北京理工大学
BEIJING INSTITUTE OF TECHNOLOGY

单周期CPU设计

主讲人：陈康冰
计算机学院

单指令周期CPU

所有指令的指令周期都**固定**为一个**统一**的CPU周期。

对一条具体的指令，其指令周期通常被划分为若干机器周期：取指周期、访存周期、执行周期等。

若译码和执行阶段考虑直接使用组合逻辑实现（瞬发），分析各种指令的指令周期：

分支跳转指令(e.g. beq、jal)：**取指周期+译码/执行周期**

运算指令(e.g. add、sub、ori)：**取指周期+译码/执行周期**

访存指令(e.g. lw、sw)：**取指周期+译码/执行周期+访存周期**

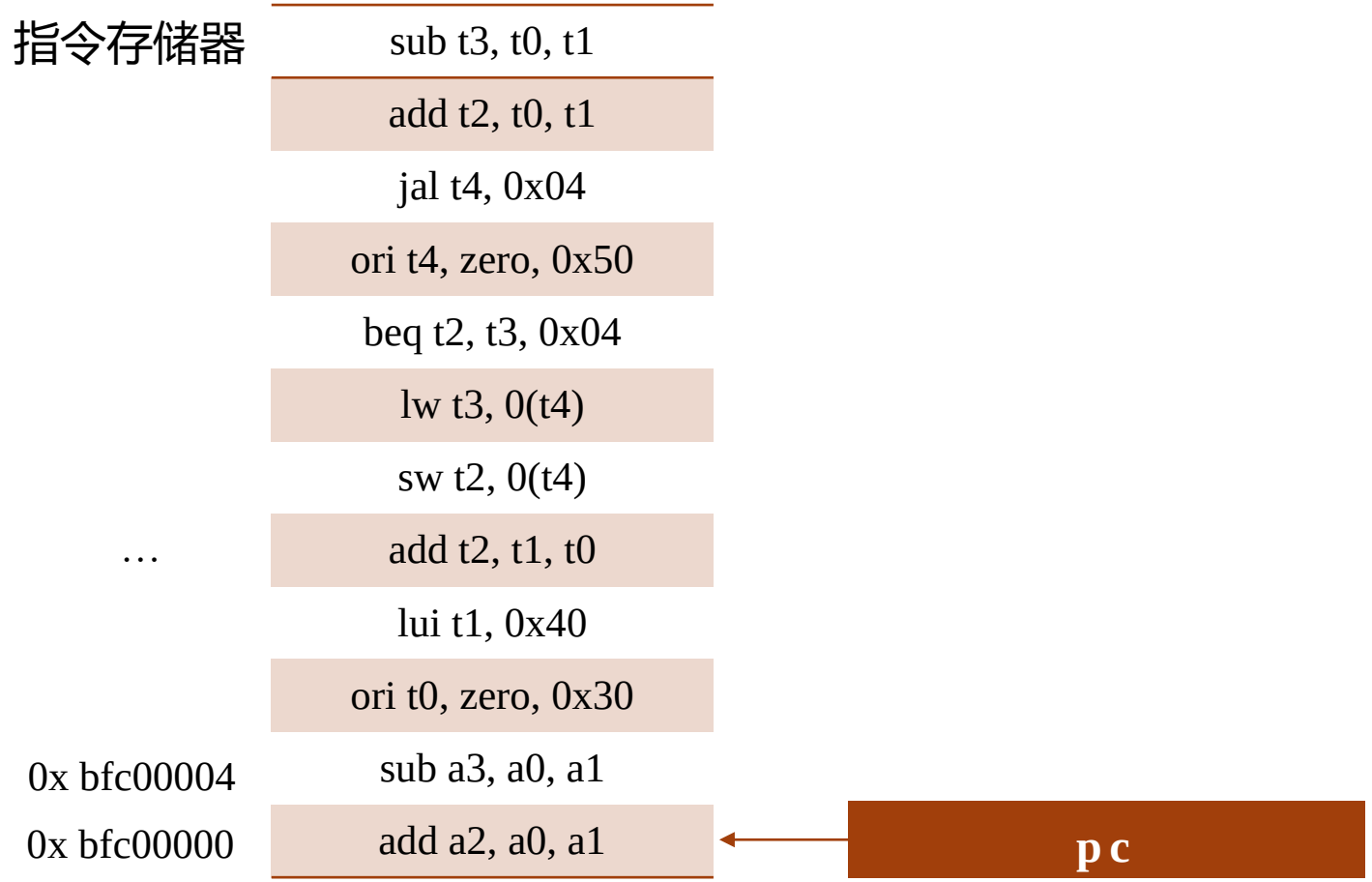
那么这种情况下，我们需要拿最长的指令周期作为统一固定的单周期时间

所以这个统一的**单周期=取指周期+译码/执行周期（1 cycle）+访存周期**。

CPU访问存储器花费的时间比较长，所以这个统一的CPU周期往往受限于**访存时间**。

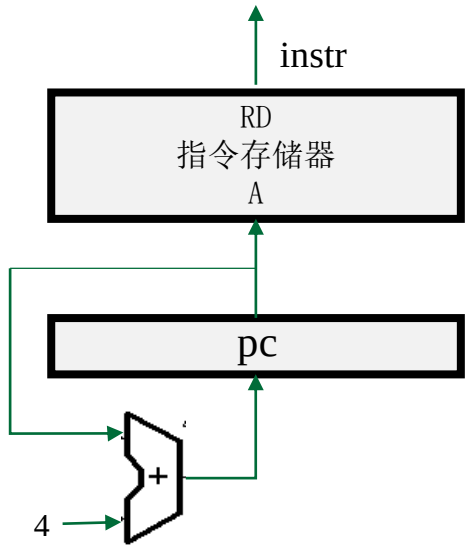
RISC-V指令集系统的一个核心子集

