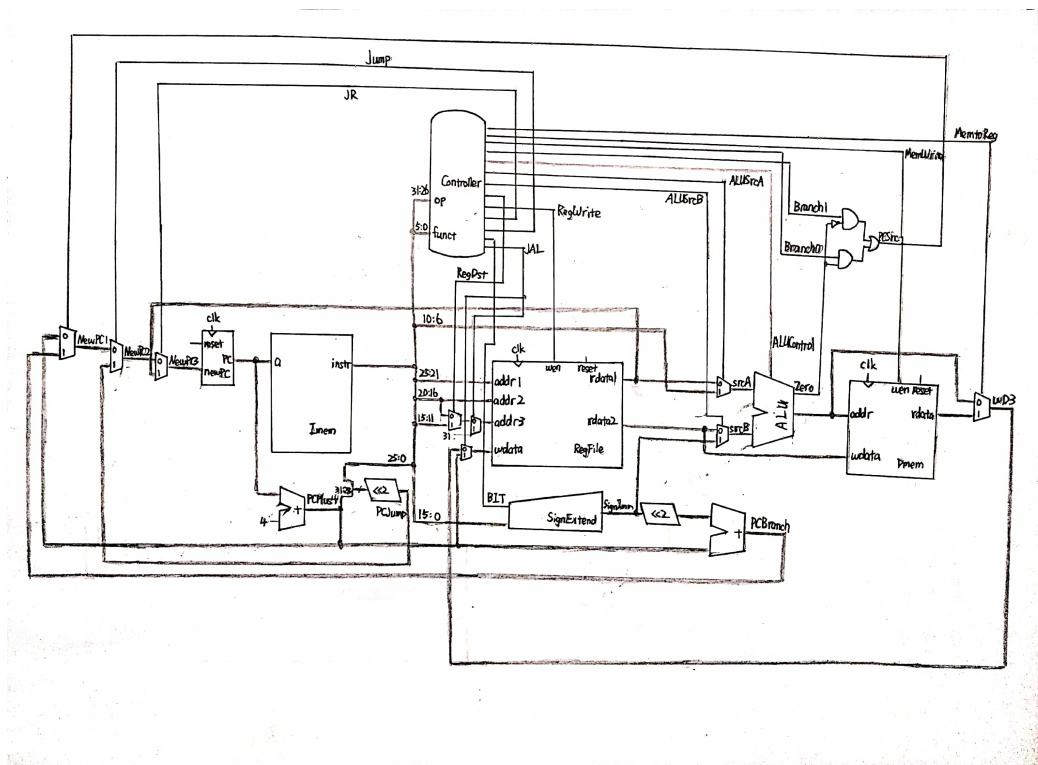


图 1: 数据通路 Datapath

2.10 数据通路 Datapath

整个数据通路连接如图 1 所示，因为新添加了几条指令，所以与课本中的通路略有不同。

功能说明：连接所有元件，传递信号。从左至右依次解释如下：

- 最左侧为三个二路选择器²，用于选择新的 PC 值。
 1. 用于分支指令 beq 和 bne, 选择信号为 PCSrc。当 PCSrc = 0 时，正常递增，选择 PC+4 的值；当 PCSrc = 1 时，选择分支地址 PCBranch。
 2. 用于直接跳转指令 j 和 jal, 选择信号为 Jump。当 Jump = 0 时，不直接跳转，选择 NewPC1；当 Jump = 1 时，选择直接跳转地址 PCJump。
 3. 用于寄存器跳转指令 jr, 选择信号为 JR。当 JR = 0 时，不发生寄存器跳转，选择 NewPC2；当 JR = 1 时，将从寄存器中读出的值 RD1 作为跳转地址。
- 因此，新的 PC 值共有四个来源：正常递增 PC+4，分支跳转 PCBranch，直接跳转 PCJump，寄存器跳转 RD1。
- 将 PC 值作为 Imem 的输入地址，读出的 instr 相应位传入 RegFile，读取相应的寄存器值。
- 左下方先用加法器计算 PC+4 的值，结果记为 PCPlus4；再将 PCPlus 的高 4 位与 instr 的后 26 位拼接，左移两位后，作为直接跳转的地址 PCJump。
- 寄存器文件的写使能由 RegWrite 信号控制，写入的寄存器有三种情况：当指令为 R 类型时，RegDst = 1，选择指令中的 rd 字段作为目的寄存器；当指令为 I 类型时，RegDst = 0，选择指令中的 rt 字段作为目的寄存器；当指令为 jal 时，目的寄存器为 \$ra。两个多路器分别由信号 RegDst、JAL 控制。写入数据有三种选择：当指令为 lw 时，MemtoReg = 1，选择从内存中读取的值；当指令为 jal 时，JAL = 1，将 PCPlus4 的值写入寄存器 \$ra；否则，选择运算结果 ALUResult 作为写入数据。
- 中间下方由扩展器，移位器和加法器组成。扩展器可根据 BIT 信号选择符号扩展或按 0 扩展。左移两位后与 PCPlus4 相加，结果作为相对寻址的地址 PCBranch。
- ALU 的功能选择信号为四位信号 ALUControl，两个运算数分别为 SrcA 与 SrcB。SrcA 可选择寄存

