同济大学计算机系

计算机组成原理实验报告

学　　号 2154312

姓　　名 郑博远

专　　业 计算机科学与技术

授课老师 陈永生

一、实验内容

1. 实验介绍

在本次实验中，我们将使用 Verilog HDL 语言实现 31 条 MIPS 指令的单周期 CPU 的设计和仿真，最后在 XILINX NEXYS4 DDR 开发板上运行。

2. 实验目标

1. 深入了解 CPU 的原理。
2. 画出实现 31 条指令的 CPU 的通路图。
3. 学习使用 Verilog HDL 语言设计实现 31 条指令的 CPU。

3. 实验原理

Mars 中 CPU 采用冯诺依曼结构，指令存储起始地址为 0x00400000，数据存储起始地址为 0x10010000。由于我们设计的 CPU 为哈佛结构，指令和数据的起始地址都为 0，需在 PC、指令存储以及数据存储模块进行地址映射。

31 条 MIPS 指令如下表所示：

Mnemonic Symbol	Format						Sample
Bit #	31..26	25..21	20..16	15..11	10..6	5..0	
R-type	op	rs	rt	rd	shamt	func	
add	000000	rs	rt	rd	0	100000	add $1,$2,$3
addu	000000	rs	rt	rd	0	100001	addu $1,$2,$3
sub	000000	rs	rt	rd	0	100010	sub $1,$2,$3
subu	000000	rs	rt	rd	0	100011	subu $1,$2,$3
and	000000	rs	rt	rd	0	100100	and $1,$2,$3
or	000000	rs	rt	rd	0	100101	or $1,$2,$3
xor	000000	rs	rt	rd	0	100110	xor $1,$2,$3
nor	000000	rs	rt	rd	0	100111	nor $1,$2,$3
slt	000000	rs	rt	rd	0	101010	slt $1,$2,$3
sltu	000000	rs	rt	rd	0	101011	sltu $1,$2,$3
sll	000000	0	rt	rd	shamt	000000	sll $1,$2,10
srl	000000	0	rt	rd	shamt	000010	srl $1,$2,10
sra	000000	0	rt	rd	shamt	000011	sra $1,$2,10
sllv	000000	rs	rt	rd	0	000100	sllv $1,$2,$3
srlv	000000	rs	rt	rd	0	000110	srlv $1,$2,$3
srav	000000	rs	rt	rd	0	000111	srav $1,$2,$3
jr	000000	rs	0	0	0	001000	jr $31

Bit #	31..26	25..21	20..16	15..0	
I-type	op	rs	rt	immediate	
addi	001000	rs	rt	Immediate(- ~ +)	addi $1,$2,100
addiu	001001	rs	rt	Immediate(- ~ +)	addiu $1,$2,100
andi	001100	rs	rt	Immediate(0 ~ +)	andi $1,$2,10
ori	001101	rs	rt	Immediate(0 ~ +)	andi $1,$2,10
xori	001110	rs	rt	Immediate(0 ~ +)	andi $1,$2,10
lw	100011	rs	rt	Immediate(- ~ +)	lw $1,10($2)
sw	101011	rs	rt	Immediate(- ~ +)	sw $1,10($2)
beq	000100	rs	rt	Immediate(- ~ +)	beq $1,$2,10
bne	000101	rs	rt	Immediate(- ~ +)	bne $1,$2,10
slti	001010	rs	rt	Immediate(- ~ +)	slti $1,$2,10
sltiu	001011	rs	rt	Immediate(- ~ +)	sltiu $1,$2,10
lui	001111	00000	rt	Immediate(- ~ +)	Lui $1, 10

Bit #	31..26	25..0	
J-type	op	Index	
j	000010	address	j 10000
jal	000011	address	jal 10000

二、各指令数据通路图

1. add

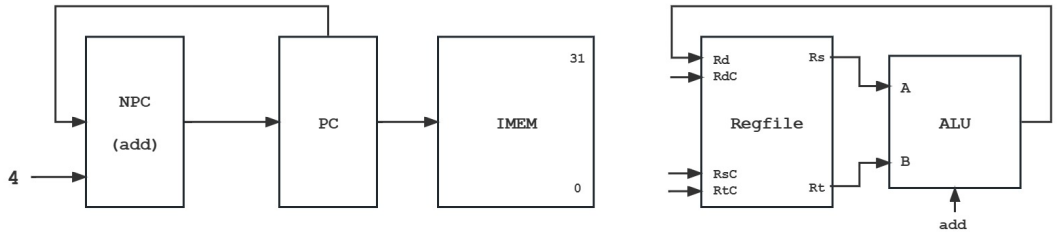
操作：取指令，rd←rs+rt, PC←NPC(PC+4)

所需部件：PC, NPC, IMEM, Regfile, ALU

输入输出关系：

add	PC	NPC	IMEM	Regfile		ALU	
				Rd	Rdc	A	B
	NPC	PC	PC	ALU	15-11	Rs	Rt

add



2. addu

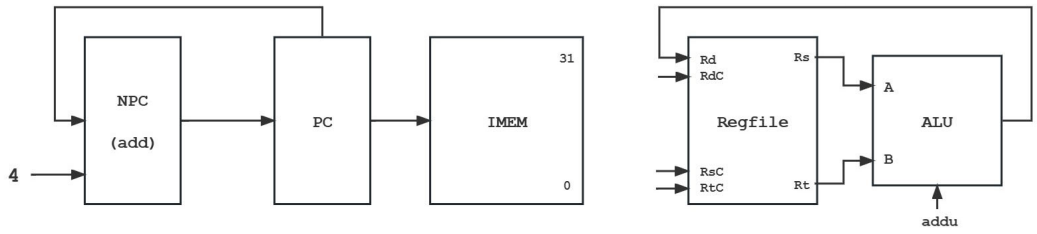
操作：取指令，rd←rs+rt, PC←NPC(PC+4)

所需部件：PC, NPC, IMEM, Regfile, ALU

输入输出关系：

addu	PC	NPC	IMEM	Regfile		ALU	
				Rd	Rdc	A	B
	NPC	PC	PC	ALU	15-11	Rs	Rt

addu



3. sub

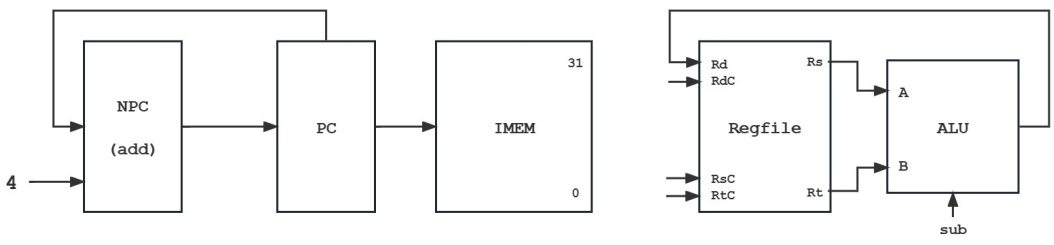
操作：取指令，rd←rs-rt, PC←NPC(PC+4)

所需部件：PC, NPC, IMEM, Regfile, ALU

输入输出关系：

sub	PC	NPC	IMEM	Regfile		ALU	
				Rd	Rdc	A	B
	NPC	PC	PC	ALU	15-11	Rs	Rt

sub



4. subu

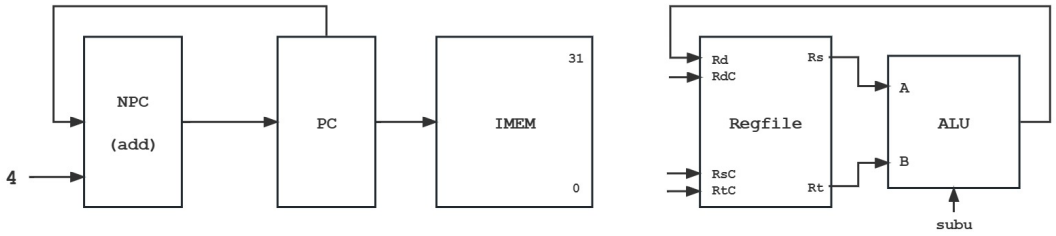
操作：取指令，rd←rs-rt, PC←NPC(PC+4)

所需部件：PC, NPC, IMEM, Regfile, ALU

输入输出关系：

subu	PC	NPC	IMEM	Regfile		ALU	
				Rd	Rdc	A	B
	NPC	PC	PC	ALU	15-11	Rs	Rt

subu



5. and

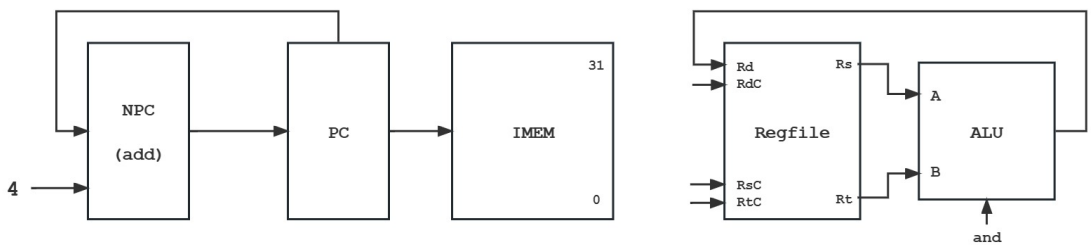
操作：取指令，rd←rs&rt, PC←NPC(PC+4)

所需部件：PC, NPC, IMEM, Regfile, ALU

输入输出关系：

and	PC	NPC	IMEM	Regfile		ALU	
				Rd	Rdc	A	B
	NPC	PC	PC	ALU	15-11	Rs	Rt

and



6. or

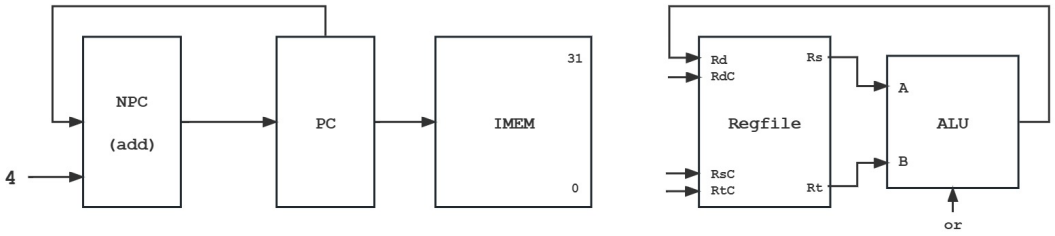
操作：取指令，rd←rs|rt, PC←NPC(PC+4)

