

硬件课程设计1

实验报告

|  |  |
| --- | --- |
| **姓 名：** |  |
| **学 号：** |  |
| **学 院：** | 计算机科学与工程学院 |
| **专 业：** | 软件工程 |
| **题 目：** | MIPS多周期CPU设计 |
| **指导老师：** |  |

2022 年 9 月

目录

[1. 实验目的 3](#_Toc113177032)

[2. 实验内容 3](#_Toc113177033)

[2.1 MIPS指令的3种格式 3](#_Toc113177034)

[2.2 需完成设计的MIPS指令集 3](#_Toc113177035)

[2.3 MIPS指令解析 4](#_Toc113177036)

[3. 实验原理 5](#_Toc113177037)

[3.1 指令周期 5](#_Toc113177038)

[3.3 MIPS多周期CPU控制部件电路结构 7](#_Toc113177039)

[4. 实验仪器 8](#_Toc113177040)

[5. 实验过程和结果 8](#_Toc113177041)

[5.1 主要模块 8](#_Toc113177042)

[5.2 模块代码分析 9](#_Toc113177043)

[5.3 测试代码 19](#_Toc113177044)

[5.4 测试代码分析 21](#_Toc113177045)

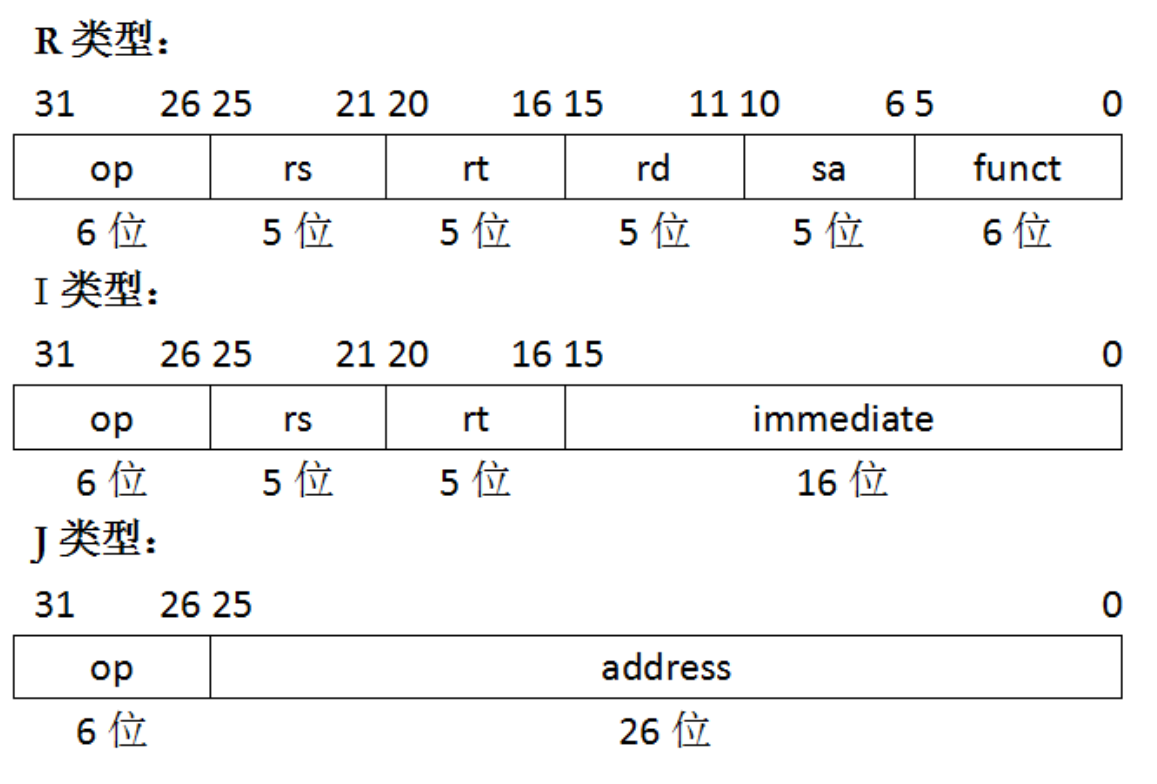
[6. 心得 30](#_Toc113177046)

# 实验目的

1. 认识并掌握MIPS多周期CPU数据通路的构成、原理和设计方法。
2. 了解MIPS多周期CPU的代码实现方法。
3. 认识MIPS指令和MIPS多周期CPU之间的关系。
4. 测试MIPS多周期CPU的执行。

# 实验内容

**2.1 MIPS指令的3种格式**



其中，

op：操作码。

rs：第一个源操作数寄存器。（共32个寄存器）

rt：第二个源操作数寄存器。

rd：目的操作数寄存器。

shamt：位移量，移位指令用于指定移位多少。

func：功能码，在R类型指令中用来指定指令的功能。

immediate：16位立即数，用于无符号的逻辑操作数、有符号的算术操作数、数据的加载和保存指令（LW和SW）的数据地址字节偏移量和分支指令中相对程序计数器的有符号偏移量。

address：地址。

**2.2 需完成设计的MIPS指令集**

设计MIPS多周期CPU，使该CPU完成加减运算、移位运算、跳转、分支判断和存取等功能的指令。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | [31:26] | [25:21] | [20:16] | [15:11] | [10:6] | [5:0] | 意义 |
| ADD | 000000 | rs | rt | rd | 00000 | 100000 | 寄存器加 |
| SUB | 000000 | rs | rt | rd | 00000 | 100010 | 寄存器减 |
| AND | 000000 | rs | rt | rd | 00000 | 100100 | 寄存器与 |
| OR | 000000 | rs | rt | rd | 00000 | 100101 | 寄存器或 |
| XOR | 000000 | rs | rt | rd | 00000 | 100110 | 寄存器异或 |
| SLL | 000000 | 00000 | rt | rd | sa | 000000 | 左移 |
| SRL | 000000 | 00000 | rt | rd | sa | 000010 | 逻辑右移 |
| SRA | 000000 | 00000 | rt | rd | sa | 000011 | 算术右移 |
| JR | 000000 | rs | 00000 | 00000 | 00000 | 001000 | 寄存器跳转 |
| ADDI | 001000 | rs | rt | immediate | | | 立即数加 |
| ANDI | 001100 | rs | rt | immediate | | | 立即数与 |
| ORI | 001101 | rs | rt | immediate | | | 立即数或 |
| XORI | 001110 | rs | rt | immediate | | | 立即数异或 |
| LW | 100011 | rs | rt | immediate | | | 取整数数据字 |
| SW | 101011 | rs | rt | immediate | | | 存整数数据字 |
| BEQ | 000100 | rs | rt | immediate | | | 相等移位 |
| BNE | 000101 | rs | rt | immediate | | | 不等移位 |
| LUI | 001111 | 00000 | rt | immediate | | | 设置高位 |
| J | 000010 | address | | | | | 跳转 |
| JAL | 000011 | address | | | | | 调用 |

**2.3 MIPS指令解析**

add/sub/and/or/xor rd,rs,rt # rd <- rs op rt

rs和rt为两个源操作数的寄存器号，rd为目的寄存器号，op为将rs和rt寄存器内的值进行运算的运算符号。

sll/srl/sra rd,rt,sa # rd <- rt shift sa

这三条为移位指令，5位的ds指定移位的位数。

lui rt,imm # rt <- imm << 16

将16位立即数imm左移16位存入rt寄存器中，即把16位立即数imm存入寄存器的高16位。

addi rt,rs,imm # rt <- rs + imm

将16位立即数imm扩展到32位并与rs相加存到rd中。

andi/ori/xori rt,rs,imm # rt <- rs op imm

将rs寄存器内的值与imm做op操作存入rt中。

lw rt,offset(rs) # rt <- Mem[rs+offset]

