

《数字逻辑与处理器基础》处理器第二次大作业

1. MIPS 单周期 CPU 实现：

a) 控制器模块设计：

b) Instruction	PCSrc [1:0]	Branch	RegWrite	RegDst [1:0]	MemRead	MemWrite	MemtoReg [1:0]	ALUSrc1	ALUSrc2	ExtOp	LuOp
lw	0	0	1	0	1	0	1	0	1	1	0
sw	0	0	0	x	x	1	x	0	1	1	0
lui	0	0	1	0	x	0	0	0	1	x	1
add	0	0	1	1	x	0	0	0	0	x	x
addu	0	0	1	1	x	0	0	0	0	x	x
sub	0	0	1	1	x	0	0	0	0	x	x
subu	0	0	1	1	x	0	0	0	0	x	x
addi	0	0	1	0	x	0	0	0	1	1	0
addiu	0	0	1	0	x	0	0	0	1	1	0
and	0	0	1	1	x	0	0	0	0	x	x
or	0	0	1	1	x	0	0	0	0	x	x
xor	0	0	1	1	x	0	0	0	0	x	x
nor	0	0	1	1	x	0	0	0	0	x	x
andi	0	0	1	0	x	0	0	0	1	0	0
sll	0	0	1	1	x	0	0	1	0	x	x
srl	0	0	1	1	x	0	0	1	0	x	x
sra	0	0	1	1	x	0	0	1	0	x	x
slt	0	0	1	1	x	0	0	0	0	x	x
sltu	0	0	1	1	x	0	0	0	0	x	x
slti	0	0	1	0	x	0	0	0	1	1	0
sltiu	0	0	1	0	x	0	0	0	1	1	0
beq	zero	1	0	x	x	0	x	0	0	1	x
j	2	0	0	x	x	0	x	x	x	x	x
jal	2	0	1	2	x	0	2	x	x	x	x
jr	3	0	0	x	x	0	x	x	x	x	x
jalr	3	0	1	1	x	0	2	x	x	x	x

控制信号设置:

PCSrc [1:0]: 00 PC+4

01 beq 指令

10 jal 和 j 地址

11 jr 地址(从寄存器读)

Branch: 0 不是分支指令

1 是分支指令

RegWrite: 0 不写入寄存器

1 写入寄存器

RegDst [1:0]: 00 rt

01 rd jalr 指令也用这个

10 ra jal 指令用

MemRead: 0 不读 MEM

1 读 MEM

MemWrite: 0 不写 MEM

1 写 MEM

MemtoReg [1:0]: 00 ALU 运算结果

01 MEM 读取数据

10 PC+4, 用于 jalr (写入 rd) 和 jal (写入 ra)

ALUSrc1: 0 rs

1 移位指令的无符号扩展 shamt

ALUSrc2: 0 rt

1 ImmExt

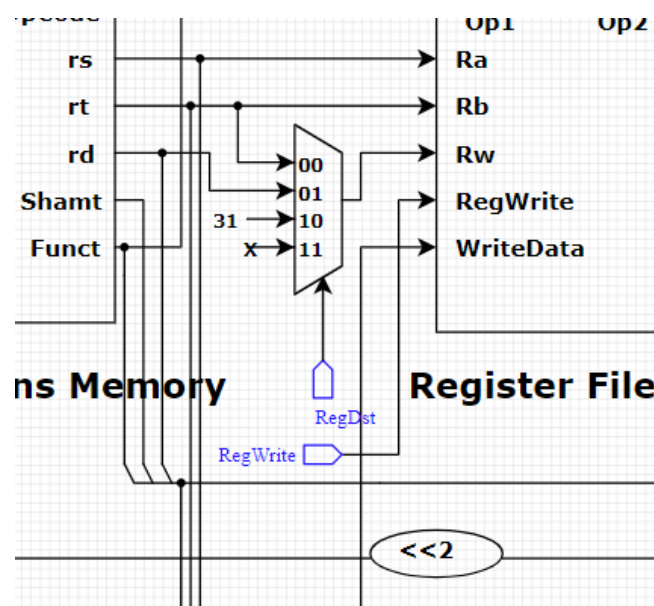
ExtOp: 0 无符号扩展

1 有符号扩展

LuOp: 0 不是 Lui

1 是 Lui, 要把 16 位立即数放入目标寄存器 rt 高 16 位

c) 数据通路设计: 整体数据通路见附件“单周期处理器数据通路.pdf”。



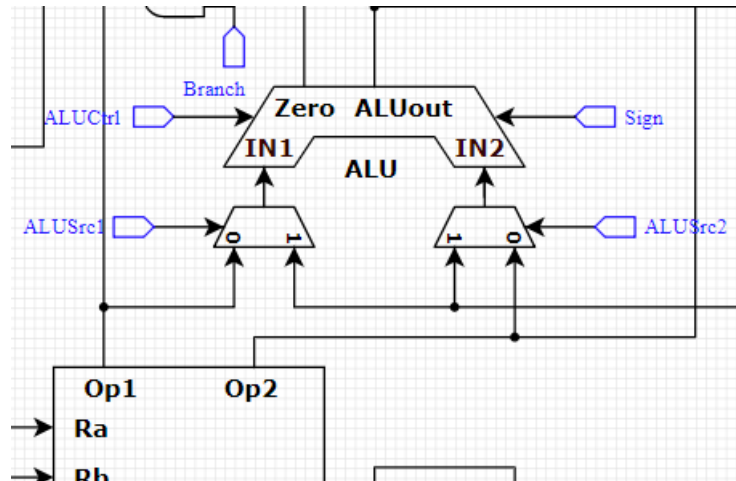
MUX1

```

61 //Register File
62 case(RegDst)
63     2'b00: Rw=rt;
64     2'b01: Rw=rd;
65     2'b10: Rw=5'd31;
66 endcase