- 算术、逻辑、比较、位移指令
  - add、sub、ori、lui
- 存储访问指令
  - lw (store word)、sw (store word)
- 跳转指令
  - beq (branch equal)、jal (jump and link)



接下来简单设计一个包含这8条指令的单周期CPU。虽然指令集不全，但是在建立数据通道和设计控制部分时用到的关键原理都会得以体现。

PC取指设计

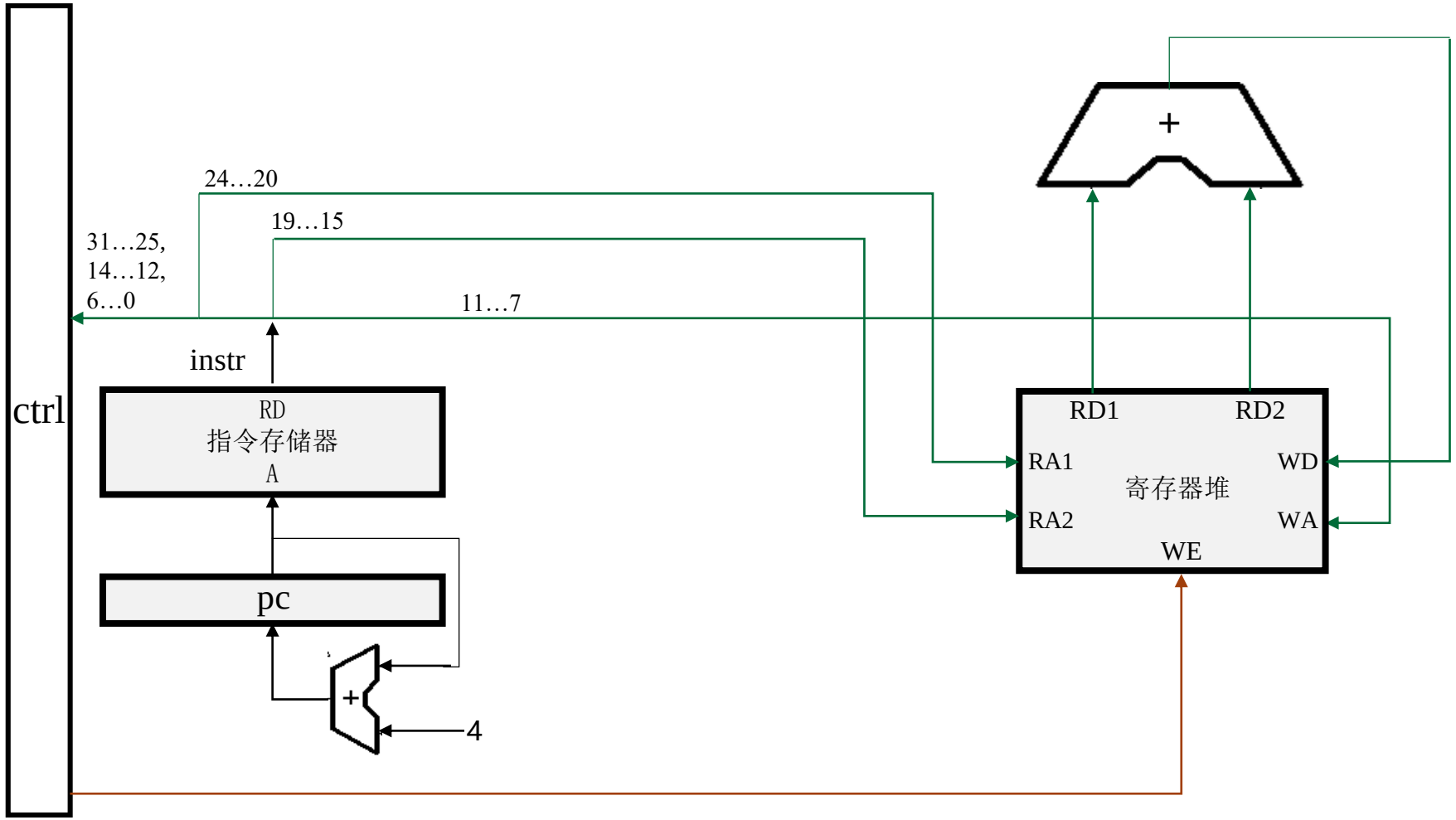


sub t3, t0, t1
add t2, t0, t1
jal t4, 0x04
ori t4, zero, 0x50
beq t2, t3, 0x04
lw t3, 0(t4)
sw t2, 0(t4)
add t2, t1, t0
lui t1, 0x40
ori t0, zero, 0x30
sub a3, a0, a1
add a2, a0, a1

pc

增加ADD指令数据通路

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00000	00	rs2	rs1	000	rd	01100	11