所需部件：PC, NPC, IMEM, Regfile, ALU

输入输出关系：

or	PC	NPC	IMEM	Regfile		ALU	
				Rd	Rdc	A	B
	NPC	PC	PC	ALU	15-11	Rs	Rt

or



7. xor

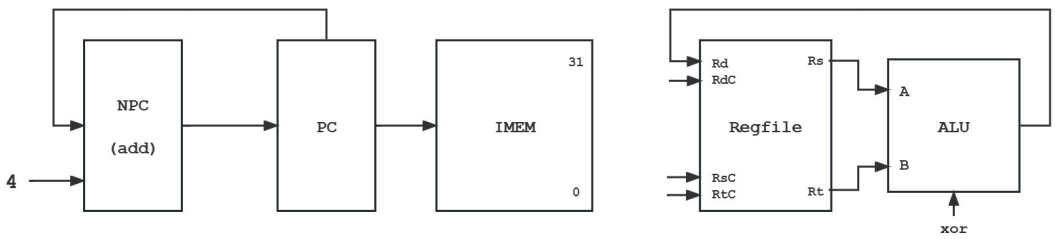
操作：取指令，rd←rs^rt, PC←NPC(PC+4)

所需部件：PC, NPC, IMEM, Regfile, ALU

输入输出关系：

xor	PC	NPC	IMEM	Regfile		ALU	
				Rd	Rdc	A	B
	NPC	PC	PC	ALU	15-11	Rs	Rt

xor



8. nor

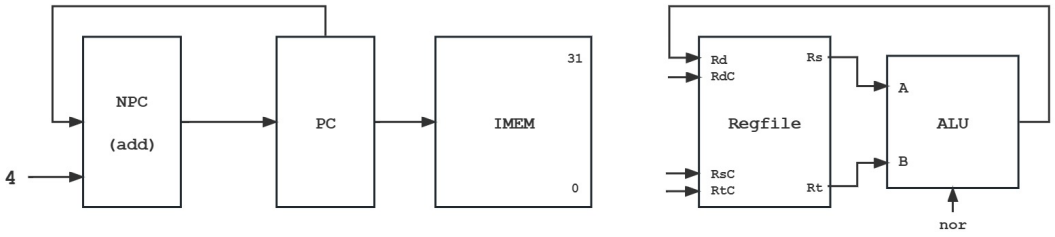
操作：取指令，rd←~(rs|rt), PC←NPC(PC+4)

所需部件：PC, NPC, IMEM, Regfile, ALU

输入输出关系：

nor	PC	NPC	IMEM	Regfile		ALU	
				Rd	Rdc	A	B
	NPC	PC	PC	ALU	15-11	Rs	Rt

nor



9. slt

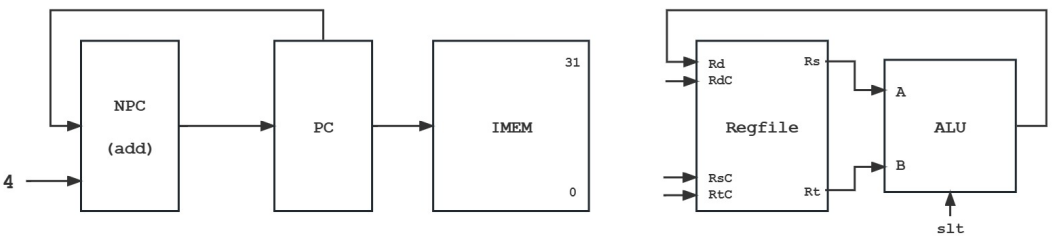
操作：取指令，rd←rs<rt, PC←NPC(PC+4)

所需部件：PC, NPC, IMEM, Regfile, ALU

输入输出关系：

slt	PC	NPC	IMEM	Regfile		ALU	
				Rd	Rdc	A	B
	NPC	PC	PC	ALU	15-11	Rs	Rt

slt



10. sltu

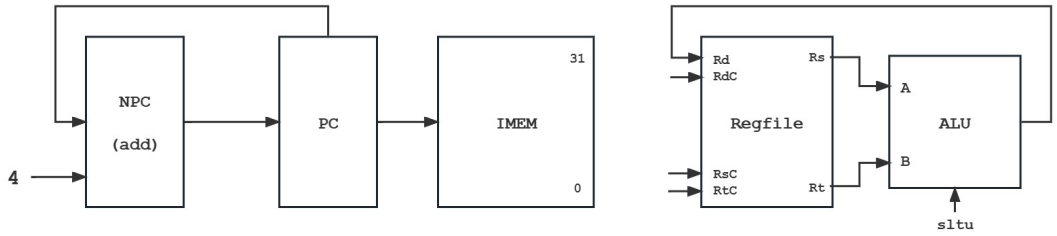
操作：取指令，rd←rs<rt, PC←NPC(PC+4)

所需部件：PC, NPC, IMEM, Regfile, ALU

输入输出关系：

sltu	PC	NPC	IMEM	Regfile		ALU	
				Rd	Rdc	A	B
	NPC	PC	PC	ALU	15-11	Rs	Rt

sltu



11. sll

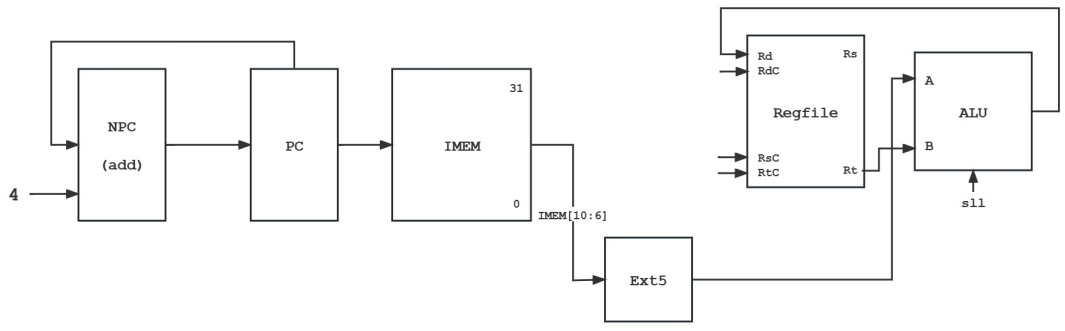
操作：取指令, rd←rt << sa , PC←NPC(PC+4)

所需部件：PC, NPC, IMEM, Regfile, ALU, Ext5

输入输出关系：

sll	PC	NPC	IMEM	Regfile		ALU		Ext5
				Rd	Rdc	A	B	
	NPC	PC	PC	ALU	15-11	Ext5	Rt	Sa

sll



12. srl

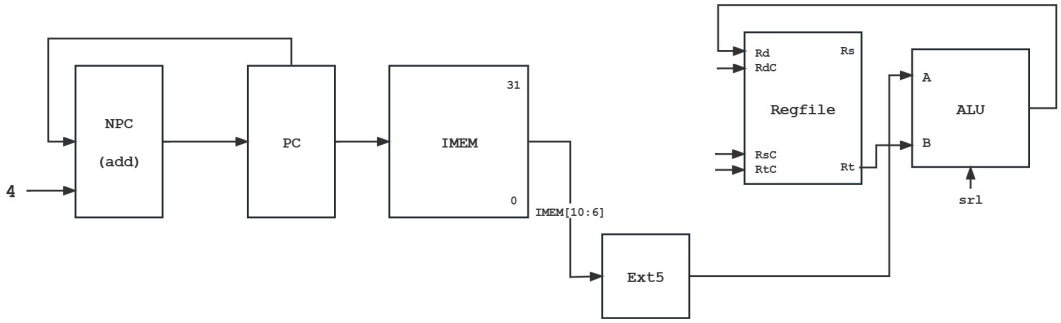
操作：取指令, rd←rt >> sa , PC←NPC(PC+4)

所需部件：PC, NPC, IMEM, Regfile, ALU, Ext5

输入输出关系：

srl	PC	NPC	IMEM	Regfile		ALU		Ext5
				Rd	Rdc	A	B	
	NPC	PC	PC	ALU	15-11	Ext5	Rt	Sa

srl



13. sra

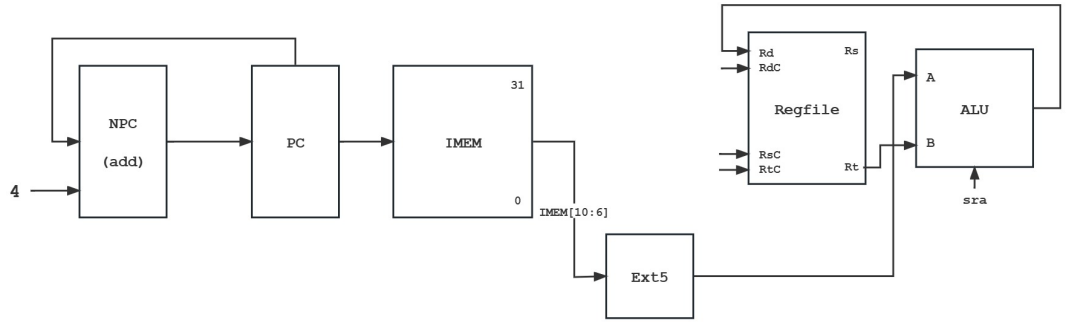
操作：取指令, rd←rt >>> sa , PC←NPC(PC+4)

所需部件：PC, NPC, IMEM, Regfile, ALU, Ext5

输入输出关系：

sra	PC	NPC	IMEM	Regfile		ALU		Ext5
				Rd	Rdc	A	B	
	NPC	PC	PC	ALU	15-11	Ext5	Rt	Sa

sra



14. sllv

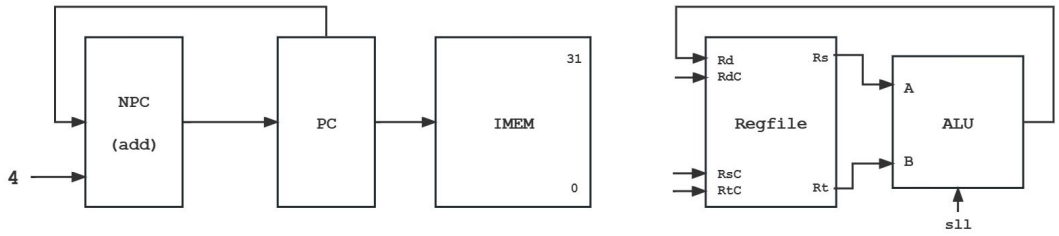
操作：取指令, rd←rt << rs , PC←NPC(PC+4)