```

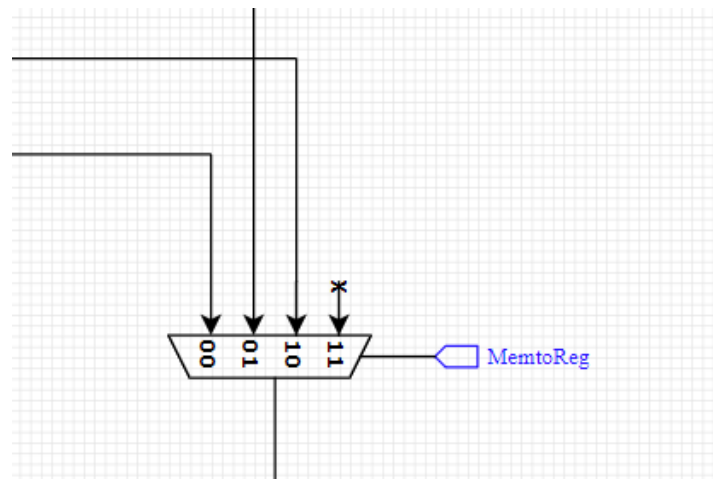
由 RegDst 控制，控制写回的寄存器编号，\$rt,\$rd 或者 \$ra



MUX2 (左), MUX3 (右)

MUX2 由 ALUSrc1 控制，选择 ALU 的输入 1 是 rs 寄存器的值还是无符号扩展的 Shamt

MUX3 由 ALUSrc2 控制，选择 ALU 的输入 2 是 rt 寄存器的值还是扩展后的立即数



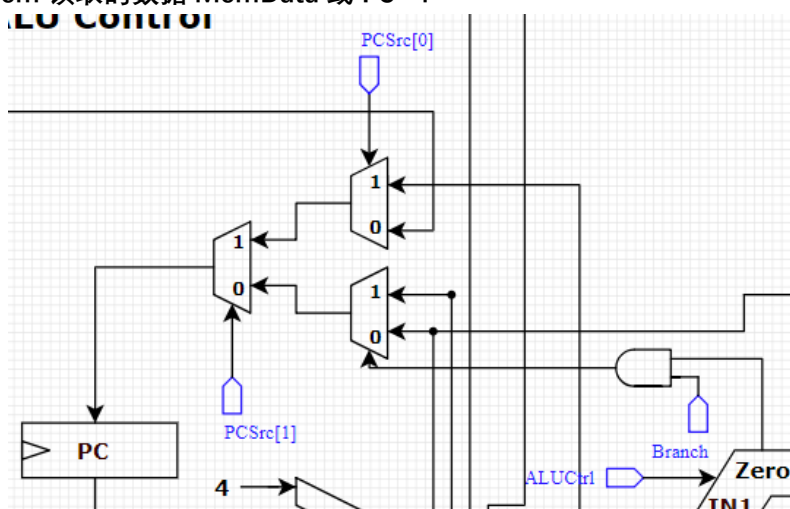
MUX4

```

67 case(MemtoReg)
68     2'b00: WriteData_RF=ALUOut;
69     2'b01: WriteData_RF=MemData;
70     2'b10: WriteData_RF=PC+32'd4;
71 endcase

```

MUX4 由 MemtoReg 控制，控制写回寄存器堆的数据来源：ALU 运算的结果
ALUout, Mem 读取的数据 MemData 或 PC+4



从左到右，从上到下 MUX5, MUX6, MUX7

```

81  always@(posedge reset or posedge clk)
82  begin
83    if(reset)
84      PC<=32'b0;
85    else begin
86      case(PCSrc[1])
87        1'b0:begin
88          case(zero&Branch)
89            1'b0:PC<=PC+32'd4;
90            1'b1:PC<=PC+32'd4+{ExtOut<<2};
91          endcase
92        end
93        1'b1:begin
94          case(PCSrc[0])
95            1'b0:begin PC<=PC+32'd4;
96            PC<={PC[31:28],{rs,rt,rd,Shamt,Funct}<<2};
97            end
98            1'b1:PC<=Op1;
99          endcase
100        end
101      endcase
102    end
103  end

```

MUX6 由 PCSrc[0]控制，控制输入 MUX5 的数据：PC+4 高四位和 J 型指令 26 位左移后拼接成的地址或寄存器储存的值。

MUX7 由 Branch and Zero 控制，控制输入 MUX5 的数据：PC+4 或 PC+4+SignExt (Imm) <<2。

MUX5 由 PCSrc[1]控制，控制写入 PC 寄存器的数据：分支成立的地址和正常 PC+4，或者 jal, j 和 jr

三个 MUX 实现了对 PC 寄存器 4 种更新值的选择。

d) 汇编程序分析-1:

i.