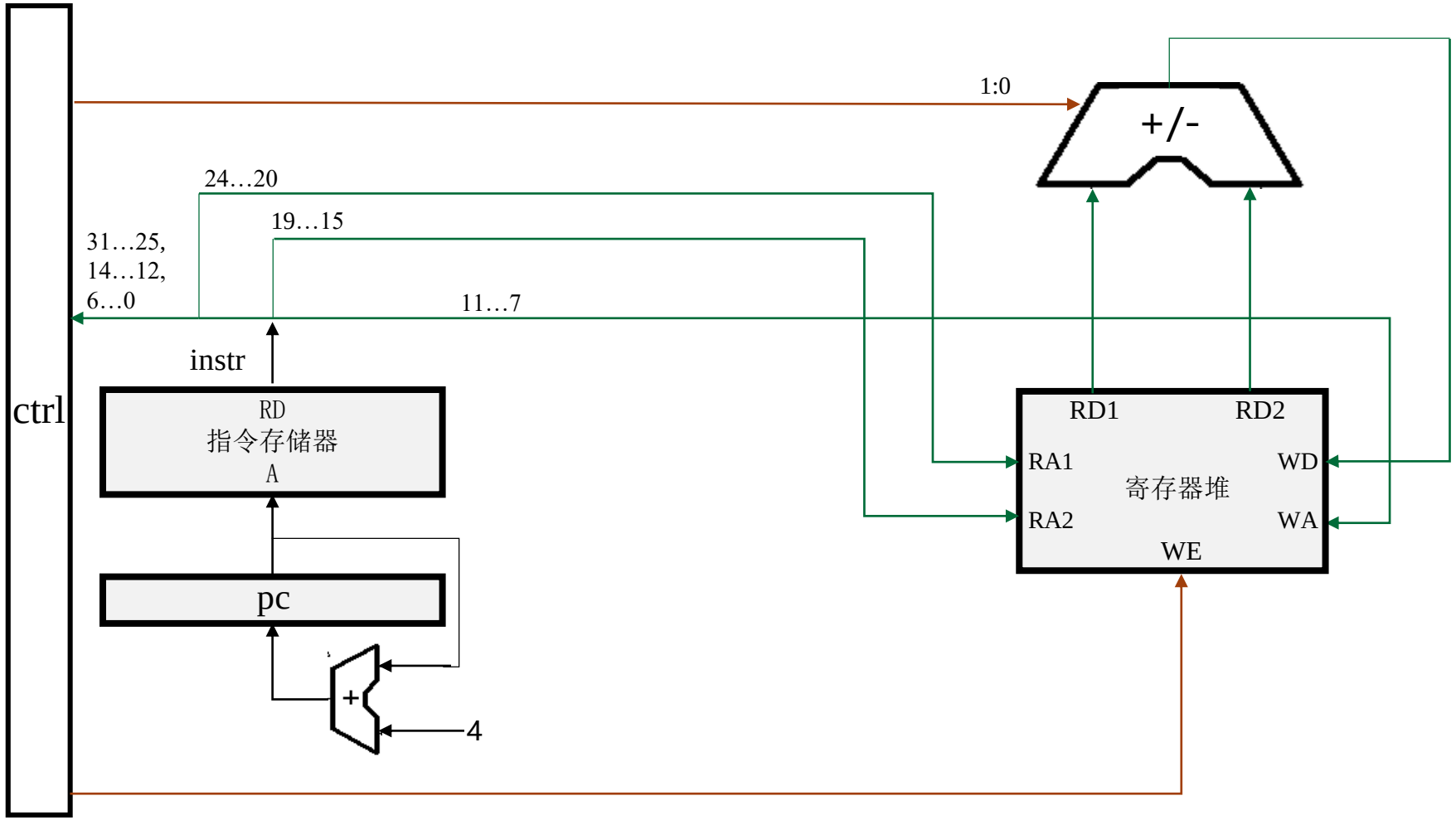
功能描述：将寄存器rs1的值与rs2的值相加，结果写入rd寄存器中。

sub t3, t0, t1
add t2, t0, t1
jal t4, 0x04
ori t4, zero, 0x50
beq t2, t3, 0x04
lw t3, 0(t4)
sw t2, 0(t4)
add t2, t1, t0
lui t1, 0x40
ori t0, zero, 0x30
sub a3, a0, a1
add a2, a0, a1

PC start

新增SUB指令数据通路

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
01000	00	rs2	rs1	000	rd	01100	11