所需部件：PC, NPC, IMEM, Regfile, ALU

输入输出关系：

sllv	PC	NPC	IMEM	Regfile		ALU	
				Rd	Rdc	A	B
	NPC	PC	PC	ALU	15-11	Rs	Rt

sllv



15. srlv

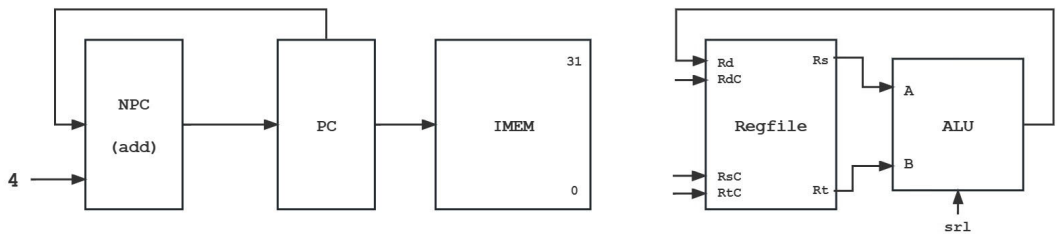
操作：取指令, rd←rt >> rs , PC←NPC(PC+4)

所需部件：PC, NPC, IMEM, Regfile, ALU

输入输出关系：

srlv	PC	NPC	IMEM	Regfile		ALU	
				Rd	Rdc	A	B
	NPC	PC	PC	ALU	15-11	Rs	Rt

srlv



16. srav

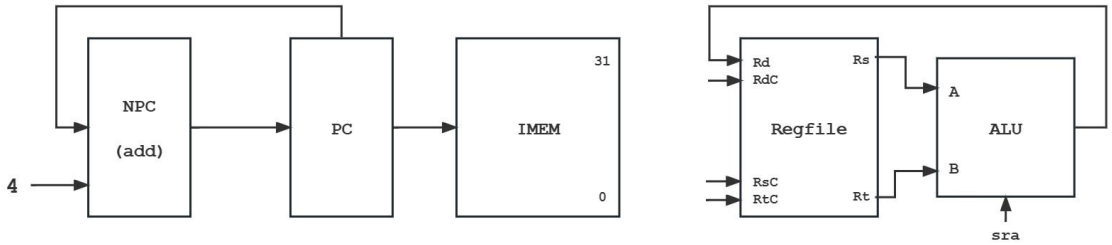
操作：取指令, rd←rt >>> rs , PC←NPC(PC+4)

所需部件：PC, NPC, IMEM, Regfile, ALU

输入输出关系：

srav	PC	NPC	IMEM	Regfile		ALU	
				Rd	Rdc	A	B
	NPC	PC	PC	ALU	15-11	Rs	Rt

srav



17. jr

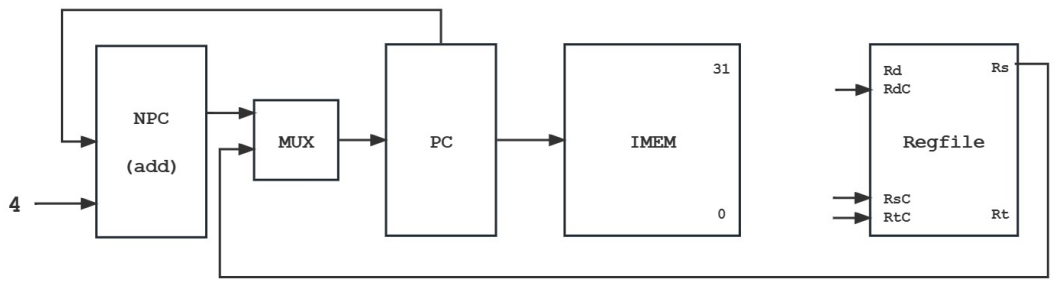
操作：取指令, PC←rs,　PC←NPC(PC+4)

所需部件：PC, NPC, IMEM, Regfile

输入输出关系：

jr	PC	NPC	IMEM	Regfile		ALU	
				Rd	Rdc	A	B
	Rs	PC	PC				

jr



18. addi

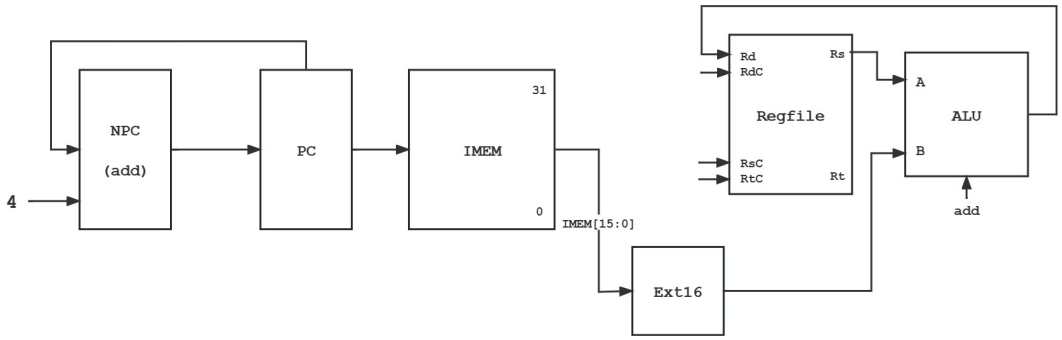
操作：取指令、rt←rs+imm16(符号扩展) 、PC←NPC(PC+4)

所需部件：PC、NPC、IMEM、Regfile、ALU、Ext16

输入输出关系：

addi	PC	NPC	IMEM	Regfile		ALU		Ext16
				Rd	Rdc	A	B	
	NPC	PC	PC	ALU	20-16	Rs	Ext16	Imm16

addi



19. addiu

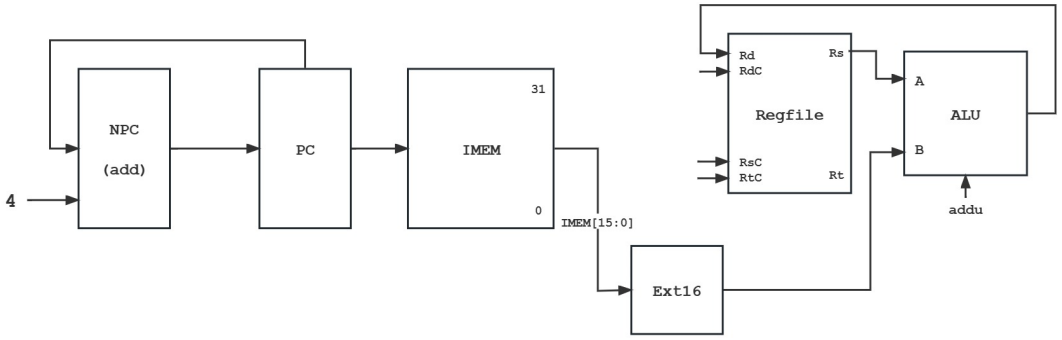
操作：取指令、rt←rs+imm16(符号扩展) 、PC←NPC(PC+4)

所需部件：PC、NPC、IMEM、Regfile、ALU、Ext16

输入输出关系：

addiu	PC	NPC	IMEM	Regfile		ALU		Ext16
				Rd	Rdc	A	B	
	NPC	PC	PC	ALU	20-16	Rs	Ext16	Imm16

addiu



20. andi

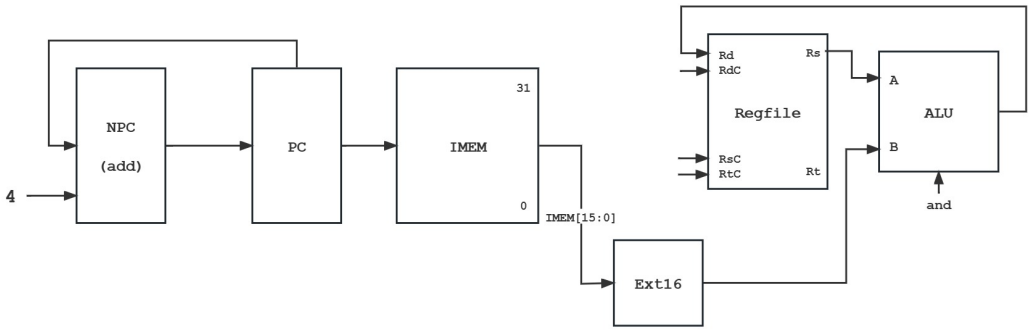
操作：取指令、rt←rs&imm16(零扩展) 、PC←NPC(PC+4)

所需部件：PC、NPC、IMEM、Regfile、ALU、Ext16

输入输出关系：

andi	PC	NPC	IMEM	Regfile		ALU		Ext16
				Rd	Rdc	A	B	
	NPC	PC	PC	ALU	20-16	Rs	Ext16	Imm16

andi



21. ori

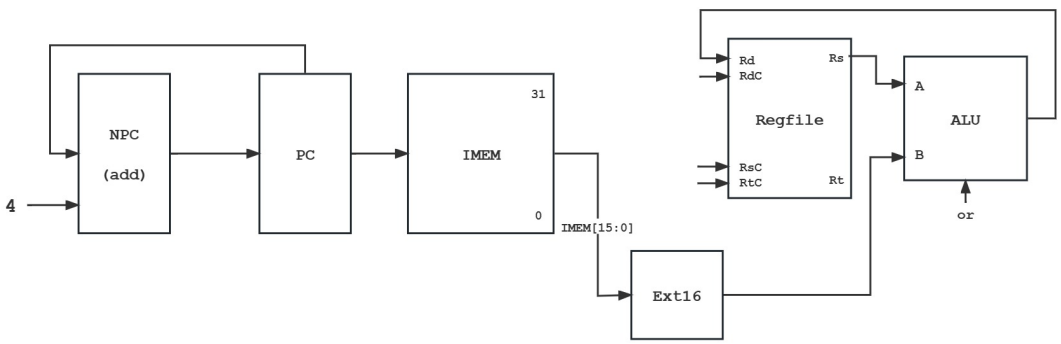
操作：取指令、rt←rs|imm16(零扩展) 、PC←NPC(PC+4)