0 a0 存入 $(0+12123)_{10}=0x00002f5b$

1 a1 存入 $(0-12345)_{10}=0xffffcfc7$

2 a2 存入 $0xffffcfc7<<16=0xcfc70000$

3 a3 存入\$signed(0xcfc70000)>>16=0xffffcfc7

4 \$a3=\$a1,因此跳转到 L1 标签

L1:

6 t0 存入\$a2+\$a0=0xcfc70000+0x00002f5b=0xcfc72f5b

7 t1 存入\$signed(0xcfc72f5b)>>8=0xffcfc72f

8 t2 存入(0-12123)₁₀=0xffffd0a5

9 由于\$a0 存正值, \$t2 存负值, 因此\$v0 写入 0

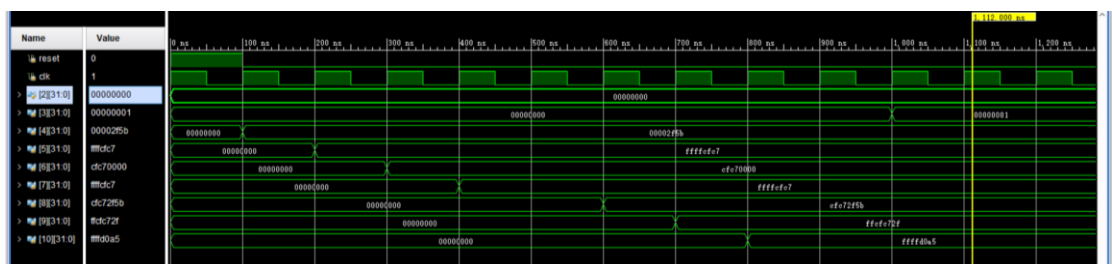
10 无符号数比较\$t2 最高位比\$a0 最高位大, 因此\$v0 写入 1

Loop:

11 执行到这个语句后就一直在这里循环

因此寄存器\$a0 到\$a3 分别是 0x00002f5b、0xffffcfc7、0xcfc70000、0xffffcfc7, \$t0 到 t2 分别是 0xcfc72f5b、0xffcfc72f、0xffffd0a5, \$v0、\$v1 分别是 0x00000000 和 0x00000001

ii.



可以看出\$a0~\$a3,\$t0~\$t2,\$v0,\$v1 的终值都符合计算结果, 各寄存器值的变化顺序符合预测结果,可以验证单周期处理器的正确性。

2. MIPS 多周期 CPU 实现:

a) 多周期状态机控制器:

见附件状态转移图.pdf

b) 多周期 CPU 的 ALU 控制逻辑与功能实现:

ALU.v 设计与单周期的 ALU 完全相同, 区别在于 ALU 控制信号在同一条指令执行期间要随着状态的改变而发生改变。

```
254     always @(*)
255     begin
256         case(state)
257             sIF:ALUOp<=4'b0;
258             sID:ALUOp<=4'b0;
259             3'd2:begin case(OpCode)
260                 6'h0:ALUOp<=4'b0011;// R-type
261                 6'h09:ALUOp<=4'b0; //addiu
262                 6'h08,6'h23,6'h2b:ALUOp<=4'b0100; //addi, lw, sw
263                 6'h0a:ALUOp<=4'b0101; //slti
264                 6'h0b:ALUOp<=4'b0001; //sltiu
265                 6'h0c:ALUOp<=4'b0010; //andi
266                 6'h04:ALUOp<=4'b0110; //beq
267             endcase
268         end
269     endcase
270 end
```

在 Controller.v 中根据状态转移决定 ALUOp, IF 阶段 ALU 用来计算 PC+4, ID 阶段计算可能用到的分支地址, 两个阶段都用到 ALU 的加法功能;

剩下只有执行 & 分支/跳转完成阶段需要使用 ALU, 非 R 型指令可以由 OpCode 唯一确定需要的 ALU 功能, R 型指令需要在 ALUControl.v 中进一步根据 Funct 决定 ALUConf 的值

```
41 //R-type
42 4'b0011:begin
43     case(Funct)
44         //add
45         6'h20:begin
46             ALUConf=5'b00000;
47             Sign=1'b1;
48             end
49         //addu
50         6'h21:begin
51             ALUConf=5'b00000;
52             Sign=1'b0;
53             end
54         //sub
55         6'h22:begin
56             ALUConf=5'b00001;
57             Sign=1'b1;
58             end
59         //subu
60         6'h23:begin
61             ALUConf=5'b00001;
62             Sign=1'b0;
63             end
64         //and
65         6'h24:begin
66             ALUConf=5'b00010;
67             Sign=1'b1;
68             end
69         //or
70         6'h25:begin
71             ALUConf=5'b00011;
72             Sign=1'b1;
73             end
74     end
75 end
```

```
74 //xor
75 6'h26:begin
76     ALUConf=5'b00100;
77     Sign=1'b1;
78     end
79 //nor
80 6'h27:begin
81     ALUConf=5'b00101;
82     Sign=1'b1;
83     end
84 //sll
85 6'h00:begin
86     ALUConf=5'b00110;
87     Sign=1'b0;
88     end
89 //srl
90 6'h02:begin
91     ALUConf=5'b00111;
92     Sign=1'b0;
93     end
94 //sra
95 6'h03:begin
96     ALUConf=5'b01000;
97     Sign=1'b1;
98     end
99 //slt
100 6'h2a:begin
101     ALUConf=5'b01001;
102     Sign=1'b1;
103     end
104 //sltu
105 6'h2b:begin
106     ALUConf=5'b01001;
107     Sign=1'b0;
108     end
```

R 型指令根据 Funct 决定 ALUConf

```

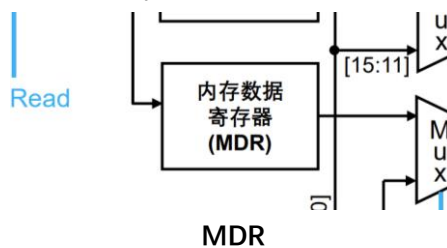
116 //addi, lw, sw
117 4'b0100:begin
118     ALUConf=5'b0;
119     Sign=1'b1;
120 end
121 //slti
122 4'b0101:begin
123     ALUConf=5'b01001;
124     Sign=1'b1;
125 end
126 //sltiu
127 4'b0001:begin
128     ALUConf=5'b01001;
129     Sign=1'b0;
130 end
131 //andi
132 4'b0010:begin
133     ALUConf=5'b00010;
134     Sign=1'b0;
135 end
136 //beq
137 4'b0110:begin
138     ALUConf=5'b00001;
139     Sign=1'b0;
140 end
141
35 case(ALUOp)
36 //addiu
37 4'h0:begin
38     ALUConf=5'b00000;
39     Sign=1'b0;
40 end
41 //R_type