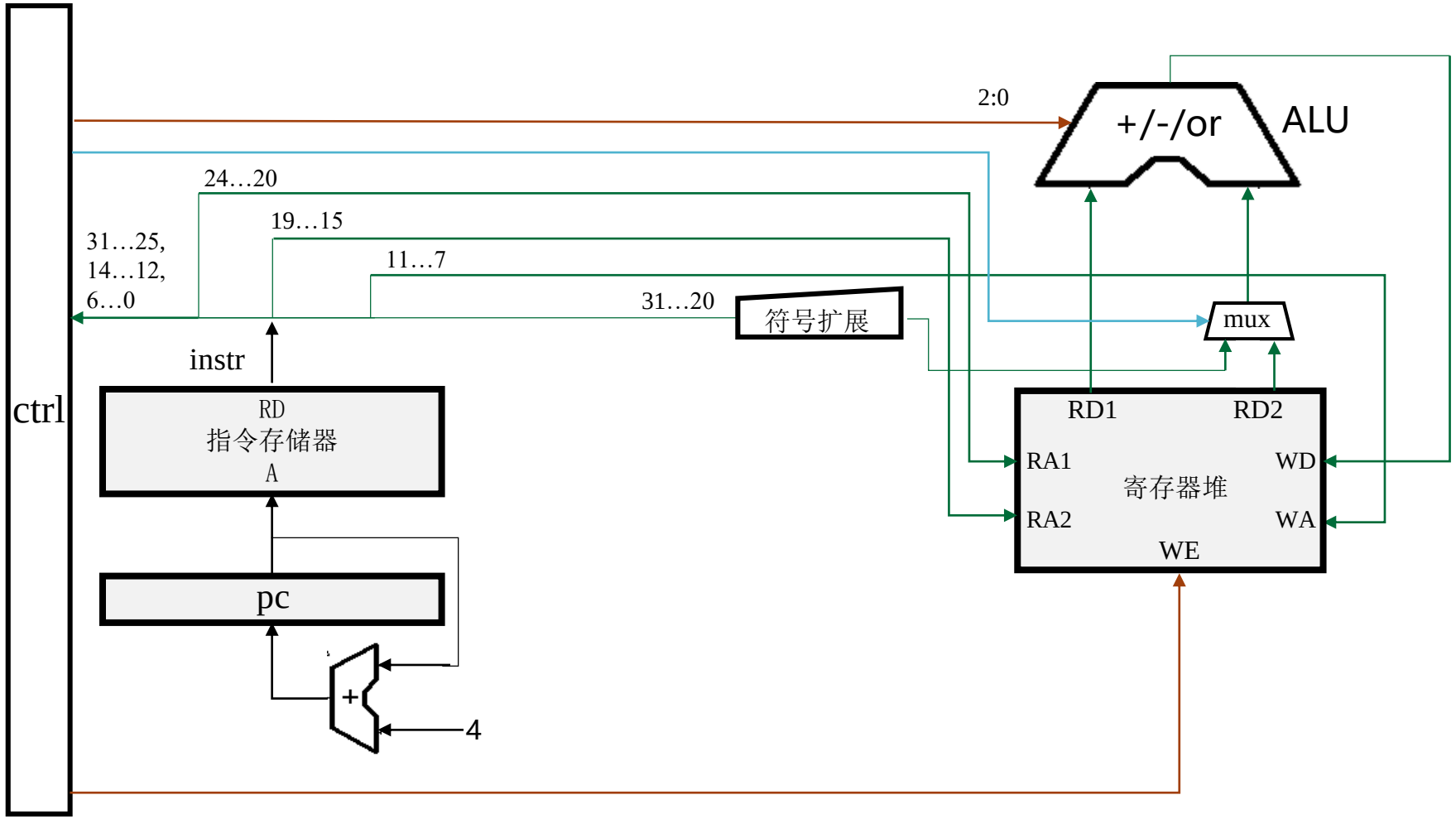
功能描述：将寄存器rs1的值与寄存器rs2的值相减，结果写入rd寄存器中。

```
sub t3, t0, t1
add t2, t0, t1
jal t4, 0x04
ori t4, zero, 0x50
beq t2, t3, 0x04
lw t3, 0(t4)
sw t2, 0(t4)
add t2, t1, t0
lui t1, 0x40
ori t0, zero, 0x30
sub a3, a0, a1
add a2, a0, a1
```

PC:start+4

新增ORI指令数据通路

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
imm[11:0]			rs1	110	rd	00100	11