寄存器rs的内容与offset相加得到存储器地址，将存储器中取出的数据存入rt寄存器中。

sw rt,offset(rs) # Mem[rs+offset] <- rt

将rt寄存器的内容存入rs寄存器的内容与offset相加所得到的存储器地址中。

beq rs,rt,label # if(rs==rt) PC <- label

当rs寄存器和rt寄存器的内容相同时，跳转到label。

bne rs,rt,label # if(rs!=rt) PC <- label

当rs寄存器和rt寄存器的内容不同时，跳转到label。

j target # PC <- target

无条件跳转到target。

jal target # r31<- PC + 8; PC <- target

首先将返回地址保存在约定好的r31中，然后跳转到target。

jr rs # PC <- rs

将rs寄存器的内容写入PC中，待本条指令执行结束后会跳转到rs寄存器内容的地址中。

# 实验原理

**3.1 指令周期**

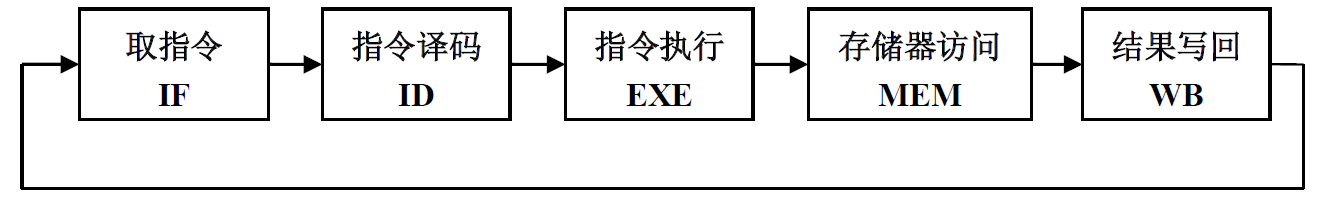
单周期CPU用一个时钟周期执行一条指令，而确定时钟周期的时间长度时要考虑执行时间最长的指令，因此一旦时钟频率确定之后，一个时钟周期的长度也就固定了。因此不管每条指令的复杂程度如何，单周期CPU都花费相同的时间去执行每条指令，就会造成资源浪费。而多周期CPU的中心思想是将整个CPU的执行过程分成若干个小周期，每个小周期只用一个时钟去完成，再开始下一条指令的执行。每种指令的复杂程度不同，所以执行的指令周期也不同。

|  |  |
| --- | --- |
| 指令 | 周期数 |
| ADD | 4 |
| SUB | 4 |
| AND | 4 |
| OR | 4 |
| XOR | 4 |
| SLL | 4 |
| SRL | 4 |
| SRA | 4 |
| JR | 2 |
| ADDI | 4 |
| ANDI | 4 |
| ORI | 4 |
| XORI | 4 |
| LW | 5 |
| SW | 4 |
| BEQ | 3 |
| BNE | 3 |
| LUI | 4 |
| J | 2 |
| JAL | 2 |

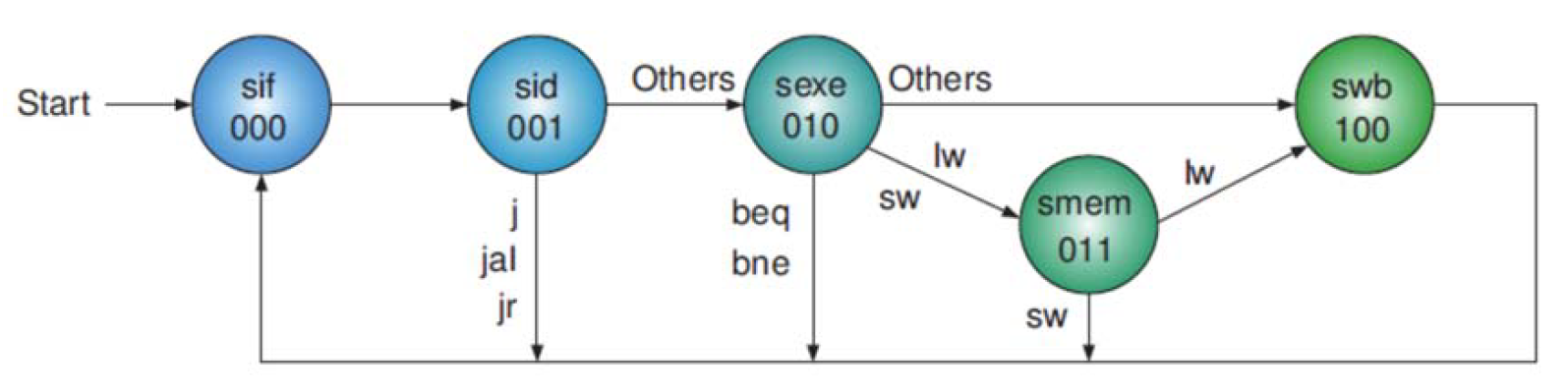
CPU在执行指令时，一般要经过以下阶段：

1. 取指令（IF）：根据程序计数器PC中的指令地址从存储器中取出一条指令放入指令寄存器IR，同时PC根据指令字长自动递增产生下一条指令所需要的指令地址。如果遇到转移指令，则控制器把转移地址送入PC。
2. 指令译码（ID）：J,JAL,JR指令在指令译码周期会修改PC的值以达到跳转的目的。除了J,JAL,JR指令，其他指令在指令译码阶段对取指令操作中得到的指令进行分析并译码，确定该指令需要完成的操作，产生相应的控制信号，用于驱动执行状态中的各种操作。
3. 指令执行（EXE）：BEQ,BNE指令在指令执行周期将判断两个寄存器的内容是否相等，根据判断结果跳转或继续执行。其他指令根据指令译码器得到的操作控制信号具体地执行指令动作，再转移到结果写回状态。
4. 存储器访问（MEM）：只有LW和SW指令会进入存储器访问周期。在该阶段，根据存放在寄存器中的存储器地址，LW指令从存储器读取数据，而SW指令往存储器中存入数据。
5. 结果写回（WB）：指令执行的结果或访问存储器中得到的数据写回相应的寄存器中。

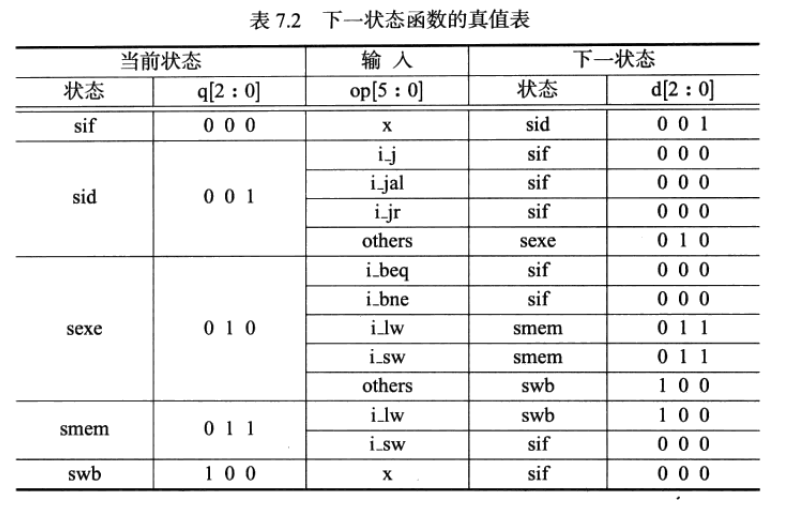
实验按照这5个阶段设计，每条指令结束后会回到取指令阶段，以此循环执行指令。如图：