```

非 R 型指令根据 ALUOp 可以唯一确定 ALUConf

c) 数据通路设计:

除去 PC 寄存器和 RF 中的寄存器,额外需要四个寄存器



```

90 //IR,MDR
91 assign Instruction=Mem_data;
92 assign MDR_i=Mem_data;

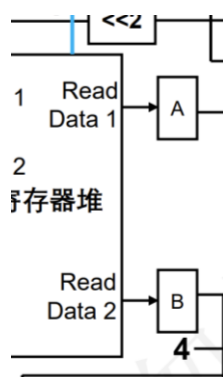
```

```

112 case(MemtoReg)
113 2'b00: Write_data_r=MDR_o;
114 2'b01: Write_data_r=Result;

```

储存上一周期内存读出的数据,连到选择寄存器堆 WriteData 的 MUX 上,使得内存数据可以写回寄存器



寄存器 A,B

```

117 //ALU
118 case(ALUSrcA)
119 2'b00: In1=PC_o;
120 2'b01: In1=A_o;
121 2'b10: In1={27'b0,Shamt};

```

```

101 //PC
102 case(PCSource[1])
103 1'b0:PC_i=(PCSource[0]==1'b1)?ALU_o:Result;
104 1'b1:PC_i=(PCSource[0]==1'b1)?A_o:{PC_o[31:28],{rs,rt,Imm}<<2};
105 endcase

```

寄存器 A 输入 RF 的 ReadData1,输出连接到选择 ALU 输入 1 的 MUX 以及选择更新 PC 数据来源的 MUX 上

```

87 //Mem
88 assign Address=(IorD==1'b0)?PC_o:ALU_o;
89 assign Write_data_m=B_o;

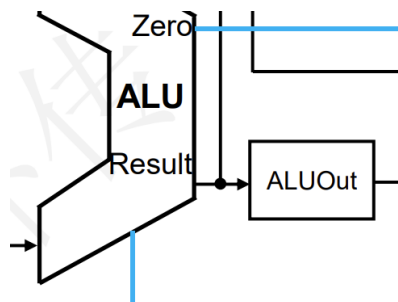
```

```

123 case(ALUSrcB)
124 2'b00: In2=B_o;
125 2'b01: In2=32'd4;
126 2'b10: In2=ImmExtOut;
127 2'b11: In2=ImmExtShift;
128 endcase
129 end

```

寄存器 B 输入 RF 的 ReadData2,输出连接到选择 ALU 输入 2 的 MUX 以及 Mem 的 WriteData 上



ALUOut

```

87 //Mem
88 assign Address=(IorD==1'b0)?PC_o:ALU_o;
89 assign Write_data_m=B_o;

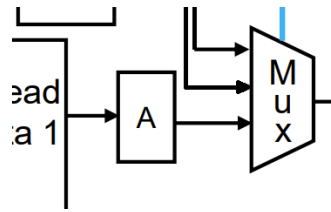
```

```

101 //PC
102 case(PCSource[1])
103 1'b0:PC_i=(PCSource[0]==1'b1)?ALU_o:Result;
104 1'b1:PC_i=(PCSource[0]==1'b1)?A_o:{PC_o[31:28],{rs,rt,Imm}<<2};
105 endcase

```

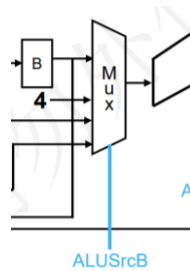
由于 ALU 运算结果产生和需要 ALU 结果在相邻的周期,因此需要一个寄存器来储存上一个周期 ALU 运算的结果,ALUOut 的输出连接到选择 Mem 地址的 MUX 和更新 PC 的 MUX 上



ALUSrcA 控制的 MUX

```
118 case(ALUSrcA)
119 2'b00: In1=PC_o;
120 2'b01: In1=A_o;
121 2'b10: In1={27'b0,Shamt};
```

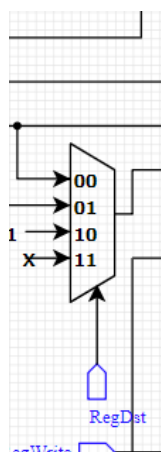
从 PC 寄存器输出,寄存器 A 输出,扩展的位移量 Shamt 中选择 ALU 输入 1



ALUSrcB 控制的 MUX

```
123 case(ALUSrcB)
124 2'b00: In2=B_o;
125 2'b01: In2=32'd4;
126 2'b10: In2=ImmExtOut;
127 2'b11: In2=ImmExtShift;
128 endcase
```

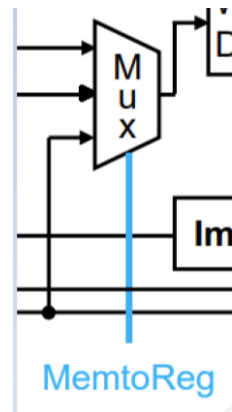
从寄存器 B 输出,32'd4,扩展的 Imm 和左移两位的扩展后 Imm 中选择 ALU 输入 2



RegDst 控制的 MUX

```
107 case(RegDst)
108 2'b00: Write_register=rt;
109 2'b01: Write_register=rd;
110 2'b10: Write_register=5'd31;
111 endcase
```