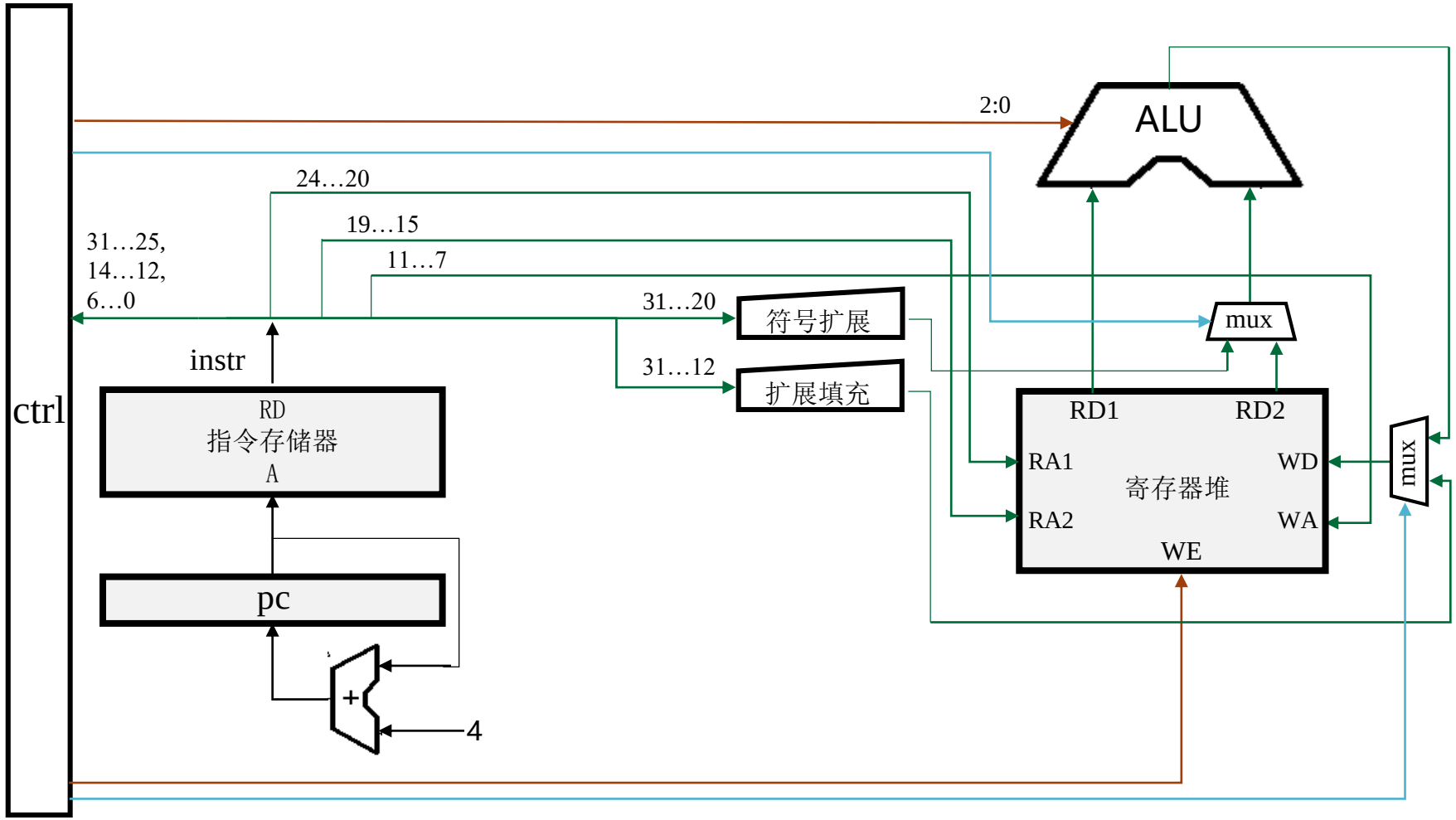
功能描述：寄存器rs1中的值与12-bit立即数imm的值按位逻辑或，结果写入寄存器rd中。

sub t3, t0, t1
add t2, t0, t1
jal t4, 0x04
ori t4, zero, 0x50
beq t2, t3, 0x04
lw t3, 0(t4)
sw t2, 0(t4)
add t2, t1, t0
lui t1, 0x40
ori t0, zero, 0x30
sub a3, a0, a1
add a2, a0, a1

PC:start+8

新增LUI指令数据通路

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
imm[31:12]					rd	01101	11



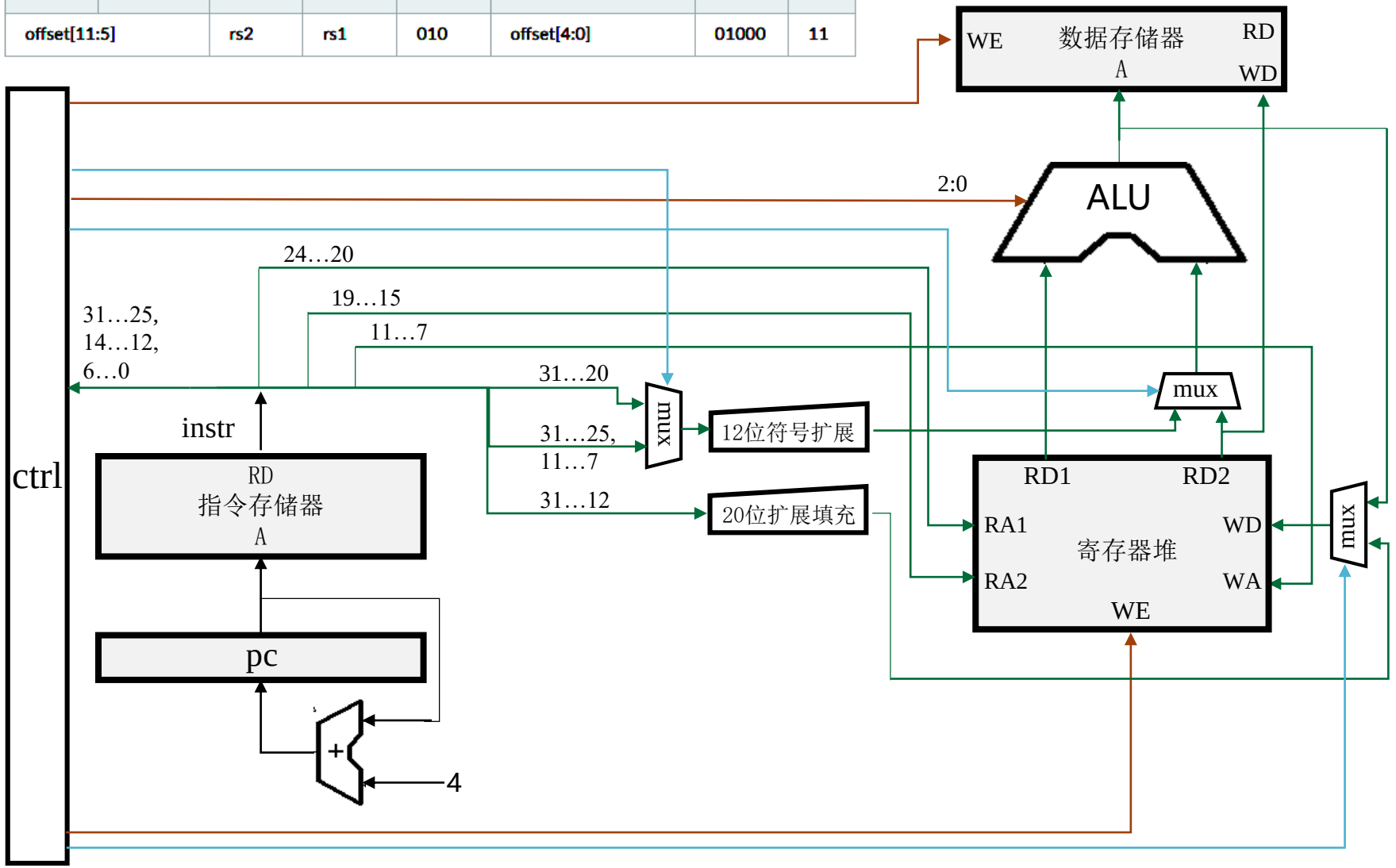
功能描述：20-bit立即数imm的值左移12位（低位用0填充），结果写入寄存器rd中。