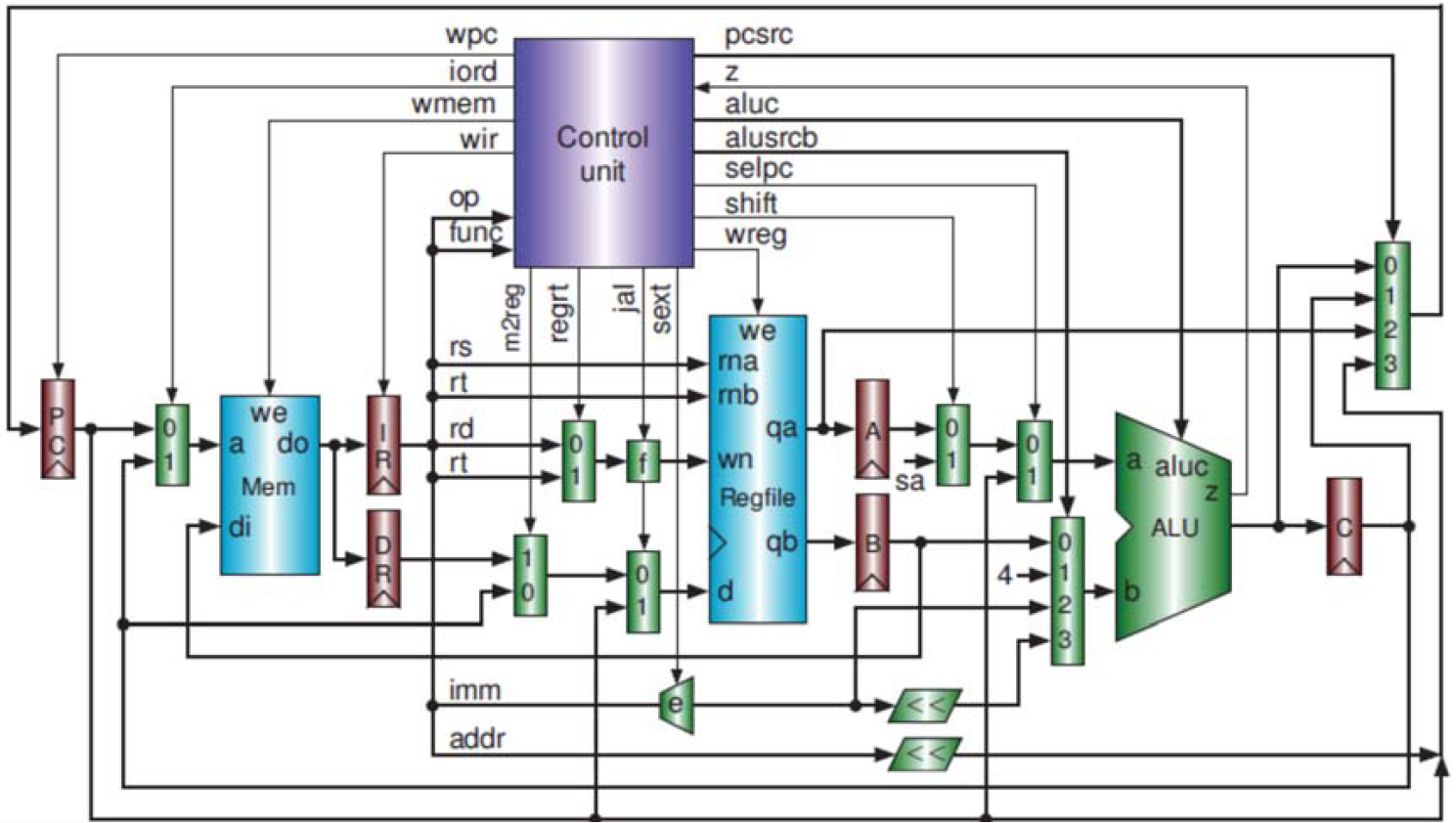
每个状态代表一个周期，一共有5个状态，所以一条指令最多可以使用5个周期去实现功能，但最少需要2个周期去实现功能。IF状态跳转到ID状态是无条件的，其他状态之间的转换都是有条件的。EXE状态可以跳转到MEM，也可以跳转到WB，跳转到哪个状态由指令功能决定，即指令操作码决定。如图：



下一状态的真值表：



**3.3 MIPS多周期CPU控制部件电路结构**

上图为一个多周期CPU上完成MIPS指令所需要的数据通路以及控制线路图。其中，指令和数据存储在存储器中的不同位置。

控制信号的含义：



控制信号真值表：



算术逻辑单元运算功能表：

|  |  |
| --- | --- |
| aluc[3:0] | 功能 |
| x000 | ADD |
| x100 | SUB |
| x001 | AND |
| x101 | OR |
| x010 | XOR |
| x110 | LUI |
| 0011 | SLL |
| 0111 | SRL |
| 1111 | SRA |

# 实验仪器

个人电脑，Xilinx Vivado2019.2。

# 实验过程和结果

* 1. **主要模块**

首先将CPU需要的各个模块分开设计，最后用一个顶层模块将它们连起来。具体模块如下：

1. mccomp顶层模块：将其他模块连接起来，构成CPU。
2. mccpu模块：结构描述风格的多周期CPU。
3. control\_unit控制单元：负责根据指令来决定输出控制其他各个单元的控制信号。
4. PC程序计数器：用于存放当前执行指令的地址。
5. ir指令寄存器：解析指令并送往相应寄存器。
6. dr数据寄存器：将Mem中输出的数据保存起来。
7. ra：保存将要参与ALU计算的第一个操作数。
8. rb：保存将要参与ALU计算的第二个操作数。
9. rc：保存ALU计算出的数据。
10. reg\_wn：选择rt或rd。
11. mem\_address：数据和指令的选择器。
12. result：运算结果的选择器。
13. link：选择PC或J跳转指令。
14. oprand\_a：操作数的选择器。
15. alu\_a：ALU第一个输入端的选择器。
16. alu\_b：ALU第二个输入端的选择器。
17. nextpc：下一个PC的选择器。
18. rf：寄存器堆，存储程序所需要的临时数据和中间结果。
19. aluunit算术逻辑单元：用于算术逻辑运算，根据控制信号从输入的数据中选取操作数再输出结果。
20. memory：存储指令和数据。
    1. **模块代码分析**
21. mccomp：

代码如下：

`timescale 1ns / 1ps

module mccomp (clock,resetn,q,a,b,alu,adr,tom,fromm,pc,ir,mem\_clk);

input clock, resetn, mem\_clk;

output [31:0] a,b,alu, adr, tom, fromm,pc, ir;

output [2:0] q;

wire wmem;

mccpu mc\_cpu (clock, resetn, fromm,pc, ir,a,b, alu, wmem, adr, tom,q) ;

mcmem memory (clock, fromm, tom, adr , wmem, mem\_clk, mem\_clk) ;

endmodule

1. mccpu：

代码如下：

`timescale 1ns / 1ps

module mccpu (clock, resetn, frommem, pc, inst, alua, alub, alu, wmem, madr, tomem,state);

input [31:0] frommem;

input clock, resetn;

output [31:0] pc, inst,alua, alub, alu, madr, tomem;

output [2:0] state;

output wmem;

wire [3:0] aluc;

wire [4:0] reg\_dest;

wire z, wpc, wir, wmem, wreg, iord, regrt, m2reg, shift, selpc, jal, sext;

wire [31:0] npc, rega, regb, regc, mem, qa, qb, res, opa,bra, alub, alu\_mem;

wire [1:0] alusrcb, pcsource;

wire [31:0] sa = {27'b0, inst [10:6]};

wire [31:0] jpc={pc[31:28],inst[25:0],1'b0,1'b0};

mccu control\_unit(inst[31:26], inst[5:0], z,clock, resetn,wpc,wir, wmem, wreg, iord, regrt, m2reg, aluc,

shift,selpc,alusrcb,pcsource,jal,sext,state);

wire e = sext & inst[15];

wire [15:0] imm = {16{e}};

wire [31:0] immediate = {imm, inst[15:0]};

wire [31:0] offset = {imm[13:0], inst[15:0],1'b0,1'b0};

dffe32 ip (npc, clock, resetn, wpc,pc);

dffe32 ir (frommem, clock, resetn,wir, inst);

dff32 dr (frommem, clock, resetn, mem);

dff32 ra (qa,clock, resetn, rega);

dff32 rb (qb, clock, resetn, regb);

dff32 rc (alu, clock, resetn, regc);

assign tomem = regb;

mux2x5 reg\_wn (inst[15:11], inst [20:16], regrt, reg\_dest);

wire[4:0] wn = reg\_dest | {5{jal}}; // jal: r31 <-- p4;

mux2x32 mem\_address (pc, regc, iord, madr);

mux2x32 result (regc , mem, m2reg,alu\_mem);

mux2x32 link(alu\_mem, pc, jal, res);

mux2x32 oprand\_a (rega,sa, shift, opa);

mux2x32 alu\_a (opa,pc,selpc,alua);

mux4x32 alu\_b (regb, 32'h4, immediate, offset, alusrcb, alub) ;

mux4x32 nextpc (alu, regc, qa, jpc, pcsource,npc); // next pc

regfile rf (inst [25:21], inst [20:16], res, wn, wreg, clock, resetn, qa,qb) ;

alu aluunit (alua,alub, aluc,alu,z) ;

endmodule

1. Control Unit：

代码如下：

`timescale 1ns / 1ps

module mccu (op,func,z,clock,resetn,wpc,wir,wmem,wreg,iord,regrt,m2reg,aluc,shift