所需部件：PC、NPC、IMEM、Regfile、ALU、Ext16

输入输出关系：

ori	PC	NPC	IMEM	Regfile		ALU		Ext16
				Rd	Rdc	A	B	
	NPC	PC	PC	ALU	20-16	Rs	Ext16	Imm16

ori



22. xori

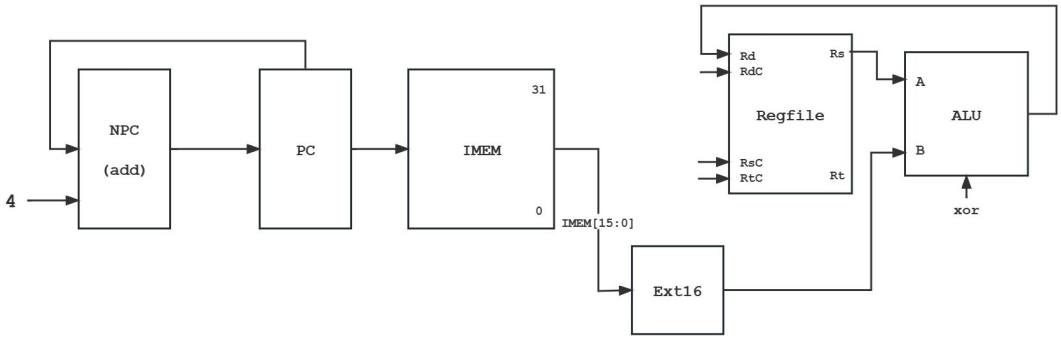
操作：取指令、rt←rs^imm16(零扩展) 、PC←NPC(PC+4)

所需部件：PC、NPC、IMEM、Regfile、ALU、Ext16

输入输出关系：

xori	PC	NPC	IMEM	Regfile		ALU		Ext16
				Rd	Rdc	A	B	
	NPC	PC	PC	ALU	20-16	Rs	Ext16	Imm16

xori



23. lw

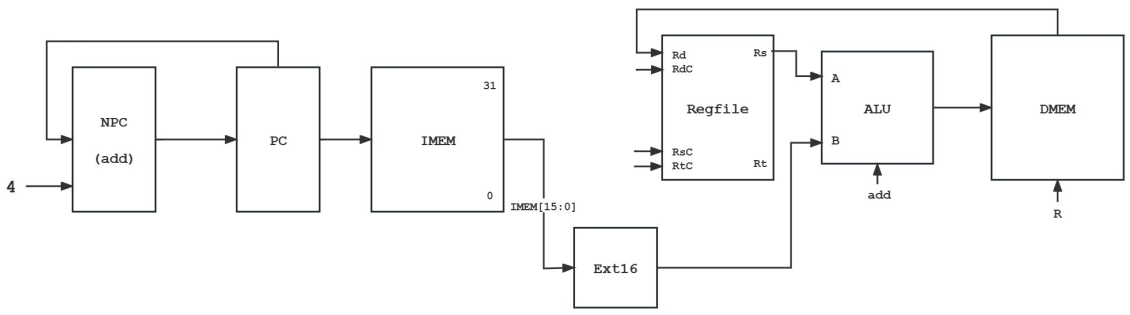
操作：取指令、rt←DMEM[rs + offset]、PC←NPC(PC+4)

所需部件：PC、NPC、IMEM、ALU、Ext16、DMEM

输入输出关系：

lw	PC	NPC	IMEM	Regfile		ALU		Ext16	DMEM	
				Rd	Rdc	A	B		Addr	Data
	NPC	PC	PC	DMEM	20-16	Rs	Ext16	Offset	ALU	

lw



24. sw

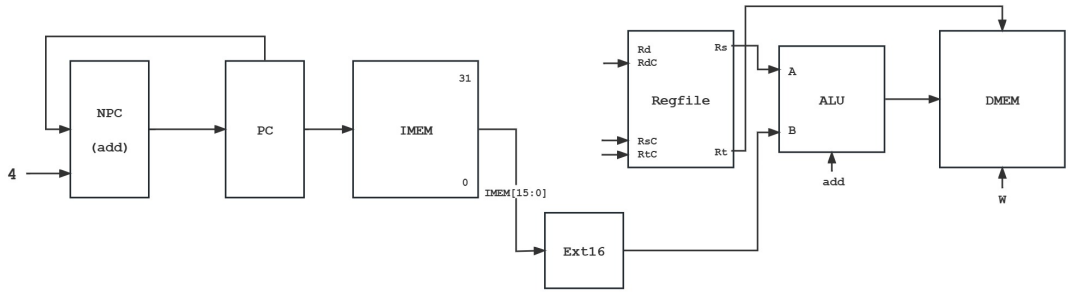
操作：取指令、DMEM[rs + offset]←rt、PC←NPC(PC+4)

所需部件：PC、NPC、IMEM、ALU、Ext16、DMEM

输入输出关系：

sw	PC	NPC	IMEM	Regfile		ALU		Ext16	DMEM	
				Rd	Rdc	A	B		Addr	Data
	NPC	PC	PC	DMEM		Rs	Ext16	Offset	ALU	Rt

sw



25. beq

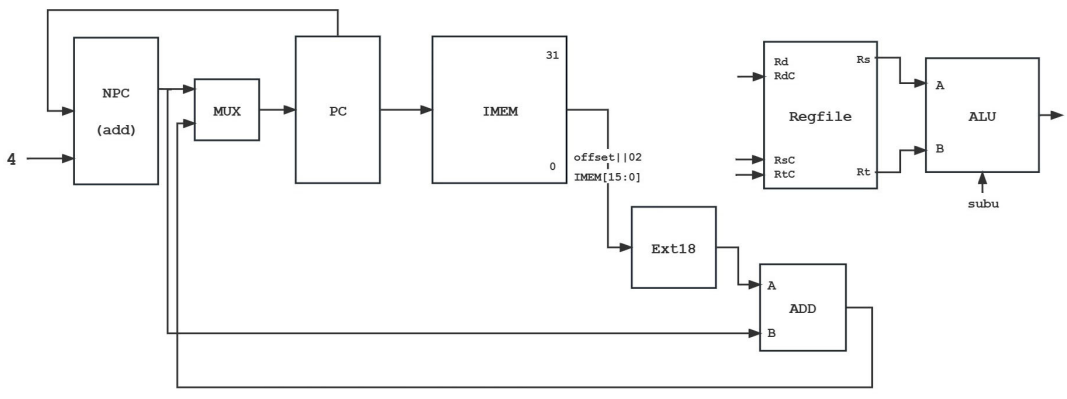
操作： if (rs==rt) PC←NPC + ext18(offset || 02) else PC← NPC(PC+4)

所需部件：PC、NPC、IMEM、Regfile、ALU、Ext18、ADD

输入输出关系：

beq	PC	NPC	IMEM	Regfile		ALU		Ext18	ADD	
				Rd	Rdc	A	B		A	B
	MUX	PC	PC			Rs	Rt	Offset	Ext18	NPC

beq



26. bne

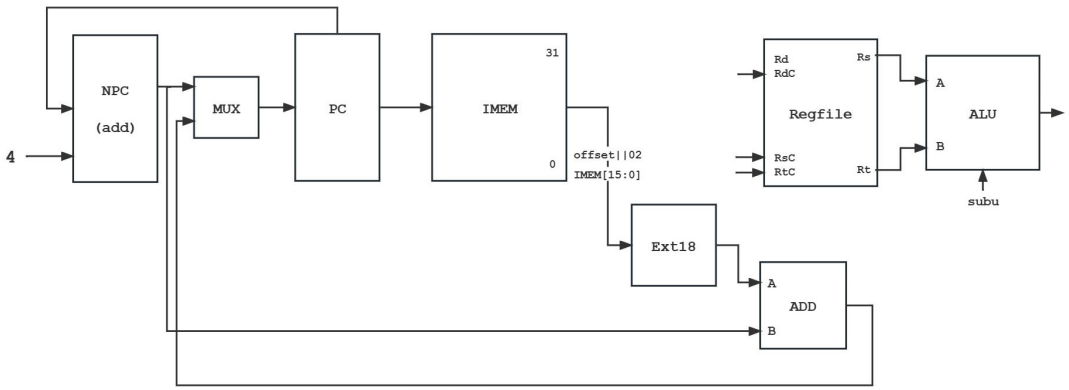
操作： if (rs!=rt)　PC←NPC + ext18(offset || 02)　else PC←　NPC(PC+4)

所需部件：PC、NPC、IMEM、Regfile、ALU、Ext18、ADD

输入输出关系：

bne	PC	NPC	IMEM	Regfile		ALU		Ext18	ADD	
				Rd	Rdc	A	B		A	B
	MUX	PC	PC			Rs	Rt	Offset	Ext18	NPC

bne



27. slti

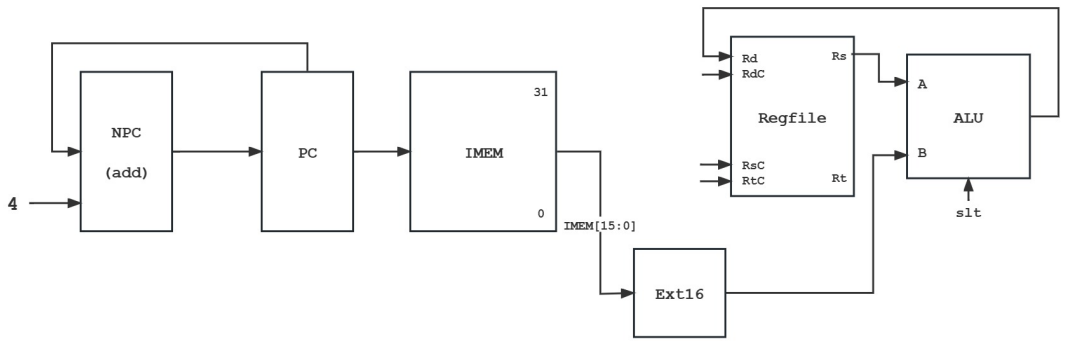
操作：取指令、rt←rs < imm16（符号扩展）、PC←NPC(PC+4)