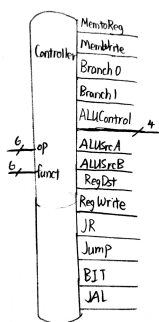
²因为指令是一条一条添加的，所以多路器也是一个一个添加的，也可以仅用一个多路器实现。

器文件 RD1 的值,也可以是指令中 shamt 字段的值,后者用于移位操作。SrcB 可选择寄存器文件 RD2 的值,也可以是立即数扩展后的值,后者主要用于 I 类型指令。两个多路器的选择信号分别为 ALUSrcA 与 ALUSrcB。

- Dmem 始终将 ALUResult 作为读写的地址, RD2 作为写入的数据,写使能由信号 MemWrite 控制。
- 当指令为 beq 且零标志为 1 或指令为 bne 且零标志为 0 时,分支指令的信号 PCSrc = 1,进行分支跳转。

实现细节:我在实现时并未单独开一个 datapath 模块,而是选择直接在顶层模块中连接电路。为了便于检查元件位置与数量,我在书写代码的时候是按照图中从左至右的顺序进行书写的。

2.11 控制模块 Controller



功能说明:根据指令码分配控制信号,详见表 2。为减小表格面积,表头均使用缩写,信号全称依次为 RegWrite、RegDst、ALUSrcA、ALUSrcB、Branch0、Branch1、MemWrite、MemtoReg、JR、Jump、JAL、BIT、aluop。

- 对于需要修改寄存器的指令, RegWrite = 1
- 对于 R 类型指令, RegDst = 1; 对于 I 类型指令, RegDst = 0
- 对于移位操作, ALUSrc = 1
- 对于部分需要立即数计算的 I 类型指令, ALUSrcB = 1
- 对于 beq 指令, Branch0 = 1; 对于 bne 指令, Branch1 = 1
- 对于需要写内存的 sw 指令, MemWrite = 1

- 对于需要将内存读出的值写入寄存器的 lw 指令, MemtoReg = 1
- 对于 jr 指令, JR = 1; 对于 jal 指令, JAL = 1
- 对于直接跳转的 j 和 jal 指令, J = 1
- 对于需要补 0 的 I 类型位运算, BIT = 1
- 对于 R 类型指令, aluop = 010, 具体操作由 funct 决定; 对于部分 I 类型指令, 选择相应的运算; 对于 beq 和 bne 指令, 需要做减法, 设置 aluop = 001

实现细节:为了避免冗杂的赋值语句,我们可以先将控制信号压成一个向量,然后直接用向量赋值即可。

```
assign {RegWrite, RegDst, ALUSrcB, Branch0, Branch1,
       MemWrite, MemtoReg, Jump, aluop, BIT} = controls
;
always @(*)
    case (op)
        6'b000000: controls <= 12'b110000000100;
        6'b001000: controls <= 12'b101000000000;
        6'b001100: controls <= 12'b101000000111;
        6'b001101: controls <= 12'b101000001001;
        6'b001010: controls <= 12'b101000001010;
        6'b101011: controls <= 12'b001001000000;
        6'b100011: controls <= 12'b101000100000;
        6'b000010: controls <= 12'b00000001xxxx0;
        6'b000100: controls <= 12'b000100000010;
        6'b000101: controls <= 12'b000010000010;
        6'b000011: controls <= 12'b10000001xxxx0;
        default: controls <= 12'bxxxxxxxxxxxx;
    endcase
```