,alusrca,alusrcb,pcsource,jal,sext,state);

input [5:0] op,func;

input z,clock,resetn;

output reg wpc,wir,wmem,wreg,iord,regrt,m2reg;

output reg [3:0] aluc;

output reg [1:0] alusrcb,pcsource;

output reg shift,alusrca,jal,sext;

output reg [2:0] state;

reg [2:0] next\_state;

parameter [2:0] sif = 3'b000,

sid = 3'b001,

sexe = 3'b010,

smem = 3'b011,

swb = 3'b100;

wire r\_type,i\_add,i\_sub,i\_and,i\_or,i\_xor,i\_sll,i\_srl,i\_sra,i\_jr;

wire i\_addi,i\_andi,i\_ori,i\_xori,i\_lw,i\_sw,i\_beq,i\_bne,i\_lui,i\_j,i\_jal;

and(r\_type,~op[5],~op[4],~op[3],~op[2],~op[1],~op[0]);

and(i\_add,r\_type,func[5],~func[4],~func[3],~func[2],~func[1],~func[0]);

and(i\_sub,r\_type,func[5],~func[4],~func[3],~func[2],func[1],~func[0]);

and(i\_and,r\_type,func[5],~func[4],~func[3],func[2],~func[1],~func[0]);

and(i\_or,r\_type,func[5],~func[4],~func[3],func[2],~func[1],func[0]);

and(i\_xor,r\_type,func[5],~func[4],~func[3],func[2],func[1],~func[0]);

and(i\_sll,r\_type,~func[5],~func[4],~func[3],~func[2],~func[1],~func[0]);

and(i\_srl,r\_type,~func[5],~func[4],~func[3],~func[2],func[1],~func[0]);

and(i\_sra,r\_type,~func[5],~func[4],~func[3],~func[2],func[1],func[0]);

and(i\_jr,r\_type,~func[5],~func[4],func[3],~func[2],~func[1],~func[0]);

and(i\_addi,~op[5],~op[4],op[3],~op[2],~op[1],~op[0]);

and(i\_andi,~op[5],~op[4],op[3],op[2],~op[1],~op[0]);

and(i\_ori,~op[5],~op[4],op[3],op[2],~op[1],op[0]);

and(i\_xori,~op[5],~op[4],op[3],op[2],op[1],~op[0]);

and(i\_lw,op[5],~op[4],~op[3],~op[2],op[1],op[0]);

and(i\_sw,op[5],~op[4],op[3],~op[2],op[1],op[0]);

and(i\_beq,~op[5],~op[4],~op[3],op[2],~op[1],~op[0]);

and(i\_bne,~op[5],~op[4],~op[3],op[2],~op[1],op[0]);

and(i\_lui,~op[5],~op[4],~op[3],op[2],op[1],op[0]);

and(i\_j,~op[5],~op[4],~op[3],~op[2],op[1],~op[0]);

and(i\_jal,~op[5],~op[4],~op[3],~op[2],op[1],op[0]);

and(i\_bgezal,op[5],~op[4],~op[3],~op[2],~op[1],op[0]);

wire i\_shift;

or(i\_shift,i\_sll,i\_srl,i\_sra);

always @ \* begin

wpc = 0;

wir = 0;

wmem = 0;

wreg = 0;

iord = 0;

aluc = 4'bx000;

alusrca = 0;

alusrcb = 2'h0;

regrt = 0;

m2reg = 0;

shift = 0;

pcsource = 2'h0;

jal = 0;

sext = 1;

case (state)

sif:begin

wpc = 1;

wir = 1;

alusrca = 1;

alusrcb = 2'h1;

next\_state = sid;

end

sid:begin

if(i\_j)begin

pcsource = 2'h3;

wpc = 1;

next\_state = sif;

end

else if(i\_jal)begin

pcsource = 2'h3;

wpc = 1;

jal = 1;

wreg = 1;

next\_state = sif;

end

else if(i\_jr)begin

pcsource = 2'h2;

wpc = 1;

next\_state = sif;

end

else begin

aluc = 4'bx000;

alusrca = 1;

alusrcb = 2'h3;

next\_state = sexe;

end

end

sexe:begin

aluc[3] = i\_sra;

aluc[2] = i\_sub | i\_or | i\_srl | i\_sra | i\_ori | i\_lui;

aluc[1] = i\_xor | i\_sll| i\_srl | i\_sra | i\_xori| i\_beq | i\_bne | i\_lui;

aluc[0] = i\_and | i\_or | i\_sll | i\_srl | i\_sra | i\_andi| i\_ori;

if(i\_beq || i\_bne) begin

pcsource = 2'h1;

wpc = i\_beq & z | i\_bne & ~z;

next\_state = sif;

end

else begin

if(i\_lw || i\_sw) begin

alusrcb = 2'h2;

next\_state = smem;

end

else begin

if(i\_shift) shift = 1;

if(i\_addi || i\_andi || i\_ori || i\_xori ||i\_lui)

alusrcb = 2'h2;

if(i\_andi || i\_ori || i\_xori)

sext=0;

next\_state = swb;

end

end

end

smem:begin

iord = 1;

if(i\_lw) begin

next\_state = swb;

end else begin

wmem = 1;

next\_state = sif;

end

end

swb:begin

if(i\_lw) m2reg = 1;

if(i\_lw || i\_addi || i\_andi || i\_ori || i\_xori ||i\_lui)

regrt = 1;

wreg = 1;

next\_state = sif;

end

default:begin

next\_state = sif;

end

endcase

end

always @ (posedge clock or negedge resetn)begin

if(resetn == 0) begin

state <= sif;

end

else begin

state <= next\_state;

end

end

endmodule

1. PC：

代码如下：

`timescale 1ns / 1ps

module dffe32 (d,clk,clrn,e,q);

input [31:0]d;

input clk,clrn,e;

output [31:0]q;

reg [31:0]q;

always @ (negedge clrn or posedge clk)

if (clrn == 0) q<= 0;

else if (e) q<= d;

endmodule

1. ir：

代码如下：

`timescale 1ns / 1ps

module dffe32 (d,clk,clrn,e,q);

input [31:0]d;

input clk,clrn,e;

output [31:0]q;

reg [31:0]q;

always @ (negedge clrn or posedge clk)

if (clrn == 0) q<= 0;

else if (e) q<= d;

endmodule

1. dr：

代码如下：

`timescale 1ns / 1ps

module dff32 (d,clk,clrn,q);

input [31:0]d;

input clk,clrn;

output [31:0]q;

reg [31:0]q;

always @ (posedge clk or negedge clrn) begin

if (clrn == 0) q <= 0;

else q <= d;

end

endmodule

1. ra：

代码如下：

`timescale 1ns / 1ps

module dff32 (d,clk,clrn,q);

input [31:0]d;

input clk,clrn;

output [31:0]q;

reg [31:0]q;

always @ (posedge clk or negedge clrn) begin

if (clrn == 0) q <= 0;

else q <= d;

end

endmodule

1. rb：

代码如下：

`timescale 1ns / 1ps

module dff32 (d,clk,clrn,q);

input [31:0]d;

input clk,clrn;

output [31:0]q;

reg [31:0]q;

always @ (posedge clk or negedge clrn) begin

if (clrn == 0) q <= 0;

else q <= d;

end

endmodule

1. rc：

代码如下：

`timescale 1ns / 1ps

module dff32 (d,clk,clrn,q);

input [31:0]d;

input clk,clrn;

output [31:0]q;

reg [31:0]q;

always @ (posedge clk or negedge clrn) begin

if (clrn == 0) q <= 0;

else q <= d;

end

endmodule

1. reg\_wn：

代码如下：

`timescale 1ns / 1ps

module mux2x5 (a0, a1,s,y);

input [4:0] a0, a1;

input s;

output [4:0] y;

assign y =s ? a1 : a0;

endmodule

1. mem\_address：

代码如下：

`timescale 1ns / 1ps

module mux2x32 (a0, a1,s,y);

input [31:0] a0, a1;

input s;

output [31:0] y;

assign y =s ? a1 : a0;

endmodule

1. result：

代码如下：

`timescale 1ns / 1ps

module mux2x32 (a0, a1,s,y);

input [31:0] a0, a1;

input s;

output [31:0] y;

assign y =s ? a1 : a0;

endmodule

1. link：

代码如下：

`timescale 1ns / 1ps

module mux2x32 (a0, a1,s,y);

input [31:0] a0, a1;

input s;

output [31:0] y;

assign y =s ? a1 : a0;

endmodule

1. oprand\_a：

代码如下：

`timescale 1ns / 1ps

module mux2x32 (a0, a1,s,y);

input [31:0] a0, a1;

input s;

output [31:0] y;

assign y =s ? a1 : a0;

endmodule

1. alu\_a：

代码如下：

`timescale 1ns / 1ps

module mux2x32 (a0, a1,s,y);

input [31:0] a0, a1;

input s;

output [31:0] y;

assign y =s ? a1 : a0;

endmodule

1. alu\_b：

代码如下：

`timescale 1ns / 1ps

module mux4x32 (a0 , a1,a2,a3,s,y);

input [31:0] a0, a1, a2,a3;

input [1:0] s;

output [31:0] y;

function [31:0] select;

input [31:0] a0, a1, a2,a3;

input [1:0] s;

case (s )