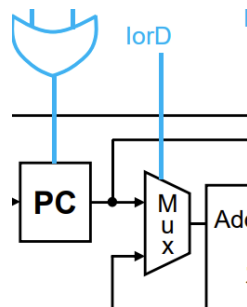
控制写回的寄存器编号是\$rt,\$rd 或者\$ra



MemtoReg 控制的 MUX

```
112 case(MemtoReg)
113   2'b00: Write_data_r=MDR_o;
114   2'b01: Write_data_r=Result;
115   2'b10: Write_data_r=PC_o; //PC已经更新成PC+4了
116 endcase
```

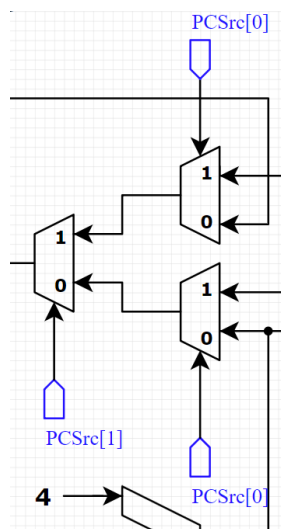
控制写入 RF 的数据是上一周期读取的内存数据,ALU 计算结果或 PC+4



IorD 控制的 MUX

```
87 //Mem
88 assign Address=(IorD==1'b0)?PC_o:ALU_o;
89 assign Write_data_m=B_o;
```

选择 Mem 的地址来源,是指令地址还是数据地址



PCSrc 控制的 MUX

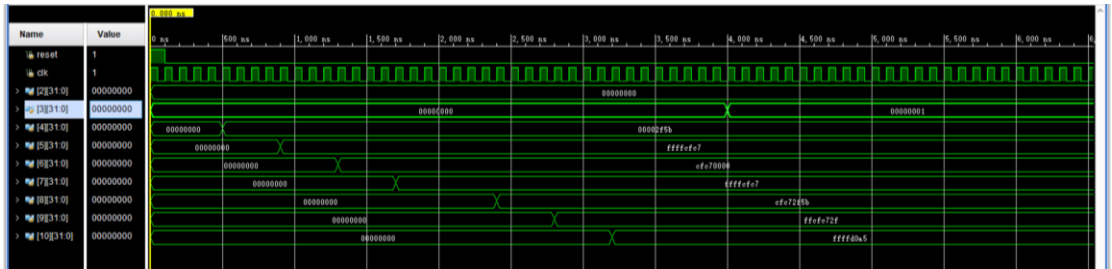
```

102 case(PCSource[1])
103 1'b0:PC_i=(PCSource[0]==1'b1)?ALU_o:Result;
104 1'b1:PC_i=(PCSource[0]==1'b1)?A_o:{PC_o[31:28],{rs,rt,Imm}<<2};
105 endcase

```

从 PC+4,分支地址,寄存器储存地址和 jump 地址中选择 PC 更新值

d) 功能验证 (汇编程序分析-1):



可以看出 \$a0~\$a3,\$t0~\$t2,\$v0,\$v1 的终值都符合计算结果,且变化过程也和预测的一致,可以验证多周期处理器的正确性。

3. MIPS 单周期和多周期 CPU 的性能对比:

a) 汇编程序分析-2:

i

- 0 addi \$a0, \$zero, 5 # \$a0 赋初值 5
- 1 xor \$v0, \$zero, \$zero # \$v0 赋初值 0
- 2 jal sum # 跳转到 sum 标签并将 3 号指令地址存入 \$ra 中

Loop:

- 3 beq \$zero, \$zero, Loop # 死循环作为程序的结束

sum:

- 4 addi \$sp, \$sp, -8 # 开辟两条指令大小的栈空间
- 5 sw \$ra, 4(\$sp) # 存入 jal 指令写入 \$ra 的地址
- 6 sw \$a0, 0(\$sp) # 存入 \$a0 的值
- 7 slti \$t0, \$a0, 1 # \$a0 值每次减 1, 在 \$a0 减小到 0 时 \$t0 为 1, 此时下一条分支条件不成立
- 8 beq \$t0, \$zero, L1 # 结合上一条指令控制入栈次数, 保证入栈的数据是 n, n-1, n-2...1
- 9 addi \$sp, \$sp, 8 # 栈指针上移两个数据位, 这条指令只用一次, 跳过 \$a0 等于零时存入的 \$a0 值和地址, 因为加 0 不改变 \$v0 的值
- 10 jr \$ra # 只使用一次, 对正整数 n 来说 \$a0 减小到 0 一定是从 L1 跳回 sum, 因此 \$ra 存 L1 号指令地址

L1:

- 11 add \$v0, \$a0, \$v0 # 入栈过程累加 \$a0 (从 n 加到 1)
- 12 addi \$a0, \$a0, -1 # \$a0 每次减 1, 用于下次累加

- 13 jal sum #跳到 sum, 继续入栈
- 14 lw \$a0, 0(\$sp) # 取出栈里的 a0
- 15 lw \$ra, 4(\$sp) #从栈里取出之前存的 jal 地址, 出栈过程在 14-18 循环, 栈顶存的不是 14 号指令地址, 是最初的 3 号指令地址, 之后进入死循环
- 16 addi \$sp, \$sp, 8 #栈指针上移两个 4 字节
- 17 add \$v0, \$a0, \$v0 #继续累加 a0, 出栈过程从 1 加到 n
- 18 jr \$ra #跳回 14 号指令, 最后一次跳到 3 号指令

功能 递归实现 $n*(n+1)$ 的求解, \$a0 作为参数输入, 程序结束恢复\$a0 的值

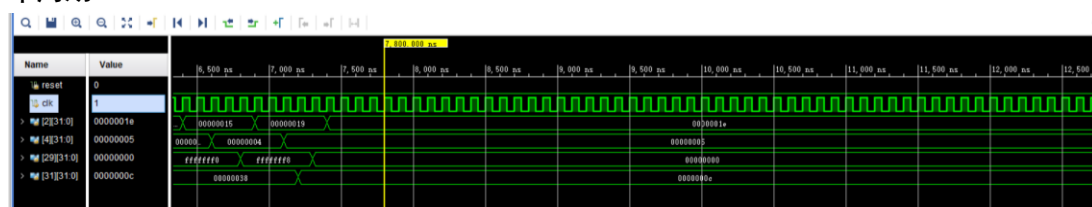
Loop 以死循环作为程序的结束, sum 实现入栈过程和出栈过程时跳过存\$a0=0 的数据以及地址的 8byte, L1 实现入栈时 n 到 1 的累加和出栈时 1 到 n 的累加。

ii.