sub t3, t0, t1
add t2, t0, t1
jal t4, 0x04
ori t4, zero, 0x50
beq t2, t3, 0x04
lw t3, 0(t4)
sw t2, 0(t4)
add t2, t1, t0
lui t1, 0x40
ori t0, zero, 0x30
sub a3, a0, a1
add a2, a0, a1

PC:start+12

新增SW指令数据通路

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[11:5]		rs2	rs1	010	offset[4:0]	01000	11



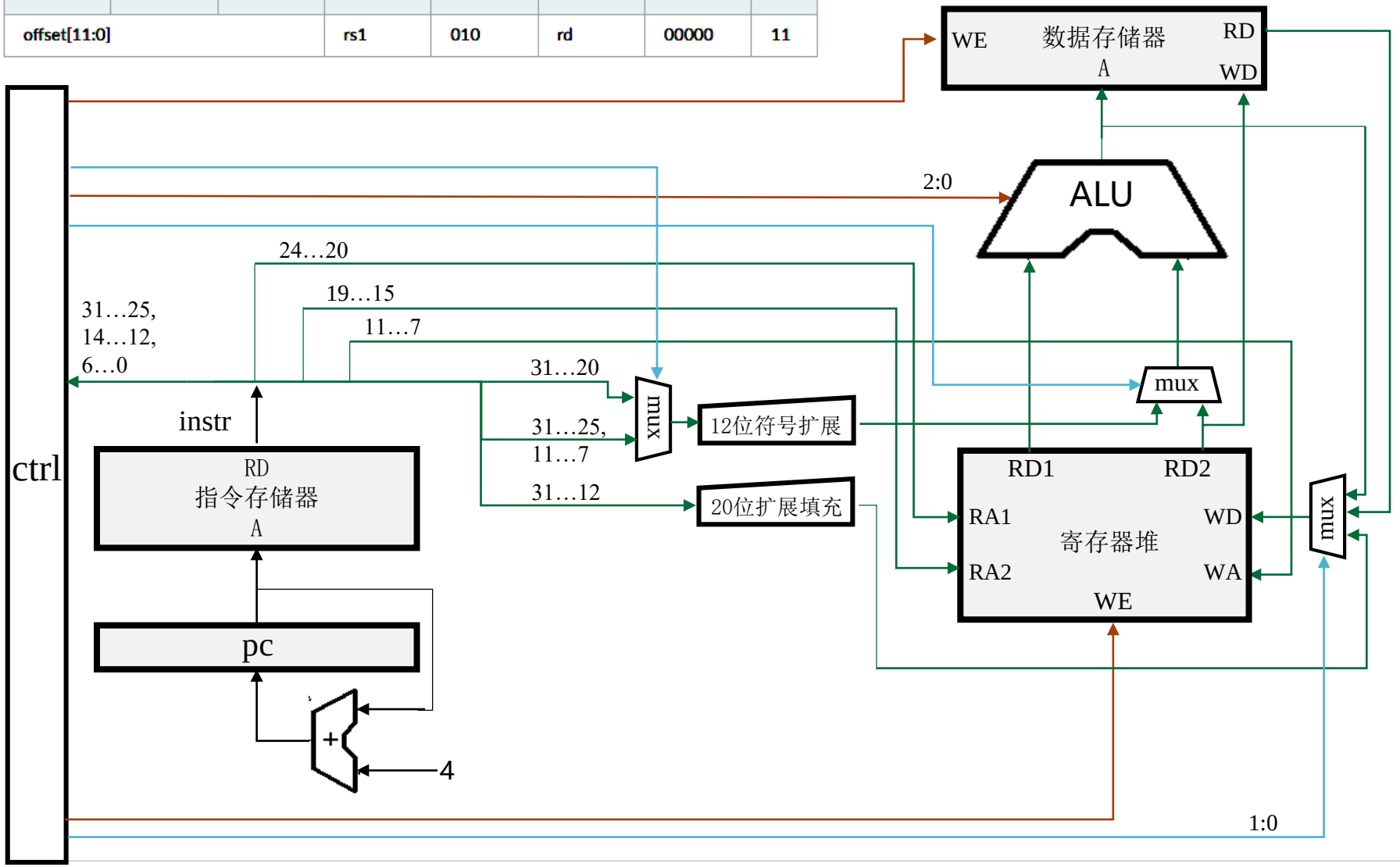
功能描述：将rs1寄存器的值加上符号扩展后的立即数offset得到访存的虚地址，据此虚地址将rs2寄存器存入存储器中。