2'b00: select= a0;

2'b01: select = a1;

2'b10: select = a2;

2'b11: select = a3;

endcase

endfunction

assign y=select(a0,a1,a2,a3,s);

endmodule

1. nextpc：

代码如下：

`timescale 1ns / 1ps

module mux4x32 (a0 , a1,a2,a3,s,y);

input [31:0] a0, a1, a2,a3;

input [1:0] s;

output [31:0] y;

function [31:0] select;

input [31:0] a0, a1, a2,a3;

input [1:0] s;

case (s )

2'b00: select= a0;

2'b01: select = a1;

2'b10: select = a2;

2'b11: select = a3;

endcase

endfunction

assign y=select(a0,a1,a2,a3,s);

endmodule

1. rf：

代码如下：

`timescale 1ns / 1ps

module regfile (rna,rnb,d,wn,we,clk,clrn, qa, qb);

input [4:0] rna,rnb,wn;

input [31:0] d;

input we,clk,clrn;

output [31:0] qa, qb;

reg [31:0] register [1:31]; // 31 x 32-bit regs

integer i;

// 2 read ports

assign qa = (rna == 0) ? 0 : register[rna];

assign qb = (rnb == 0) ? 0 : register[rnb];

// 1 write port

always @(posedge clk or negedge clrn)

if(clrn==0) begin

for (i=1; i<32; i=i+1)

register[i]<=0;

end else if ( (wn != 0)&& we)

register[wn]<= d;

endmodule

1. aluunit：

代码如下：

`timescale 1ns / 1ps

module alu(a,b,aluc,r,z);

input [31:0] a,b;

input [3:0] aluc;

output [31:0] r;

output z;

assign r = cal (a,b,aluc);

assign z = ~|r;

function [31:0] cal;

input [31:0] a,n;

input [3:0] aluc;

casex (aluc)

4'bx000: cal = a+b;

4'bx100: cal = a-b;

4'bx001: cal = a&b;

4'bx101: cal = a|b;

4'bx010: cal = a^b;

4'bx110: cal = {b[15:0],16'h0};

4'bx011: cal = b<<a[4:0];

4'b0111: cal = b>>a[4:0];

4'b1111: cal = $signed(b) >>> a[4:0];

endcase

endfunction

endmodule

1. memory：

代码如下：

`timescale 1ns / 1ps

module mcmem(clk,dataout,datain,addr,we,inclk,outclk);

input [31:0] datain;

input [31:0] addr;

input clk,we,inclk,outclk;

output [31:0] dataout;

reg [31:0] ram [0:63];

initial begin

$readmemh("test.mem",ram);

end

assign dataout = ram[addr[8:2]];

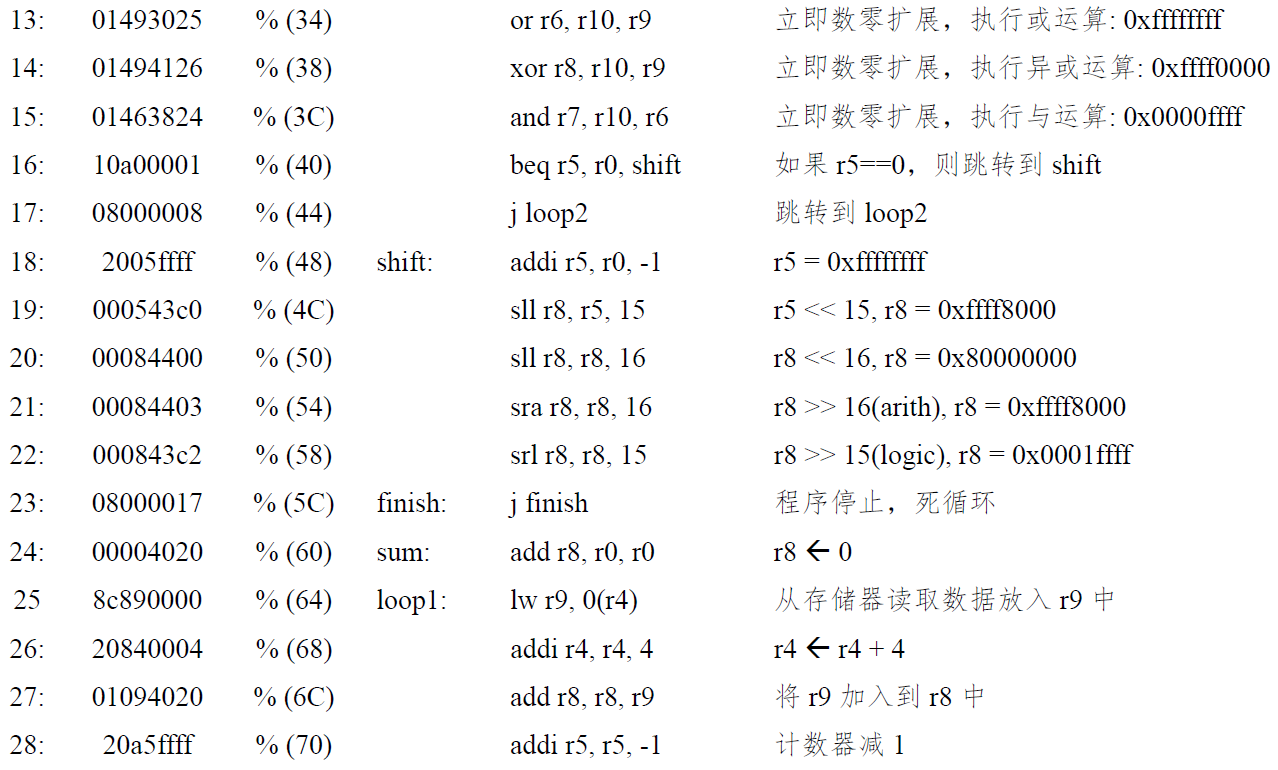
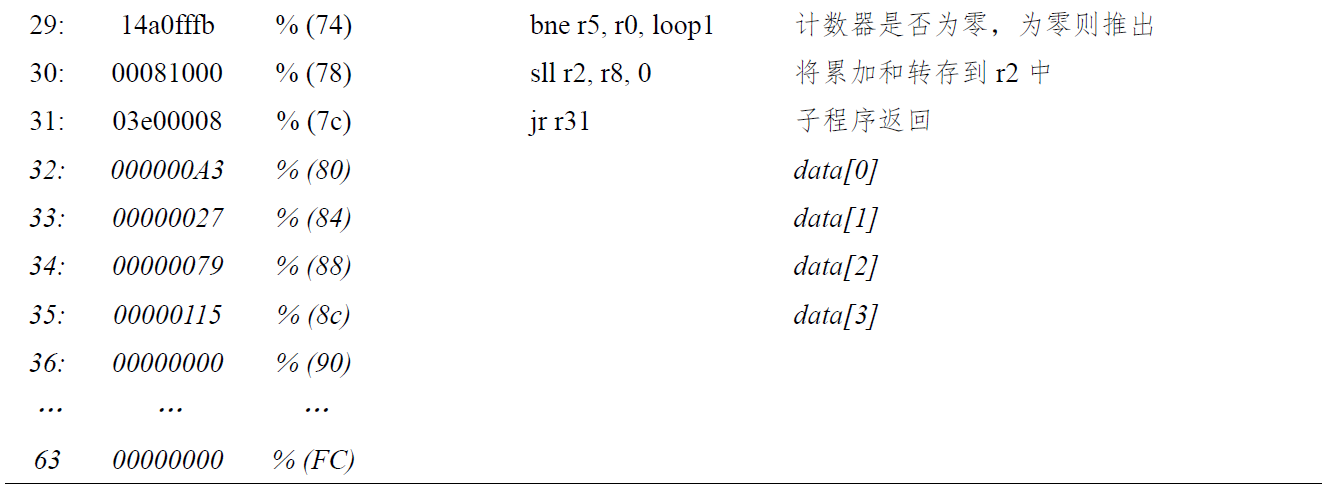
always @ (posedge clk) begin

if (we) ram[addr] = datain;