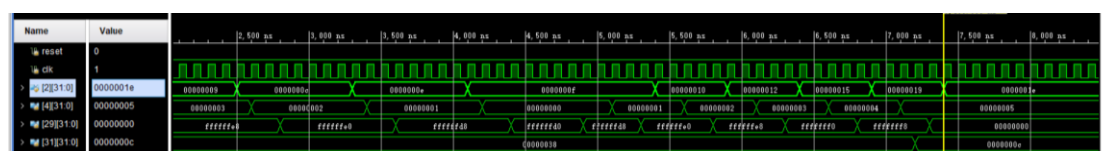
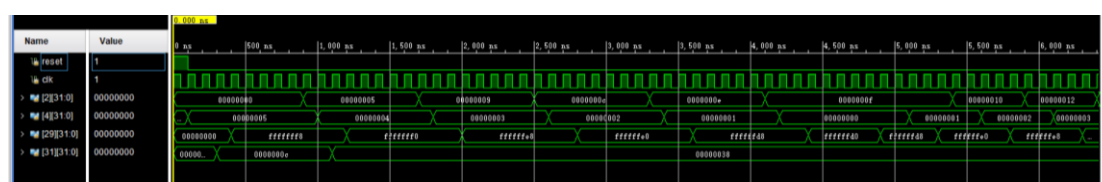
机器码见附件 InstructionMemory-2.v 和 InstAndDataMemory-2.v

iii.

单周期 CPU

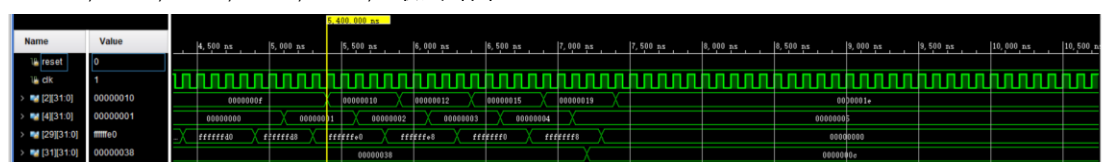


可见程序运行足够长时间后 $v0=0x1e=(30)_{10}$, $a0=0x05=(5)_{10}$



观察\$a0 值的变化, 初值为 0x0, 赋值为 0x5 后, 变化为 0x4, 0x3, 0x2, 0x1, 0x0, 之后依次变化为 0x1, 0x2, 0x3, 0x4, 0x5 后不再变化, 和预测结果一致。

\$v0 值的变化, 初值为 0x0, 入栈阶段依次加上 0x5, 0x4, 0x3, 0x2, 0x1, 出栈阶段依次加上 0x1, 0x2, 0x3, 0x4, 0x5, 最终结果 0x1e

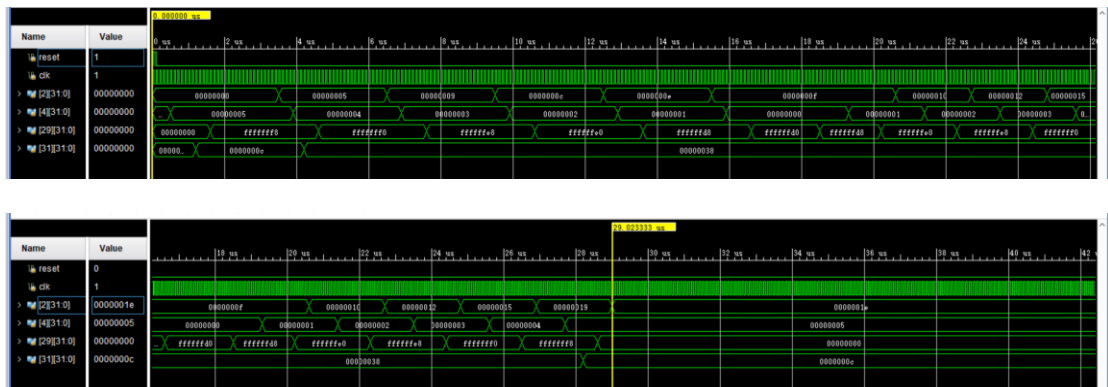


\$sp 值的变化, 初值 0x0, 图示时刻前\$sp 每次-8, 执行到指令 9 \$sp 的值加 8, 之后进行出栈过程, 栈指针依次+8 至恢复原数值(0x0)

\$ra 值的变化, 初值 0x0, 执行到指令 2 将指令 3 的地址(0x0c)存入\$ra,之后每次执行指令 13 都要写入指令 14 的地址(0x38), 出栈时\$ra 取得一直是 0x38, 直到栈顶取到指令 3 地址(0x03), 之后保持不变。