sub t3, t0, t1
add t2, t0, t1
jal t4, 0x04
ori t4, zero, 0x50
beq t2, t3, 0x04
lw t3, 0(t4)
sw t2, 0(t4)
add t2, t1, t0
lui t1, 0x40
ori t0, zero, 0x30
sub a3, a0, a1
add a2, a0, a1

PC:start+20

新增LW指令数据通路

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[11:0]			rs1	010	rd	00000	11



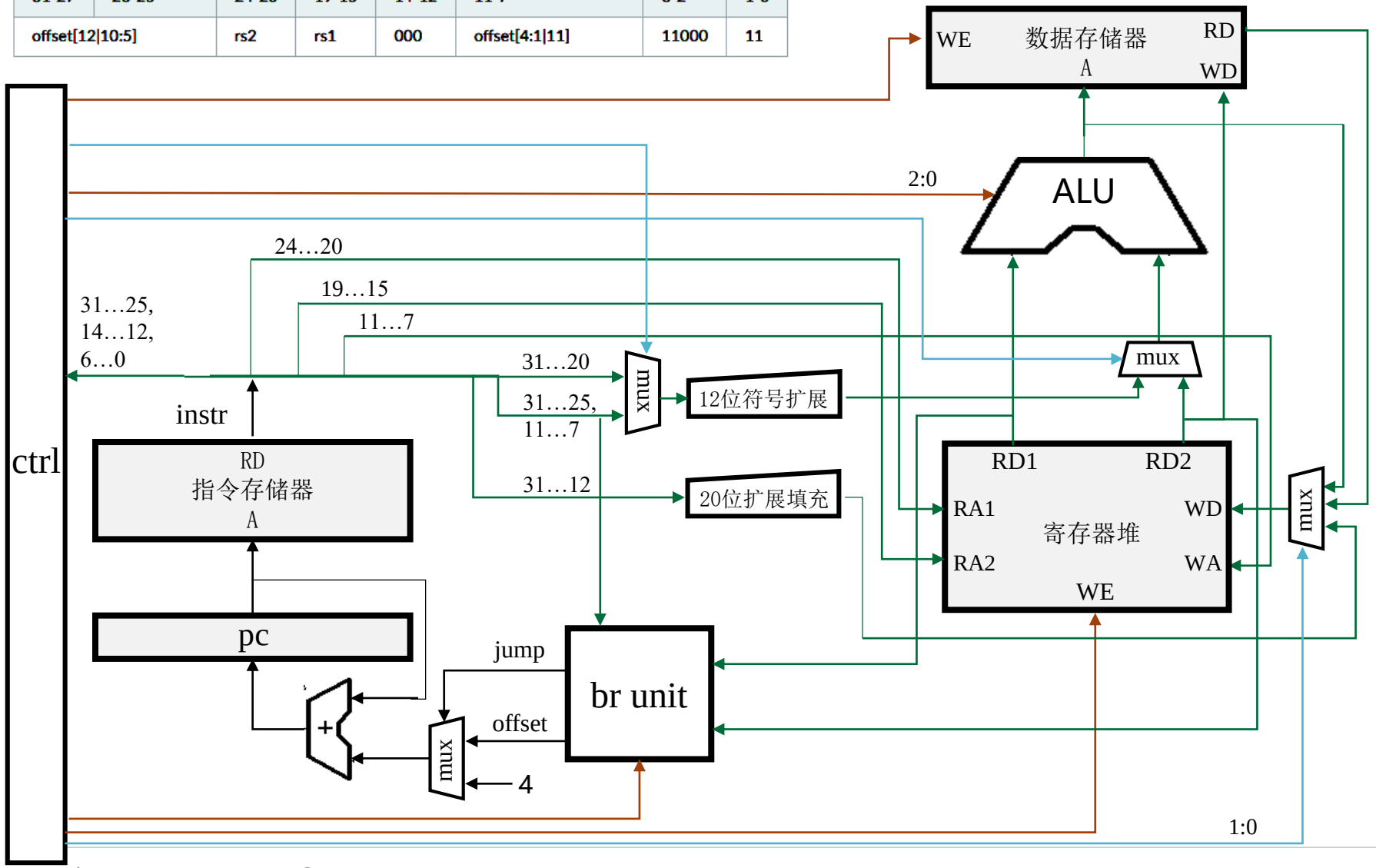
功能描述：将rs1寄存器的值加上符号扩展后的立即数offset得到访存的虚地址，据此虚地址将存储器中一个字的值取到rs2寄存器中。

sub t3, t0, t1
add t2, t0, t1
jal t4, 0x04
ori t4, zero, 0x50
beq t2, t3, 0x04
lw t3, 0(t4)
sw t2, 0(t4)
add t2, t1, t0
lui t1, 0x40
ori t0, zero, 0x30
sub a3, a0, a1
add a2, a0, a1