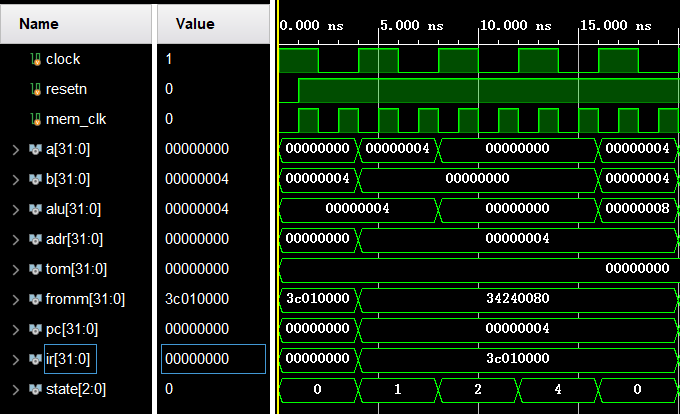
end

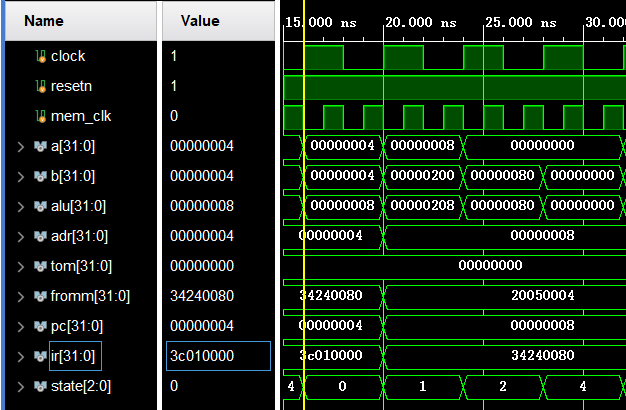
endmodule

* 1. **测试代码**

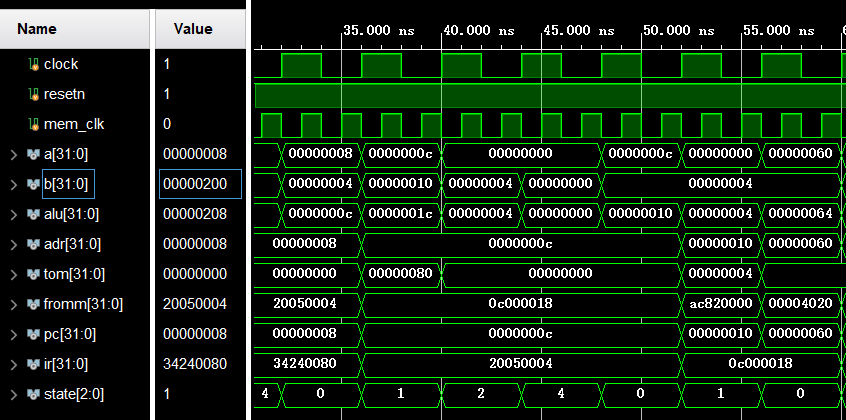
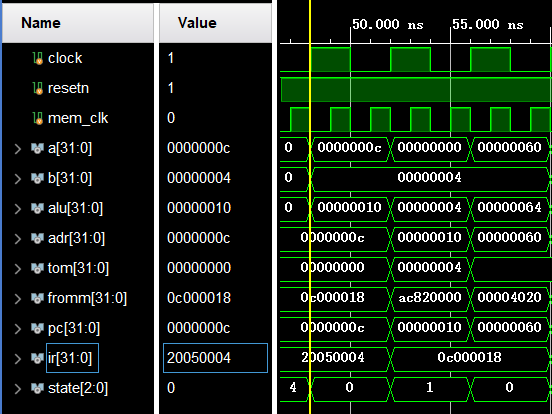
以下为测试代码，将根据测试代码对各种类型的指令作出分析。

* 1. **测试代码分析**

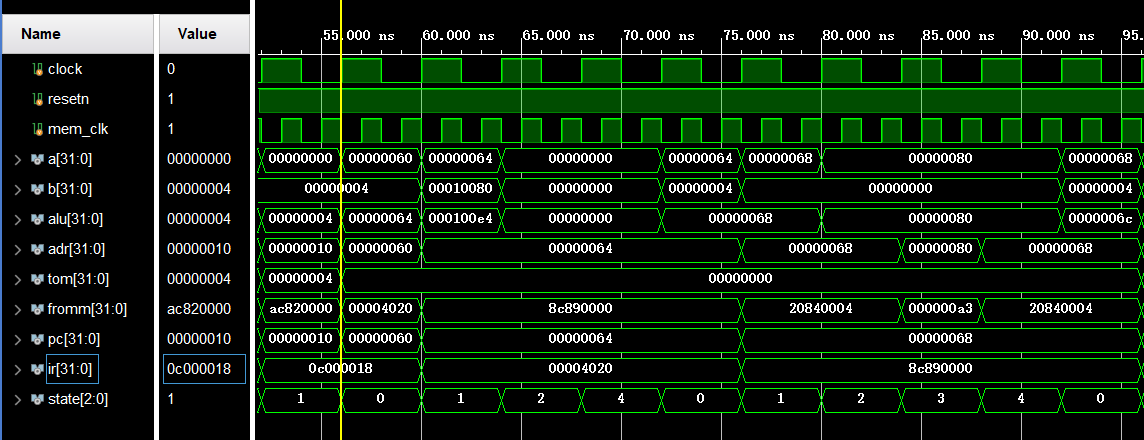
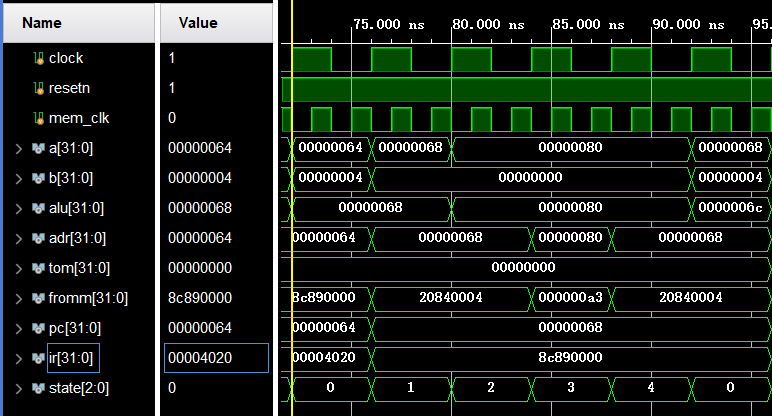
下面根据模拟波形对不同类型的指令在不同阶段的内容进行分析。

上图指令为：lui r1,0，需要4个周期运行，分别为IF，ID，EXE和WB。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将立即数存入ALU。在EXE阶段对ID阶段存入ALU的值进行移位运算。在WB阶段写回r1中。