所需部件：PC、NPC、IMEM、Rregfile、ALU、Ext16

输入输出关系：

slti	PC	NPC	IMEM	Regfile		ALU		Ext16
				Rd	Rdc	A	B	
	NPC	PC	PC	ALU	20-16	Rs	Ext16	Imm16

slti



28. sltiu

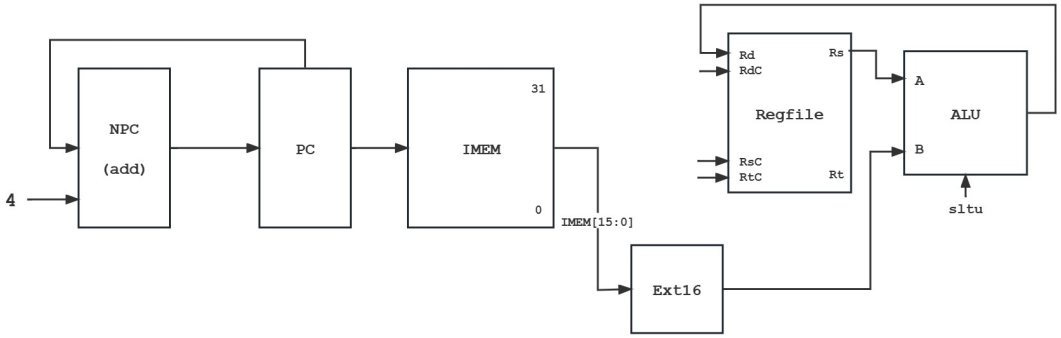
操作：取指令、rt←rs < imm16（符号扩展）、PC←NPC(PC+4)

所需部件：PC、NPC、IMEM、Rregfile、ALU、Ext16

输入输出关系：

sltiu	PC	NPC	IMEM	Regfile		ALU		Ext16
				Rd	Rdc	A	B	
	NPC	PC	PC	ALU	20-16	Rs	Ext16	Imm16

sltiu



29. lui

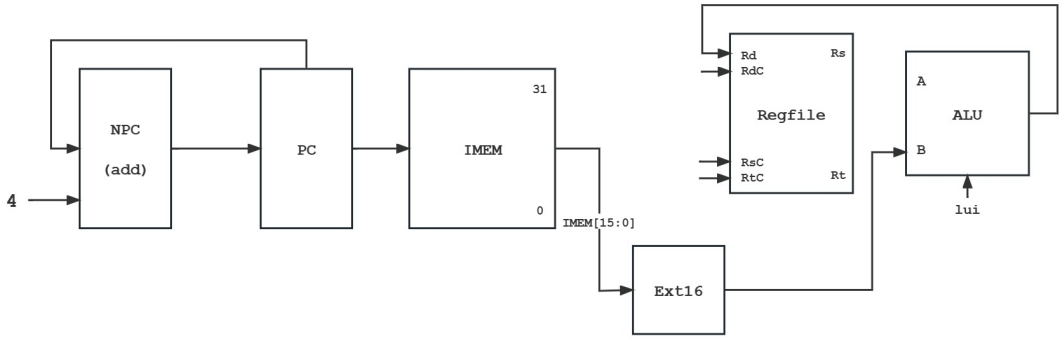
操作：取指令、rt←imm16 || 16’b0 、PC←NPC(PC+4)

所需部件：PC、NPC、IMEM、Rregfile、ALU、Ext16

输入输出关系：

lui	PC	NPC	IMEM	Rgefile		ALU		Ext16
				Rd	Rdc	A	B	
	NPC	PC	PC	ALU	20-16		Ext16	Imm16

lui



30. j

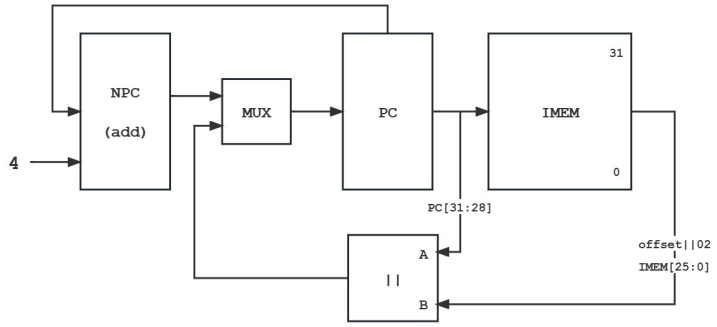
操作：取指令、PC ← PC_{31-28} || instr_index || 2’b0 , PC←NPC(PC+4)

所需部件：PC、NPC、IMEM

输入输出关系：

j	PC	NPC	IMEM	Regfile		ALU	
				Rd	Rdc	A	B
	||	PC	PC				

j



31. jal

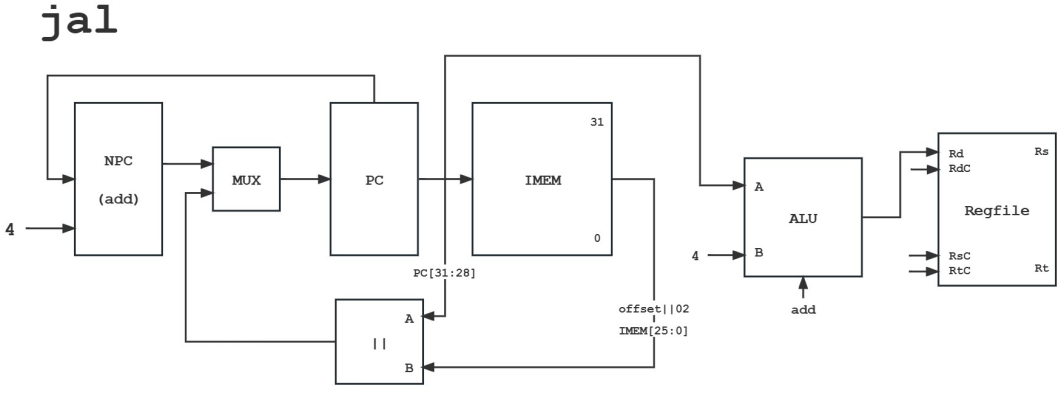
操作：取指令、Regfile[31]←PC + 8, PC←PC31-28||instr_index||2’b0 , PC←PC+4

所需部件：PC、NPC、IMEM

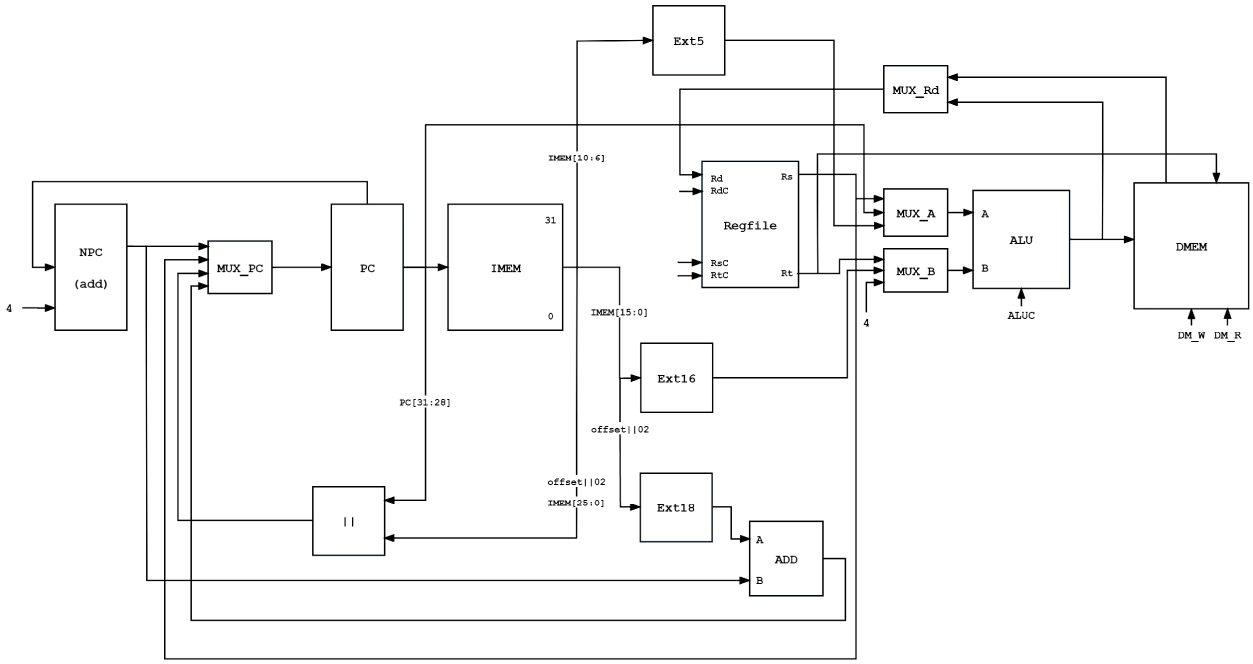
输入输出关系：

jal	PC	NPC	IMEM	Regfile		ALU	
				Rd	Rdc	A	B
	||	PC	PC	ALU		PC	8

jal



32. CPU 总数据通路图



三、模块建模

1. sccomp_dataflow

sccomp_dataflow 为顶层模块，其输入为系统的时钟信号与复位信号，输出为当前所取到的指令以及程序计数器的值，用于测试模块的信息打印。该模块负责将 CPU 模块与指令存储器 IMEM、数据存储器 DMEM 相连。

```
`timescale 1ns / 1ps

module sccomp_dataflow(
    input clk_in,   // 时钟信号
    input reset,    // 复位信号
    output [31:0] inst, // 取到的指令
    output [31:0] pc    // 程序计数器
);

    wire [31:0] imem_data;

    wire dmem_w;
```

```
    wire dmem_r;
    wire [10:0] dm_addr;
    wire [10:0] im_addr;
    wire [31:0] dmem_addr_pc;
    wire [31:0] dmem_wdata;
    wire [31:0] dmem_rdata;

    assign inst = imem_data;
    assign im_addr = (pc - 32'h00400000) / 4;
    assign dm_addr = (dmem_addr_pc - 32'h10010000) / 4;

    IMEM imem(
        .addr(im_addr),
        .data(imem_data)
    );

    DMEM dmem(
        .clk(clk_in),
        .ena(1'b1),
        .DMEM_W(dmem_w),
        .DMEM_R(dmem_r),
        .DM_addr(dm_addr),
        .DM_wdata(dmem_wdata),
        .DM_rdata(dmem_rdata)
    );

    cpu sccpu(
        .clk_in(clk_in),
        .ena(1'b1),
        .rst(reset),
        .PC(pc),
        .imem(imem_data),
        .dmem_out(dmem_rdata),
        .dmem_in(dmem_wdata),
        .dmem_addr(dmem_addr_pc),
        .dmem_w(dmem_w),
        .dmem_r(dmem_r)
    );

endmodule
```