和预测的指令功能以及寄存器数值变化完全一致。

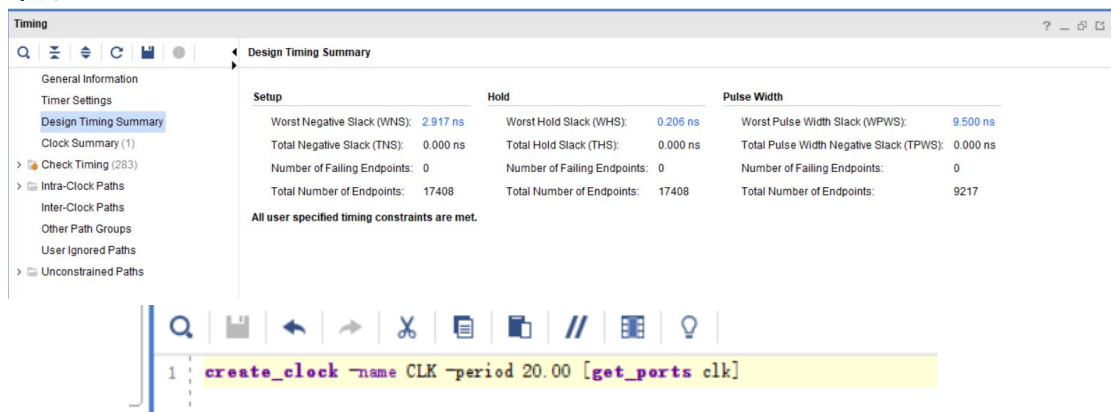
多周期 CPU



四个寄存器数值的变化和单周期处理器完全一致，可以验证预测的正确性。

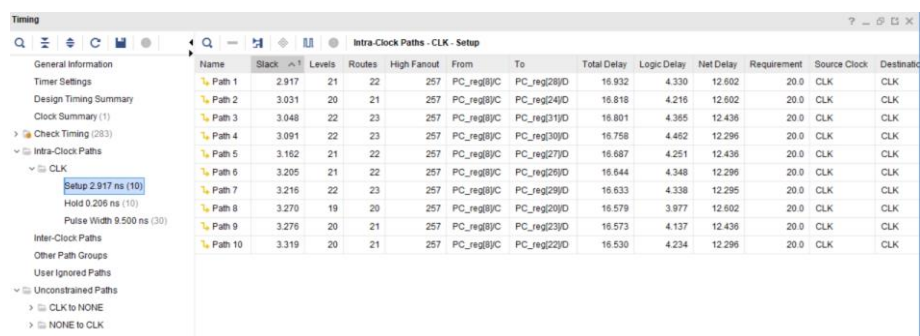
b) 资源与性能对比：

单周期 CPU



约束文件中时钟周期设置为 20ns，因此最小时钟周期 = $20n + (-2.917)n =$

$17.083ns$, 最高时钟频率 $f_{max} = \frac{1}{17.083n} = 58.538MHz$



单次计算最低延时16.530ns

Tcl Console	Messages	Log	Reports	Design Runs	Utilization
» LUT as Logic					
Hierarchy			Name	Used	
Summary			▼ CPU	5067	
▼ Slice Logic			DM (DataMemory)	2641	
▼ Slice LUTs (24%)			RF (RegisterFile)	1626	
LUT as Logic (24%)			alu (ALU)	565	
▼ Slice Registers (22%)			Leaf Cells (165)	165	
Register as Latch (<1%)			IM (InstructionMemory)	32	
Register as Flip Flop (22%)			Ctrl (Control)	25	
F8 Muxes (4%)			aluctri (ALUController)	13	
F7 Muxes (8%)					
Memory					
DSP					

CPU 占用 5067 个查找表

Tcl Console	Messages	Log	Reports	Design Runs	Utilization
» Slice Registers					
Hierarchy			Name	Used	
Summary			▼ CPU	9272	
▼ Slice Logic			DM (DataMemory)	8192	
▼ Slice LUTs (24%)			RF (RegisterFile)	992	
LUT as Logic (24%)			Leaf Cells (69)	69	
▼ Slice Registers (22%)			Ctrl (Control)	14	
Register as Latch (<1%)			aluctri (ALUController)	5	
Register as Flip Flop (22%)					
F8 Muxes (4%)					
F7 Muxes (8%)					
Memory					

CPU 占用 9272 个寄存器

多周期 CPU:

Tcl Console

Messages

Log

Reports

Design Runs

Timing

Q

⌕

⌕

⌕

⌕

⌕

⌕

⌕

⌕

⌕

⌕

⌕

Design Timing Summary

General Information

Timer Settings

Design Timing Summary

Clock Summary (1)

Check Timing (258)

Intra-Clock Paths

Inter-Clock Paths

Other Path Groups

User Ignored Paths

Unconstrained Paths

Setup

Hold

Pulse Width

Worst Negative Slack (WNS): 9.278 ns

Total Negative Slack (TNS): 0.000 ns

Number of Failing Endpoints: 0

Total Number of Endpoints: 18663

Worst Hold Slack (WHS): 0.161 ns

Total Hold Slack (THS): 0.000 ns

Number of Failing Endpoints: 0

Total Number of Endpoints: 18663

Worst Pulse Width Slack (WPWS): 9.500 ns

Total Pulse Width Negative Slack (TPWS): 0.000 ns

Number of Failing Endpoints: 0

Total Number of Endpoints: 9402

All user specified timing constraints are met.