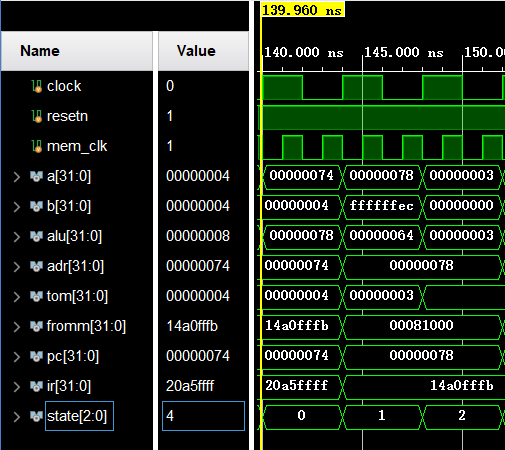
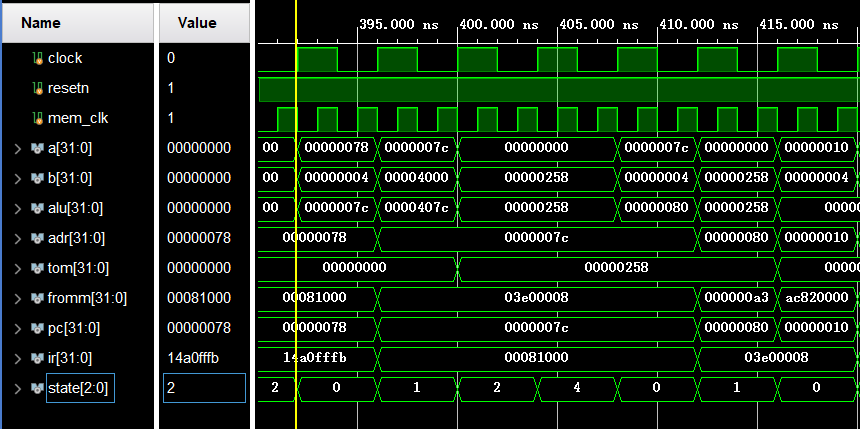
上图指令为：ori r4,r1,0x80，需要4个周期，分别为IF，ID，EXE和WB。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将r1的值存入寄存器A，将立即数存入寄存器B。在EXE阶段ALU对寄存器A与寄存器B做或运算并将结果存入寄存器C。在WB阶段将寄存器C的内容存入r4。

上图指令为addi r5,r0,4，需要4个周期运行，分别为IF，ID，EXE和WB。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将r0的内容存入寄存器A，将立即数4存入寄存器B。在EXE阶段ALU对存入寄存器A和B的值进行加法运算并存入寄存器C中。在WB阶段将寄存器C的内容存入寄存器r5。

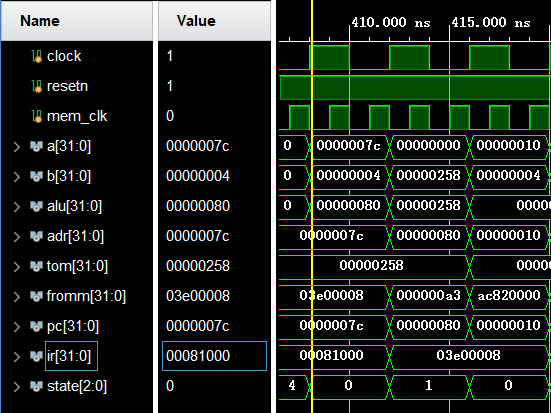
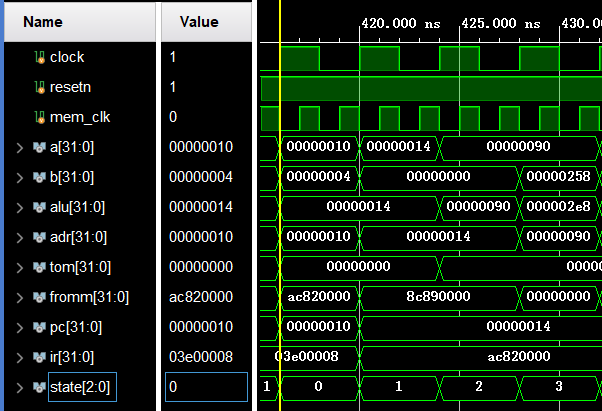
上图指令为jal sum，需要2个周期，分别是IF和ID。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段对指令进行译码，首先将PC的值存入r31，再将PC的前四位，sum的地址（即0x60）与“00”拼接起来存入PC中。

上图指令为add r8,r0,r0，需要4个周期运行，分别为IF，ID，EXE和WB。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将r0分别存入寄存器A，B。在EXE阶段ALU对寄存器A，B的值进行加法运算存入寄存器C中。在WB阶段将寄存器C的内容存入r8中。

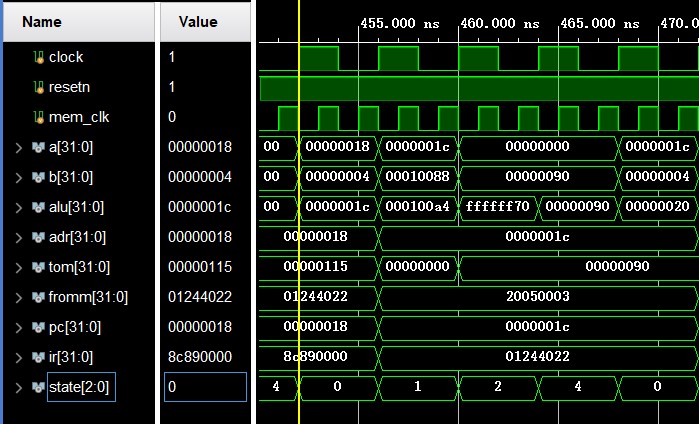
上图指令为lw r9,0(r4)，需要5个周期运行，分别为IF，ID，EXE，MEM和WB。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将r9存入寄存器A，r4存入寄存器B，扩展offset为符号数并左移两位再与PC相加存入寄存器C。在EXE阶段ALU对寄存器B，C进行加法运算并存入寄存器C中。在MEM阶段将寄存器C的内容存入数据寄存器DR。在WB阶段将DR的内容写回r9。

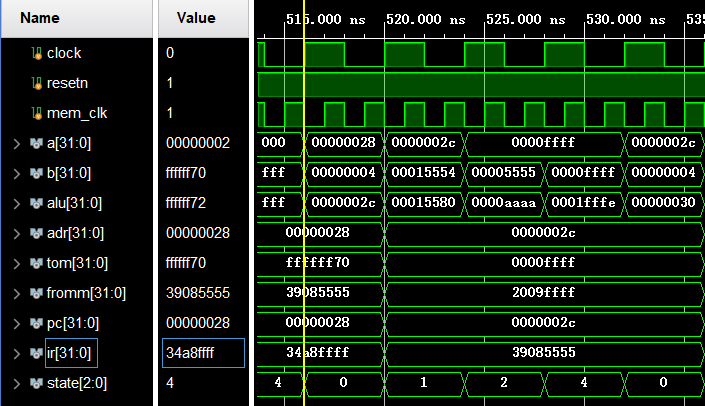
上图指令为bne r5,r0,loop1，需要3个周期运行，分别为IF，ID和EXE。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将r5放入寄存器A，将r0放入寄存器B，扩展loop1的地址为符号数并左移两位再与PC相加存入寄存器C。在EXE阶段ALU对寄存器A，B做减法运算并判断控制信号z是否为0。如果z为0说明寄存器A，B的值不相等，符合条件，将寄存器C的值存入PC，即完成跳转到loop1指向的地址；如果z为1说明寄存器A，B的值相等，不符合条件，不跳转。

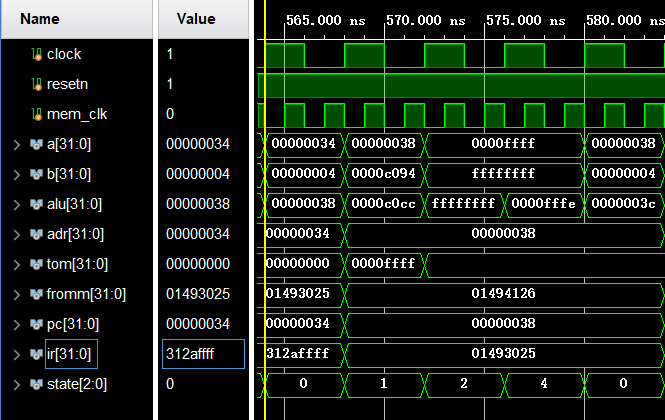
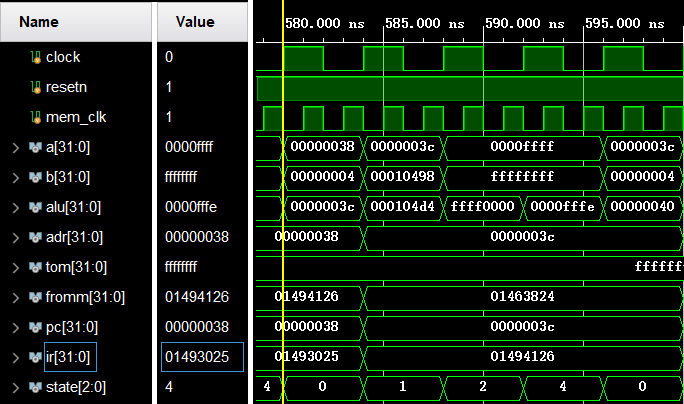
上图指令为sll r2,r8,0，需要4个周期运行，分别为IF，ID，EXE和WB。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将r2存入寄存器A，r8存入寄存器B，扩展0为符号数并左移两位再与PC相加存入寄存器C。在EXE阶段ALU将寄存器B的值左移寄存器C的值位并存入寄存器C中。在WB阶段，将寄存器C的值写回r2。