对于一些特殊的信号,只有极少数情况下为 1,则可直接特判得到。

```
assign ALUSrcA = (op == 6'b000000) & ((funct == 6'
    b000000) | (funct == 6'b000011) | (funct == 6'
    b000010));
assign JR = (op == 6'b000000) & (funct == 6'b001000);
assign JAL = (op == 6'b000011);
```

对于 R 类型指令通过 funct 字段,选择 ALUControl 的信号;否则,根据 op 字段,分配 aluop 信号,借此选择 ALUControl 信号。

```
always @(*)
    case (aluop)
        3'b000: alucontrol <= 3'b010;
        3'b001: alucontrol <= 3'b110;
        3'b011: alucontrol <= 3'b000;
        3'b100: alucontrol <= 3'b001;
        3'b101: alucontrol <= 3'b111;
```

opcode	funct	RW	RD	SA	SB	B0	B1	MW	MR	JR	J	JAL	BIT	aluop
000000	100000	add	1	1	0	0	0	0	0	0	0	0	0	010
000000	100010	sub	1	1	0	0	0	0	0	0	0	0	0	010
000000	100100	and	1	1	0	0	0	0	0	0	0	0	0	010
000000	100101	or	1	1	0	0	0	0	0	0	0	0	0	010
000000	101010	slt	1	1	0	0	0	0	0	0	0	0	0	010
001000		addi	1	0	0	1	0	0	0	0	0	0	0	000
001100		andi	1	0	0	1	0	0	0	0	0	0	1	011
001101		ori	1	0	0	1	0	0	0	0	0	0	1	100
001010		slti	1	0	0	1	0	0	0	0	0	0	0	101
101011		sw	0	x	0	1	0	0	1	0	0	0	0	000
100011		lw	1	0	0	1	0	0	0	1	0	0	0	000
000010		j	0	x	0	x	0	0	0	0	1	1	0	xxx
000000		nop	0	x	0	x	0	0	0	0	0	0	0	xxx
000100		beq	0	x	0	0	1	0	0	0	0	0	0	001
000101		bne	0	x	0	0	0	1	0	0	0	0	0	001
000000	000000	sll	1	1	1	0	0	0	0	0	0	0	0	010
000000	000011	sra	1	1	1	0	0	0	0	0	0	0	0	010
000000	000010	srl	1	1	1	0	0	0	0	0	0	0	0	010
000000	001000	jr	1	1	0	0	0	0	0	1	0	0	0	010
000011		jal	1	x	0	0	0	0	0	0	1	1	0	xxx

表 2: 控制信号

```

default: case(funct)
    6'b100000: alucontrol <= 4'b0010;
    6'b100010: alucontrol <= 4'b0110;
    6'b100100: alucontrol <= 4'b0000;
    6'b100101: alucontrol <= 4'b0001;
    6'b101010: alucontrol <= 4'b0111;
    6'b000000: alucontrol <= 4'b0011;
    6'b000010: alucontrol <= 4'b1000;
    6'b000011: alucontrol <= 4'b1001;
    default: alucontrol <= 4'bxxxx;
endcase
endcase

```

```

0x14 : sub $t0, $s2, $s3 | 02534022
0x18 : and $t0, $s3, $s1 | 02714024
0x1c : or $t0, $s1, $s2 | 02324025
0x20 : slt $t0, $s0, $s2 | 0212402a
0x24 : sw $s0, 2($0) | ac100002
0x28 : lw $t0, 2($0) | 8c080002

```

3.2 位运算测试 test_sign.in

```

0x0 : addi $s0, $0, 0xffff | 2010ffff
0x4 : add $s0, $s0, $s0 | 02108020
0x8 : andi $t0, $s0, 0xffff | 3208ffff

```

3 测试样例与结果

3.1 全部指令 all.in

```

0x0 : addi $s0, $0, 12 | 2010000c
0x4 : andi $s2, $s0, -8 | 3212ffff8
0x8 : ori $s3, $s1, 10 | 3633000a
0xc : slti $s4, $s2, 5 | 2a540005
0x10 : nop | 00000000
0x10 : add $t0, $s0, $s1 | 02114020

```

```

0x0 : addi $s0, $0, 4 | 20100004
0x4 : addi $s1, $0, 1 | 20110001
0x8 : addi $s1, $s1, 3 | 22310003
0xc : beq $s0, $s1, target | 12300002
0x10 : addi $s1, $s1, 1 | 22310001
0x14 : sub $s1, $s1, $s0 | 02308822
0x18 : | 
0x18 : target: | 