2. IMEM

IMEM 为指令存储器模块，通过调用 ROM 的 IP 核实现。代码如下：

```
`timescale 1ns / 1ps

module IMEM(
    input   [10:0]  addr,   // 指令地址
    output  [31:0]  data    // 所取指令
    );

    dist_mem_gen_0 IM(
        .a(addr),
        .spo(data)
    );

endmodule
```

3. DMEM

DMEM 为数据存储器模块。其输入信号有时钟信号、使能信号、写有效信号与读有效信号。此外，还有地址以及写入的数据作为模块输入，以及读出的数据作为模块的输出。具体代码如下：

```
`timescale 1ns / 1ps

module DMEM(
    input           clk,        // 时钟信号
    input           ena,        // 使能信号
    input           DMEM_W,     // 写有效信号
    input           DMEM_R,     // 读有效信号
    input   [10:0]  DM_addr,    // 地址
    input   [31:0]  DM_wdata,   // 写入数据
    output  [31:0]  DM_rdata    // 读出数据
    );

    reg [31:0] DM [0:2047];

    always@(posedge clk)
    begin
        if(ena && DMEM_W)
        begin
```

```
            DM[DM_addr] <= DM_wdata;
        end
    end

    assign DM_rdata = (ena && DMEM_R) ? DM[DM_addr] : 32'bz;

endmodule
```

4. CPU

根据各个指令的流程图以及上述的 CPU 总数据通路图，罗列出各个指令的控制信号表（部分）如下所示：

	IM_R	DM_R	DM_W	ALUC	MUX_PC	MUX_A	MUX_B	MUX_Rd	Reg_W
add	1	0	0	0000	00	00	00	00	1
addu	1	0	0	0001	00	00	00	00	1
sub	1	0	0	0010	00	00	00	00	1
subu	1	0	0	0011	00	00	00	00	1
and	1	0	0	0100	00	00	00	00	1
or	1	0	0	0101	00	00	00	00	1
xor	1	0	0	0110	00	00	00	00	1
nor	1	0	0	0111	00	00	00	00	1
slt	1	0	0	1000	00	00	00	00	1
sltu	1	0	0	1001	00	00	00	00	1
sll	1	0	0	1010	00	10	00	00	1
srl	1	0	0	1011	00	10	00	00	1
sra	1	0	0	1100	00	10	00	00	1
sllv	1	0	0	1010	00	00	00	00	1
srlv	1	0	0	1011	00	00	00	00	1
srav	1	0	0	1100	00	00	00	00	1
jr	1	0	0		01				0
addi	1	0	0	0000	00	00	01	00	1
addiu	1	0	0	0001	00	00	01	00	1
andi	1	0	0	0100	00	00	01	00	1
ori	1	0	0	0101	00	00	01	00	1
xori	1	0	0	0110	00	00	01	00	1
lw	1	1	0	0000	00	00	01	01	1

sw	1	0	1	0000	00	00	01		0
beq	1	0	0	0011	zero?00:11	00	00		0
bne	1	0	0	0011	~zero?00:11	00	00		0
slti	1	0	0	1000	00	00	01	00	1
sltiu	1	0	0	1001	00	00	01	00	1
lui	1	0	0	1101	00	00	01	00	1
j	1	0	0		10				0
jal	1	0	0	0000	10	01	10	00	1

根据上表，可以列出各个信号对应的布尔方程，由此进行控制信号设计。CPU 部分的代码如下：

```
`timescale 1ns / 1ps

module cpu(
    input           clk_in,     // 时钟信号
    input           ena,        // 使能信号
    input           rst,        // 复位信号
    input [31:0]    imem,       // 当前读到的指令
    input [31:0]    dmem_out,   // DMEM 读出的数据
    output [31:0]   dmem_in,    // DMEM 写入的数据
    output [31:0]   dmem_addr,  // DMEM 读/写地址
    output          dmem_w,     // DMEM 写信号
    output          dmem_r,     // DMEM 读信号
    output [31:0]   PC          // 程序计数器
    );

    // R-type
    wire Add    = (imem[31:26] == 6'b000000 && imem[5:0] == 6'b100000) ?
1'b1 : 1'b0;
    wire Addu   = (imem[31:26] == 6'b000000 && imem[5:0] == 6'b100001) ?
1'b1 : 1'b0;
    wire Sub    = (imem[31:26] == 6'b000000 && imem[5:0] == 6'b100010) ?
1'b1 : 1'b0;
    wire Subu   = (imem[31:26] == 6'b000000 && imem[5:0] == 6'b100011) ?
1'b1 : 1'b0;
    wire And    = (imem[31:26] == 6'b000000 && imem[5:0] == 6'b100100) ?
1'b1 : 1'b0;
    wire Or     = (imem[31:26] == 6'b000000 && imem[5:0] == 6'b100101) ?
1'b1 : 1'b0;
    wire Xor    = (imem[31:26] == 6'b000000 && imem[5:0] == 6'b100110) ?
```

```
1'b1 : 1'b0;
    wire Nor    = (imem[31:26] == 6'b000000 && imem[5:0] == 6'b100111) ?
1'b1 : 1'b0;
    wire Slt    = (imem[31:26] == 6'b000000 && imem[5:0] == 6'b101010) ?
1'b1 : 1'b0;
    wire Sltu   = (imem[31:26] == 6'b000000 && imem[5:0] == 6'b101011) ?
1'b1 : 1'b0;
    wire Sll    = (imem[31:26] == 6'b000000 && imem[5:0] == 6'b000000) ?
1'b1 : 1'b0;
    wire Srl    = (imem[31:26] == 6'b000000 && imem[5:0] == 6'b000010) ?
1'b1 : 1'b0;
    wire Sra    = (imem[31:26] == 6'b000000 && imem[5:0] == 6'b000011) ?
1'b1 : 1'b0;
    wire Sllv   = (imem[31:26] == 6'b000000 && imem[5:0] == 6'b000100) ?
1'b1 : 1'b0;
    wire Srlv   = (imem[31:26] == 6'b000000 && imem[5:0] == 6'b000110) ?
1'b1 : 1'b0;
    wire Srav   = (imem[31:26] == 6'b000000 && imem[5:0] == 6'b000111) ?
1'b1 : 1'b0;
    wire Jr     = (imem[31:26] == 6'b000000 && imem[5:0] == 6'b001000) ?
1'b1 : 1'b0;
    wire Rtype  = (imem[31:26] == 6'b000000);

    // I-type
    wire Addi   = (imem[31:26] == 6'b001000) ? 1'b1 : 1'b0;
    wire Addiu  = (imem[31:26] == 6'b001001) ? 1'b1 : 1'b0;
    wire Andi   = (imem[31:26] == 6'b001100) ? 1'b1 : 1'b0;
    wire Ori    = (imem[31:26] == 6'b001101) ? 1'b1 : 1'b0;
    wire Xori   = (imem[31:26] == 6'b001110) ? 1'b1 : 1'b0;
    wire Lw     = (imem[31:26] == 6'b100011) ? 1'b1 : 1'b0;
    wire Sw     = (imem[31:26] == 6'b101011) ? 1'b1 : 1'b0;
    wire Beq    = (imem[31:26] == 6'b000100) ? 1'b1 : 1'b0;
    wire Bne    = (imem[31:26] == 6'b000101) ? 1'b1 : 1'b0;
    wire Slti   = (imem[31:26] == 6'b001010) ? 1'b1 : 1'b0;
    wire Sltiu  = (imem[31:26] == 6'b001011) ? 1'b1 : 1'b0;
    wire Lui    = (imem[31:26] == 6'b001111) ? 1'b1 : 1'b0;
    wire Itype  = Addi || Addiu || Andi || Ori || Xori || Lw || Sw || Beq
|| Bne || Slti || Sltiu || Lui;

    // J-type
    wire J      = (imem[31:26] == 6'b000010) ? 1'b1 : 1'b0;
    wire Jal    = (imem[31:26] == 6'b000011) ? 1'b1 : 1'b0;
    wire Jtype  = J || Jal;
```

```
    // PC
    wire [31:0] PC_in;
    wire [31:0] NPC = PC + 32'd4;

    // ALU
    wire [3:0] ALUC;
    wire zero, carry, negative, overflow;
    wire [31:0] A;
    wire [31:0] B;
    wire [31:0] Y;

    // Regfile
    wire reg_wena;
    wire [4:0] rsc;
    wire [4:0] rtc;
    wire [4:0] rdc;
    wire [31:0] rs;
    wire [31:0] rt;
    wire [31:0] rd;

    // MUX
    wire [1:0] mux_pc;
    wire [1:0] mux_a;
    wire [1:0] mux_b;

    // Ext
    wire [31:0] ext5;
    wire        extb;
    wire [31:0] ext16;
    wire [31:0] ext18;
    wire [31:0] ext18add;
    wire [31:0] concat;

    // ALU
    assign ALUC[3] = Slt || Sltu || Sll || Srl || Sra || Sllv || Srlv ||
Srav || Slti || Sltiu || Lui;
    assign ALUC[2] = And || Or || Xor || Nor || Sra || Srav || Andi || Ori
|| Xori || Lui;
    assign ALUC[1] = Sub || Subu || Xor || Nor || Sll || Srl || Sllv || Srlv
|| Xori || Beq || Bne;
    assign ALUC[0] = Addu || Subu || Or || Nor || Sltu || Srl || Srlv ||
Addiu || Ori || Beq || Bne || Sltiu || Lui;
```

```
    // MUX
    assign mux_pc[1] = (Beq && zero) || (Bne && ~zero) || J || Jal;
    assign mux_pc[0] = Jr || (Beq && zero) || (Bne && ~zero);
    assign mux_a[1]  = Sll || Srl || Sra;
    assign mux_a[0]  = Jal;
    assign mux_b[1]  = Jal;
    assign mux_b[0]  = Addi || Addiu || Andi || Ori || Xori || Lw || Sw ||
Slti || Sltiu || Lui;

    // Regfile
    assign rsc = imem[25:21];
    assign rtc = imem[20:16];
    assign rdc = Rtype ? imem[15:11] : (Itype ? imem[20:16] : 5'd31);
    assign rd = Lw ? dmem_out : Y;
    assign reg_wena = !(Jr || Sw || Beq || Bne || J);

    // DMEM
    assign dmem_w = Sw;
    assign dmem_r = Lw;
    assign dmem_addr = (Lw || Sw) ? Y : 32'bz;
    assign dmem_in = rt;

    // Ext 和其他模块
    assign ext5  = {  27'b0, imem[10:6] };
    assign extb  = ( Andi || Ori || Xori || Lui) ? 1'b0 : imem[15];
    assign ext16 = { {16 { extb }}, imem[15:0] };
    assign ext18 = { {14{ imem[15] }}, imem[15:0], 2'b0 };
    assign ext18add = ext18 + NPC;
    assign concat   = { PC[31:28], imem[25:0], 2'b0 };


    MUX4 MUX_PC(
        .d0(NPC),
        .d1(rs),
        .d2(concat),
        .d3(ext18add),
        .s(mux_pc),
        .y(PC_in)
        );

    MUX4 MUX_A(
        .d0(rs),
```

```
        .d1(PC),
        .d2(ext5),
        .d3(32'b0),
        .s(mux_a),
        .y(A)
        );

    MUX4 MUX_B(
        .d0(rt),
        .d1(ext16),
        .d2(32'd4),
        .d3(32'b0),
        .s(mux_b),
        .y(B)
        );

    PCreg pcreg(
        .clk(clk_in),
        .ena(ena),
        .rst(rst),
        .PC_in(PC_in),
        .PC_out(PC)
        );

    ALU alu(
        .A(A),
        .B(B),
        .Y(Y),
        .ALUC(ALUC),
        .zero(zero),
        .carry(carry),
        .negative(negative),
        .overflow(overflow)
        );

    regfile cpu_ref(
        .ena(ena),
        .rst(rst),
        .clk(clk_in),
        .w_ena(reg_wena),
        .Rdc(rdc),
        .Rsc(rsc),
        .Rtc(rtc),
```

```
        .Rd(rd),
        .Rs(rs),
        .Rt(rt)
        );

endmodule
```