上图指令为jr r31，需要2个周期运行，分别为IF，ID。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将r31的内容存入PC。

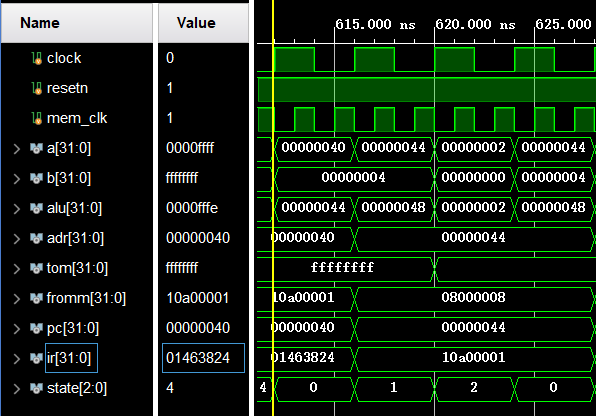
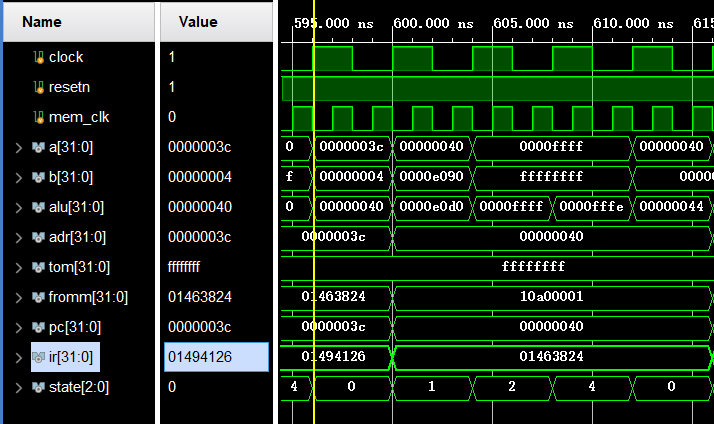
上图指令为sw r2,0(r4)，需要4个周期运行，分别为IF，ID，EXE和MEM。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将r2存入寄存器A，r4存入寄存器B，扩展offset为符号数并左移两位再与PC相加存入寄存器C。在EXE阶段ALU对寄存器B，C进行加法运算并存入寄存器C中。在MEM阶段将寄存器A的内容存入存储器r4所存的地址。

上图指令为sub r8,r9,r4，需要4个周期运行，分别为IF，ID，EXE和WB。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将r9存入寄存器A，将r4存入寄存器B。在EXE阶段ALU对寄存器A与寄存器B做减法运算（A-B，即r9-r4）并存入寄存器C中。在WB阶段将寄存器C的结果写回r8中。

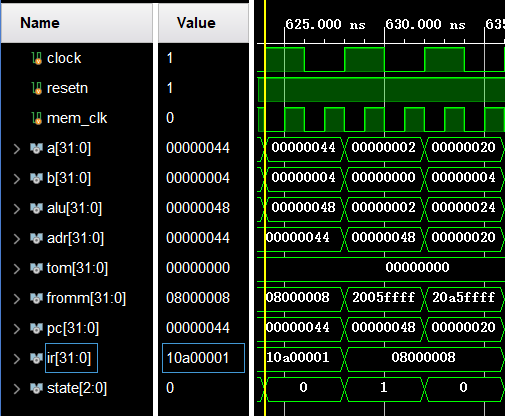
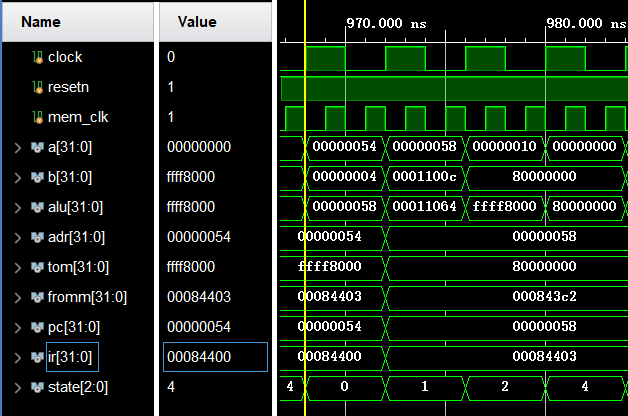
上图指令为xori r8,r8,0x5555，需要4个周期运行，分别为IF，ID，EXE和WB。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将r8的值存入寄存器A，将立即数存入寄存器B。在EXE阶段ALU对寄存器A与寄存器B做异或运算并将结果存入寄存器C。在WB阶段将寄存器C的内容写回r8。

上图指令为or r6,r10,r9，需要4个周期运行，分别为IF，ID，EXE和WB。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将r10存入寄存器A，将R9存入寄存器B。在EXE阶段ALU对寄存器A和寄存器B做或运算并将结果存入寄存器C。在WB阶段将寄存器C的结果存入r6。

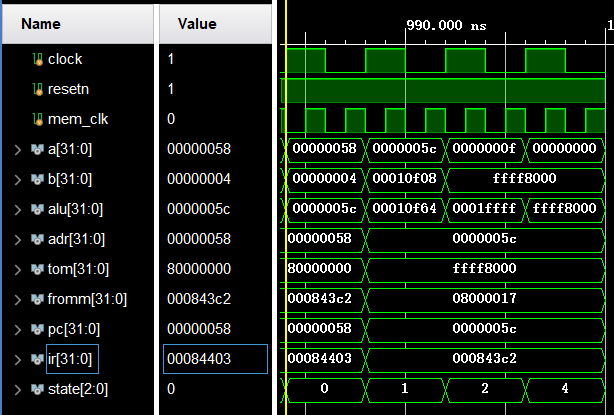
上图指令为xor r8,r10,r9，需要4个周期运行，分别为IF，ID，EXE和WB。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将r10存入寄存器A，将r9存入寄存器B。在EXE阶段ALU对寄存器A和寄存器B做异或运算并将结果存入寄存器C。在WB阶段将寄存器C的结果写回r8。

上图指令为and r7,r10,r6，需要4个周期运行，分别为IF，ID，EXE和WB。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将r10存入寄存器A，r6存入寄存器B。在EXE阶段ALU对寄存器A，B的值进行与运算存入寄存器C中。在WB阶段将寄存器C的内容存入r7中。

上图指令为beq r5,r0,shift，需要3个周期运行，分别为IF，ID和EXE。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将r5放入寄存器A，将r0放入寄存器B，扩展shift的地址为符号数并左移两位再与PC相加存入寄存器C。在EXE阶段ALU对寄存器A，B做减法运算并判断控制信号z是否为1。如果z为1说明寄存器A，B的值相等，符合条件，将寄存器C的值存入PC，即完成跳转到shift所指向的地址；如果z为0说明寄存器A，B的值不相等，不符合条件，所以不跳转。

上图为指令j loop2，需要2个周期运行，分别为IF，ID。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将PC的前4位，loop2的地址（即0x20）与“00”拼接起来存入PC中。

上图指令为sra r8,r8,16，需要4个周期运行，分别为IF，ID，EXE和WB。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将r8分别存入寄存器A，B，扩展16为有符号数并左移两位再与PC相加存入寄存器C。在EXE阶段ALU将寄存器B的值左移寄存器C的值位并存入寄存器C中。在WB阶段，将寄存器C的值写回r8。

上图指令为srl r8,r8,15，需要4个周期运行，分别为IF，ID，EXE和WB。在IF阶段将指令从存储器取出来放入IR指令寄存器，同时将PC+4的结果存入PC。在ID阶段下对指令进行译码，将r8分别存入寄存器A，B，扩展15为有符号数并左移两位再与PC相加存入寄存器C。在EXE阶段ALU将寄存器B的值左移寄存器C的值位并存入寄存器C中。在WB阶段，将寄存器C的值写回r8。

# 心得