Q

⌕

⌕

⌕

⌕

⌕

⌕

⌕

⌕

⌕

⌕

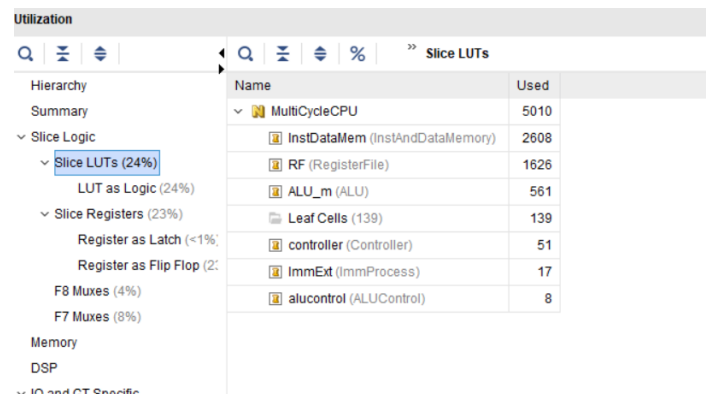
```
1 create_clock -name CLK -period 20.00 [get_ports clk]
```

约束文件时钟周期设置为 20ns，因此最小时钟周期 = $20n + (-9.278)n = 10.722ns$,

$$\text{最高时钟频率 } f_{\max} = \frac{1}{10.722n} = 93.266\text{MHz}$$

Tcl Console Messages Log Reports Design Runs Timing x											
Intra-Clock Paths - CLK - Setup											
General Information											
Timer Settings											
Design Timing Summary											
Clock Summary (1)											
Check Timing (258)											
Intra-Clock Paths											
CLK											
Setup 9.278 ns (10)											
Hold 0.161 ns (10)											
Pulse Width 9.500 ns (30)											
Inter-Clock Paths											
Other Path Groups											
Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source C
Path 1	9.278	12	13	33	controllerLuOs_regIC	pcPC_o_reg09ICE	10.340	2.109	8.231	20.0	CLK
Path 2	9.278	12	13	33	controllerLuOs_regIC	pcPC_o_reg10ICE	10.340	2.109	8.231	20.0	CLK
Path 3	9.278	12	13	33	controllerLuOs_regIC	pcPC_o_reg11ICE	10.340	2.109	8.231	20.0	CLK
Path 4	9.278	12	13	33	controllerLuOs_regIC	pcPC_o_reg12ICE	10.340	2.109	8.231	20.0	CLK
Path 5	9.278	12	13	33	controllerLuOs_regIC	pcPC_o_reg13ICE	10.340	2.109	8.231	20.0	CLK
Path 6	9.278	12	13	33	controllerLuOs_regIC	pcPC_o_reg14ICE	10.340	2.109	8.231	20.0	CLK
Path 7	9.278	12	13	33	controllerLuOs_regIC	pcPC_o_reg15ICE	10.340	2.109	8.231	20.0	CLK
Path 8	9.278	12	13	33	controllerLuOs_regIC	pcPC_o_reg16ICE	10.340	2.109	8.231	20.0	CLK
Path 9	9.278	12	13	33	controllerLuOs_regIC	pcPC_o_reg17ICE	10.340	2.109	8.231	20.0	CLK
Path 10	9.278	12	13	33	controllerLuOs_regIC	pcPC_o_reg18ICE	10.340	2.109	8.231	20.0	CLK

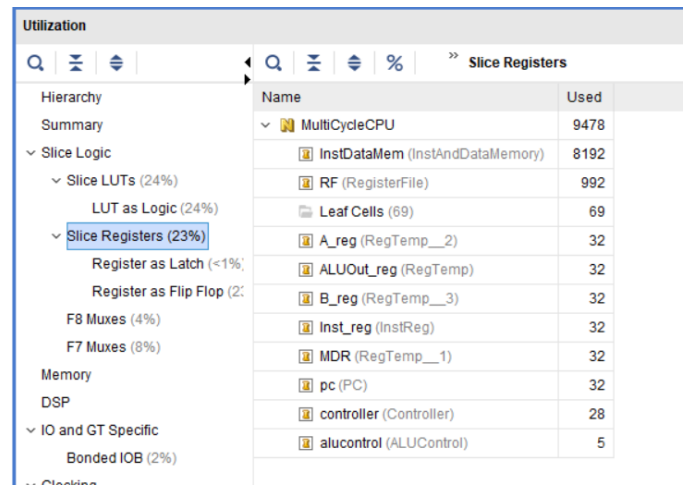
单次计算最低延时10.340ns



The screenshot shows the 'Utilization' window with the 'Slice LUTs' tab selected. The left sidebar shows a hierarchy with 'Slice LUTs (24%)' highlighted. The main table lists the usage of various LUTs.

Name	Used
MultiCycleCPU	5010
InstDataMem (InstAndDataMemory)	2608
RF (RegisterFile)	1626
ALU_m (ALU)	561
Leaf Cells (139)	139
controller (Controller)	51
ImmExt (ImmProcess)	17
alucontrol (ALUControl)	8

MultiCycleCPU 占用 5010 个查找表



The screenshot shows the 'Utilization' window with the 'Slice Registers' tab selected. The left sidebar shows a hierarchy with 'Slice Registers (23%)' highlighted. The main table lists the usage of various registers.

Name	Used
MultiCycleCPU	9478
InstDataMem (InstAndDataMemory)	8192
RF (RegisterFile)	992
Leaf Cells (69)	69
A_reg (RegTemp__2)	32
ALUOut_reg (RegTemp)	32
B_reg (RegTemp__3)	32
Inst_reg (InstReg)	32
MDR (RegTemp__1)	32
pc (PC)	32
controller (Controller)	28
alucontrol (ALUControl)	5

MultiCycleCPU 占用 9478 个寄存器

对比单周期和多周期 CPU，两者逻辑资源占用基本一致，多周期 CPU 使用的寄存器数量更多，多周期 CPU 的最高时钟频率接近单周期 CPU 最高时钟频率的两倍，单个周期内延时也明显小于单周期 CPU，但多周期 CPU 一条指令至少需要两个时钟周期，因此执行一条指令的时间仍然长于单周期 CPU。