5. PCreg

PC 寄存器模块用于实现程序计数器，从而指示下一条应该读出的指令位置。具体的模块代码如下：

```
`timescale 1ns / 1ps

module PCreg(
    input               clk,    // 时钟信号
    input               ena,    // 使能信号
    input               rst,    // 复位信号
    input      [31:0]   PC_in,  // 输入的 PC 数据
    output reg [31:0]   PC_out  // 输出的 PC 数据
    );

    always@(negedge clk or posedge rst)
    begin
        if(rst)
        begin
            PC_out <= 32'h00400000;
        end
        else if(ena)
        begin
            PC_out <= PC_in;
        end
    end

endmodule
```

6. Regfile

Regfile 模块用于实现寄存器堆。其中输入信号包括使能信号、复位信号、时钟信号、写有效信号等。通过 Rsc、Rtc 的寄存器编号输入决定读出的寄存器

内容，通过 Rdc 的寄存器编号输入决定写入的寄存器内容。对应的数据通过 Rd、Rs、Rt 传入或传出。

```
`timescale 1ns / 1ps

module regfile(
    input           ena,    // 使能信号
    input           rst,    // 复位信号
    input           clk,    // 时钟信号
    input           w_ena,  // 写有效信号
    input   [4:0]   Rdc,    // Rd 寄存器编号
    input   [4:0]   Rsc,    // Rs 寄存器编号
    input   [4:0]   Rtc,    // Rt 寄存器编号
    input   [31:0]  Rd,
    output  [31:0]  Rs,
    output  [31:0]  Rt
    );

    reg [31:0] array_reg [31:0];

    always@(posedge clk or posedge rst)
    begin
        if(ena && rst)
        begin
            array_reg[0]  <= 32'b0;
            array_reg[1]  <= 32'b0;
            array_reg[2]  <= 32'b0;
            array_reg[3]  <= 32'b0;
            array_reg[4]  <= 32'b0;
            array_reg[5]  <= 32'b0;
            array_reg[6]  <= 32'b0;
            array_reg[7]  <= 32'b0;
            array_reg[8]  <= 32'b0;
            array_reg[9]  <= 32'b0;
            array_reg[10] <= 32'b0;
            array_reg[11] <= 32'b0;
            array_reg[12] <= 32'b0;
            array_reg[13] <= 32'b0;
            array_reg[14] <= 32'b0;
            array_reg[15] <= 32'b0;
            array_reg[16] <= 32'b0;
            array_reg[17] <= 32'b0;
            array_reg[18] <= 32'b0;
```

```
            array_reg[19] <= 32'b0;
            array_reg[20] <= 32'b0;
            array_reg[21] <= 32'b0;
            array_reg[22] <= 32'b0;
            array_reg[23] <= 32'b0;
            array_reg[24] <= 32'b0;
            array_reg[25] <= 32'b0;
            array_reg[26] <= 32'b0;
            array_reg[27] <= 32'b0;
            array_reg[28] <= 32'b0;
            array_reg[29] <= 32'b0;
            array_reg[30] <= 32'b0;
            array_reg[31] <= 32'b0;
        end
        else if(ena && w_ena && Rdc)
        begin
            array_reg[Rdc] <= Rd;
        end
    end


    assign Rs = ena ? array_reg[Rsc] : 32'bz;
    assign Rt = ena ? array_reg[Rtc] : 32'bz;

endmodule
```

7. ALU

ALU 模块即运算器，负责对数据进行运算。根据 31 条指令，将不同运算操作所对应的选择信号 ALUC 整理设计如下表：

	ALUC[3]	ALUC[2]	ALUC[1]	ALUC[0]
ADD	0	0	0	0
ADDU	0	0	0	1
SUB	0	0	1	0
SUBU	0	0	1	1
AND	0	1	0	0

OR	0	1	0	1
XOR	0	1	1	0
NOR	0	1	1	1
SLT	1	0	0	0
SLTU	1	0	0	1
SLL	1	0	1	0
SRL	1	0	1	1
SRA	1	1	0	0
LUI	1	1	0	1

根据如上的选择信号表格编写代码如下：

```
`timescale 1ns / 1ps

module ALU(
    input   [31:0]  A,          // 操作数
    input   [31:0]  B,          // 操作数
    output  [31:0]  Y,          // 结果
    input   [3:0]   ALUC,       // 选择信号
    output          zero,       // 以下为标志位
    output          carry,
    output          negative,
    output          overflow
    );

    wire signed [31:0] signedA, signedB;
    reg [32:0] ans;

    assign signedA = A;
    assign signedB = B;

    parameter ADD   =   4'b0000;
    parameter ADDU  =   4'b0001;
    parameter SUB   =   4'b0010;
    parameter SUBU  =   4'b0011;
    parameter AND   =   4'b0100;
    parameter OR    =   4'b0101;
```

```
    parameter XOR   =   4'b0110;
    parameter NOR   =   4'b0111;
    parameter SLT   =   4'b1000;
    parameter SLTU  =   4'b1001;
    parameter SLL   =   4'b1010;
    parameter SRL   =   4'b1011;
    parameter SRA   =   4'b1100;
    parameter LUI   =   4'b1101;

    always@ (*)
    begin
        case(ALUC)
            ADD:
            begin
                ans = signedA + signedB;
            end
            ADDU:
            begin
                ans = A + B;
            end
            SUB:
            begin
                ans = signedA - signedB;
            end
            SUBU:
            begin
                ans = A - B;
            end
            AND:
            begin
                ans = A & B;
            end
            OR:
            begin
                ans = A | B;
            end
            XOR:
            begin
                ans = A ^ B;
            end
            NOR:
            begin
                ans = ~(A | B);
```

```
            end
            SLT:
            begin
                ans = (signedA < signedB);
            end
            SLTU:
            begin
                ans = (A < B);
            end
            SLL:
            begin
                ans = (B << A);
            end
            SRL:
            begin
                if(A == 0)
                    { ans[31:0], ans[32] } = { B, 1'b0 };
                else
                    { ans[31:0], ans[32] } = B >> (A - 1);
            end
            SRA:
            begin
                if(A == 0)
                    { ans[31:0], ans[32] } = { signedB, 1'b0 };
                else
                    { ans[31:0], ans[32] } = signedB >>> (A - 1);
            end
            LUI:
            begin
                ans = { B[15:0], 16'b0 };
            end
        endcase
    end

    assign Y        = ans[31:0];
    assign zero     = (ans == 32'b0) ? 1'b1 : 1'b0;
    assign carry    = ans[32];
    assign overflow = ans[32] ^ ans[31];
    assign negative = ans[31];

endmodule
```

四、测试模块建模

1. CPU 测试模块

```
module test_cpu(
    );

    reg clk, rst;
    wire [31:0] inst, pc;

    integer file_open;
    initial begin
        clk = 1'b0;
        rst = 1'b1;
        #10 rst = 1'b0;
    end

    always
    begin
        #20 clk = !clk;
    end

    always@(posedge clk)
    begin
        file_open = $fopen("output.txt", "a");
        $fdisplay(file_open, "pc: %h", pc);
        $fdisplay(file_open, "instr: %h", inst);
        $fdisplay(file_open, "regfile0: %h", uut.sccpu.cpu_ref.array_re
g[0]);
        $fdisplay(file_open, "regfile1: %h", uut.sccpu.cpu_ref.array_re
g[1]);
        $fdisplay(file_open, "regfile2: %h", uut.sccpu.cpu_ref.array_re
g[2]);
        $fdisplay(file_open, "regfile3: %h", uut.sccpu.cpu_ref.array_re
g[3]);
        $fdisplay(file_open, "regfile4: %h", uut.sccpu.cpu_ref.array_re
g[4]);
        $fdisplay(file_open, "regfile5: %h", uut.sccpu.cpu_ref.array_re
g[5]);
        $fdisplay(file_open, "regfile6: %h", uut.sccpu.cpu_ref.array_re
g[6]);
        $fdisplay(file_open, "regfile7: %h", uut.sccpu.cpu_ref.array_re
```