```

```
0x18 : add $s1, $s1, $s0 | 02308820
```

3.4 switch 语句 switch.in

```
0x0 : addi $s0, $0, 100 | 20100064
0x4 : | 
0x4 : case20: | 
0x4 : addi $t0, $0, 20 | 20080014
0x8 : bne $s0, $t0, case50 | 15100002
0xc : addi $s1, $0, 2 | 20110002
0x10 : j done | 0800000e
0x14 : case50: | 
0x14 : addi $t0, $0, 50 | 20080032
0x18 : bne $s0, $t0, case100 | 15100002
0x1c : addi $s1, $0, 3 | 20110003
0x20 : j done | 0800000e
0x24 : case100: | 
0x24 : addi $t0, $0, 100 | 20080064
0x28 : bne $s0, $t0, default | 15100002
0x2c : addi $s1, $0, 5 | 20110005
0x30 : j done | 0800000e
0x34 : default: | 
0x34 : add $s1, $0, $0 | 00008820
0x38 : | 
0x38 : done: |
```

3.5 while 循环 while_loop.in

```
0x0 : addi $s0, $0, 1 | 20100001
0x4 : addi $s1, $0, 0 | 20110000
0x8 : | 
0x8 : addi $t0, $0, 128 | 20080080
0xc : while: | 
0xc : beq $s0, $t0, done | 11100003
0x10 : add $s0, $s0, $s0 | 02108020
0x14 : addi $s1, $s1, 1 | 22310001
0x18 : j while | 08000003
0x1c : done: |
```

3.6 for 循环 for_loop.in

```
0x0 : add $s1, $0, $0 | 00008820
0x4 : addi $s0, $0, 0 | 20100000
0x8 : addi $t0, $0, 10 | 2008000a
0xc : | 
0xc : for: | 
0xc : beq $s0, $t0, done | 11100003
0x10 : add $s1, $s1, $s0 | 02308820
0x14 : addi $s0, $s0, 1 | 22100001
0x18 : j for | 08000003
0x1c : | 
0x1c : done: |
```

3.7 快速乘法 quick_multiply.in

```
0x0 : addi $t0, $0, 99 | 20080063
0x4 : addi $t1, $0, 37 | 20090025
0x8 : addi $s0, $0, 0 | 20100000
0xc : | 
0xc : while: | 
0xc : beq $t1, $0, done | 10090006
0x10 : andi $t2, $t1, 1 | 312a0001
0x14 : beq $t2, $0, target | 100a0001
0x18 : add $s0, $s0, $t0 | 02088020
0x1c : target: | 
0x1c : sll $t0, $t0, 1 | 00084040
0x20 : srl $t1, $t1, 1 | 00094842
0x24 : j while | 08000003
0x28 : | 
0x28 : done: |
```

3.8 递归测试 factorial.in

```
0x0 : addi $sp, $0, 128 | 201d0080
0x4 : addi $a0, $0, 5 | 20040005
0x8 : jal factorial | 0c000004
0xc : sll $v0, $v0, 1 | 00021040
0x10 : | 
0x10 : factorial: | 
0x10 : addi $sp, $sp, -8 | 23bdf0f8
0x14 : sw $a0, 4($sp) | afa40004
0x18 : sw $ra, 0($sp) | afbf0000
0x1c : addi $t0, $0, 2 | 20080002
0x20 : slt $t0, $a0, $t0 | 0088402a
0x24 : beq $t0, $0, else | 10080003
0x28 : addi $v0, $0, 1 | 20020001
0x2c : addi $sp, $sp, 8 | 23bd0008
0x30 : jr $ra | 03e00008
0x34 : else: | 
0x34 : addi $a0, $a0, -1 | 2084ffff
0x38 : jal factorial | 0c000004
0x3c : lw $ra, 0($sp) | 8fbf0000
0x40 : lw $a0, 4($sp) | 8fa40004
0x44 : addi $sp, $sp, 8 | 23bd0008
0x48 : add $v0, $a0, $v0 | 00821020
0x4c : jr $ra | 03e00008
```


4 讨论

4.1 显示寄存器选择

4.2 时钟频率调节

4.3 显示模块 Display

5 注意事项

1. Run Implementation 前必须先写好 xdc 文件，并且顶层模块要有输出口。
2. always 模块中只能对 reg 类型的变量赋值。
3. 读逻辑要写成组合逻辑，否则写入时不能同步内存中的值。