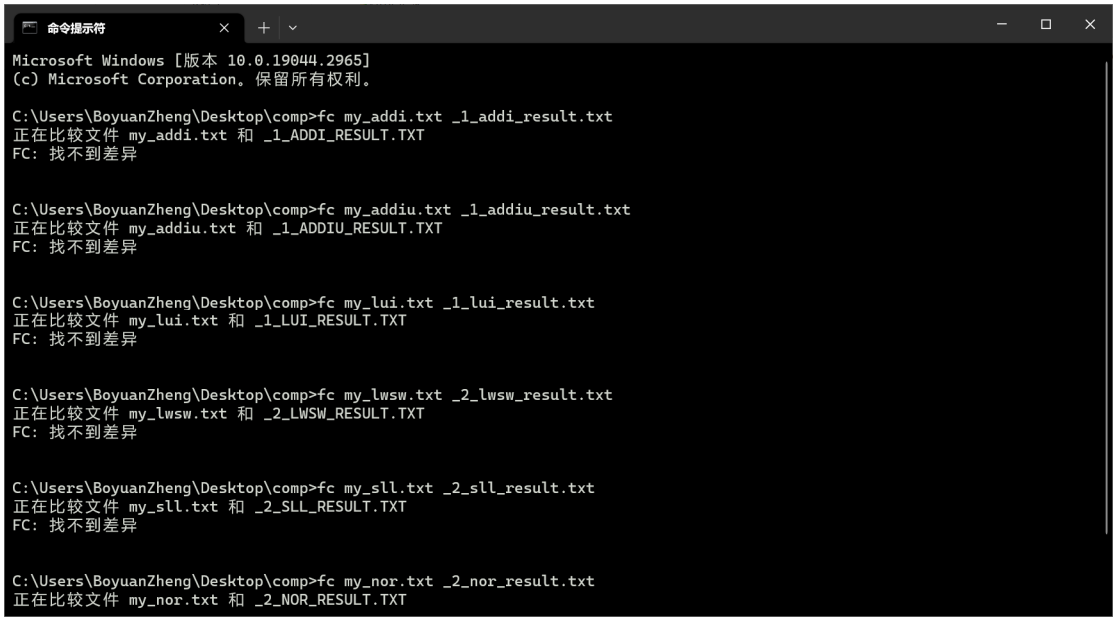
```
g[7]);
        $fdisplay(file_open, "regfile8: %h", uut.sccpu.cpu_ref.array_re
g[8]);
        $fdisplay(file_open, "regfile9: %h", uut.sccpu.cpu_ref.array_re
g[9]);
        $fdisplay(file_open, "regfile10: %h", uut.sccpu.cpu_ref.array_r
eg[10]);
        $fdisplay(file_open, "regfile11: %h", uut.sccpu.cpu_ref.array_r
eg[11]);
        $fdisplay(file_open, "regfile12: %h", uut.sccpu.cpu_ref.array_r
eg[12]);
        $fdisplay(file_open, "regfile13: %h", uut.sccpu.cpu_ref.array_r
eg[13]);
        $fdisplay(file_open, "regfile14: %h", uut.sccpu.cpu_ref.array_r
eg[14]);
        $fdisplay(file_open, "regfile15: %h", uut.sccpu.cpu_ref.array_r
eg[15]);
        $fdisplay(file_open, "regfile16: %h", uut.sccpu.cpu_ref.array_r
eg[16]);
        $fdisplay(file_open, "regfile17: %h", uut.sccpu.cpu_ref.array_r
eg[17]);
        $fdisplay(file_open, "regfile18: %h", uut.sccpu.cpu_ref.array_r
eg[18]);
        $fdisplay(file_open, "regfile19: %h", uut.sccpu.cpu_ref.array_r
eg[19]);
        $fdisplay(file_open, "regfile20: %h", uut.sccpu.cpu_ref.array_r
eg[20]);
        $fdisplay(file_open, "regfile21: %h", uut.sccpu.cpu_ref.array_r
eg[21]);
        $fdisplay(file_open, "regfile22: %h", uut.sccpu.cpu_ref.array_r
eg[22]);
        $fdisplay(file_open, "regfile23: %h", uut.sccpu.cpu_ref.array_r
eg[23]);
        $fdisplay(file_open, "regfile24: %h", uut.sccpu.cpu_ref.array_r
eg[24]);
        $fdisplay(file_open, "regfile25: %h", uut.sccpu.cpu_ref.array_r
eg[25]);
        $fdisplay(file_open, "regfile26: %h", uut.sccpu.cpu_ref.array_r
eg[26]);
        $fdisplay(file_open, "regfile27: %h", uut.sccpu.cpu_ref.array_r
eg[27]);
        $fdisplay(file_open, "regfile28: %h", uut.sccpu.cpu_ref.array_r
eg[28]);
```

```
        $fdisplay(file_open, "regfile29: %h", uut.sccpu.cpu_ref.array_r
eg[29]);
        $fdisplay(file_open, "regfile30: %h", uut.sccpu.cpu_ref.array_r
eg[30]);
        $fdisplay(file_open, "regfile31: %h", uut.sccpu.cpu_ref.array_r
eg[31]);
        $fclose(file_open);
    end

    sccomp_dataflow uut(
        .clk_in(clk),
        .reset(rst),
        .inst(inst),
        .pc(pc)
    );

endmodule
```

五、实验结果

1. 前仿真结果

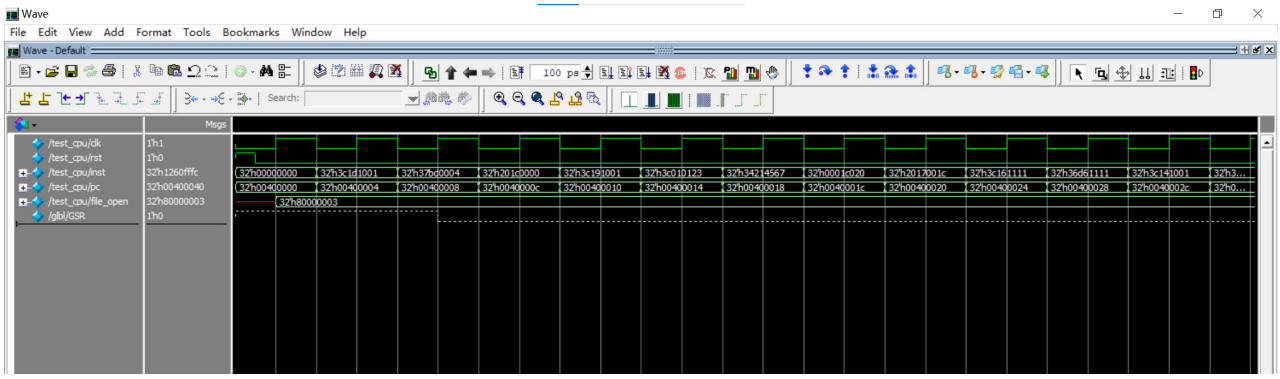
将 31 条指令依次输出到文件，与 MARS 标准文件进行比对。每条指令对应的 Regfile 输出信息均一致。具体如下图：



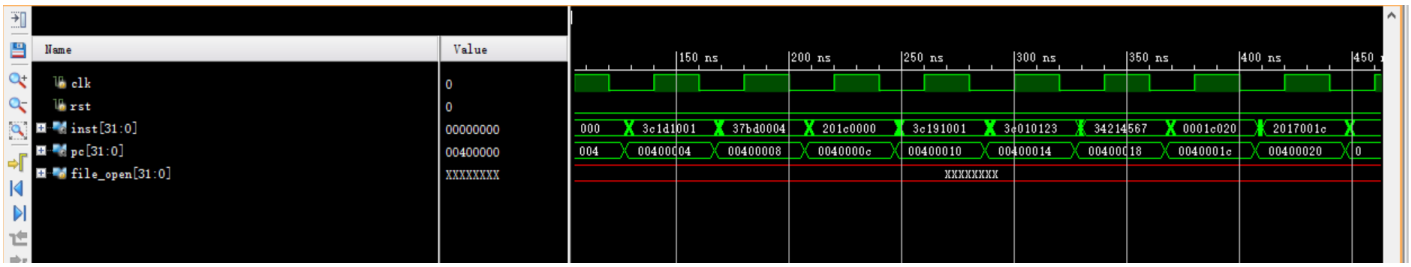
my_add.txt	2023/5/12 15:10	Text Document	101 KB
my_addi.txt	2023/5/11 16:56	Text Document	40 KB
my_addiu.txt	2023/5/12 14:51	Text Document	40 KB
my_addu.txt	2023/5/12 15:52	Text Document	101 KB
my_and.txt	2023/5/12 16:01	Text Document	103 KB
my_andi.txt	2023/5/12 16:15	Text Document	44 KB
my_beq.txt	2023/5/12 23:39	Text Document	40 KB
my_bne.txt	2023/5/12 23:41	Text Document	51 KB
my_j.txt	2023/5/12 23:43	Text Document	67 KB
my_jal.txt	2023/5/12 23:45	Text Document	66 KB
my_jr.txt	2023/5/12 23:46	Text Document	13 KB
my_lui.txt	2023/5/12 14:55	Text Document	23 KB
my_lwsw.txt	2023/5/12 21:36	Text Document	67 KB
my_lwsw2.txt	2023/5/12 21:39	Text Document	67 KB
my_nor.txt	2023/5/12 21:43	Text Document	103 KB
my_or.txt	2023/5/12 21:45	Text Document	103 KB
my_ori.txt	2023/5/12 21:56	Text Document	44 KB
my_sll.txt	2023/5/12 22:10	Text Document	23 KB
my_sllv.txt	2023/5/12 22:13	Text Document	40 KB
my_slt.txt	2023/5/12 22:14	Text Document	45 KB
my_slti.txt	2023/5/12 22:16	Text Document	44 KB
my_sltiu.txt	2023/5/12 22:21	Text Document	44 KB
my_sltu.txt	2023/5/12 22:25	Text Document	45 KB
my_sra.txt	2023/5/12 22:26	Text Document	23 KB
my_srav.txt	2023/5/12 22:27	Text Document	40 KB
my_srl.txt	2023/5/12 22:32	Text Document	23 KB
my_srlv.txt	2023/5/12 22:34	Text Document	40 KB
my_sub.txt	2023/5/12 22:36	Text Document	81 KB
my_subu.txt	2023/5/12 22:37	Text Document	101 KB
my_xor.txt	2023/5/12 22:40	Text Document	103 KB
my_xori.txt	2023/5/12 22:45	Text Document	44 KB

2. 后仿真结果

用 mips_31_mars_simulate.coe 初始化 IMEM ip 核，比较前仿真与后仿真结果的波形图。观察到，考虑了门电路延迟等情况的后仿真 PC 取指等情况与前仿真一致无误。



前仿真结果



后仿真结果

3. 下板结果

用 mips_31_mars_board_switch.coe 初始化 IMEM ip 核，并引入 7 段数码管模块、引入约束文件、编写顶层模块。在 7 段数码管上显示 PC 值，观察到能够正常读取指令如下：