PC:start+24

新增BEQ指令数据通路

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[12|10:5]		rs2	rs1	000	offset[4:1|11]	11000	11



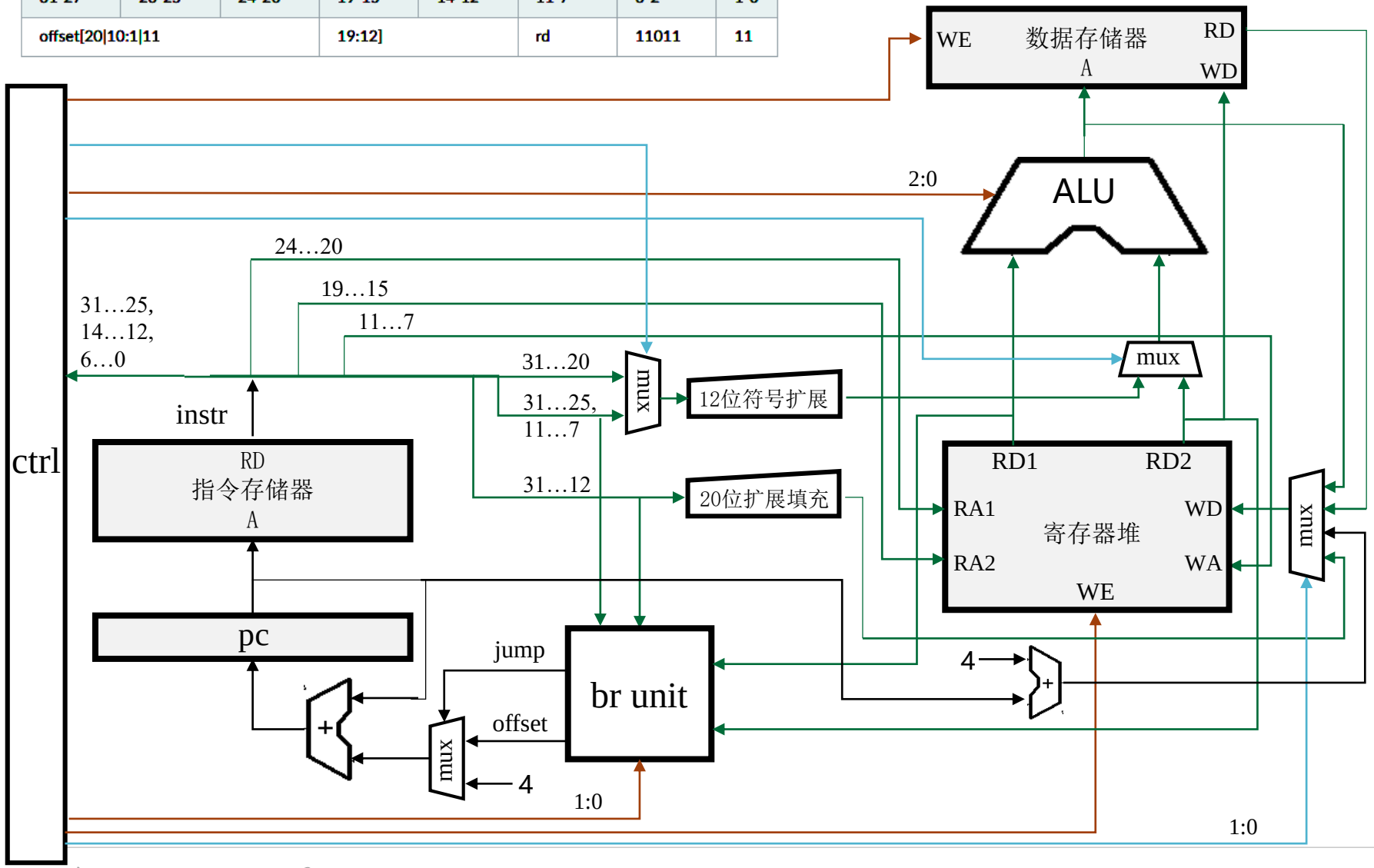
功能描述：如果寄存器rs1的值等于寄存器rs2的值则转移，否则顺序执行。转移目标由offset取低12位至低1位（即最低位置0之后的低12位）并进行有符号扩展的值加上当前分支指令的PC计算得到。

sub t3, t0, t1
add t2, t0, t1
jal t4, 0x04
ori t4, zero, 0x50
beq t2, t3, 0x04
lw t3, 0(t4)
sw t2, 0(t4)
add t2, t1, t0
lui t1, 0x40
ori t0, zero, 0x30
sub a3, a0, a1
add a2, a0, a1

PC:start+28

新增JAL指令数据通路

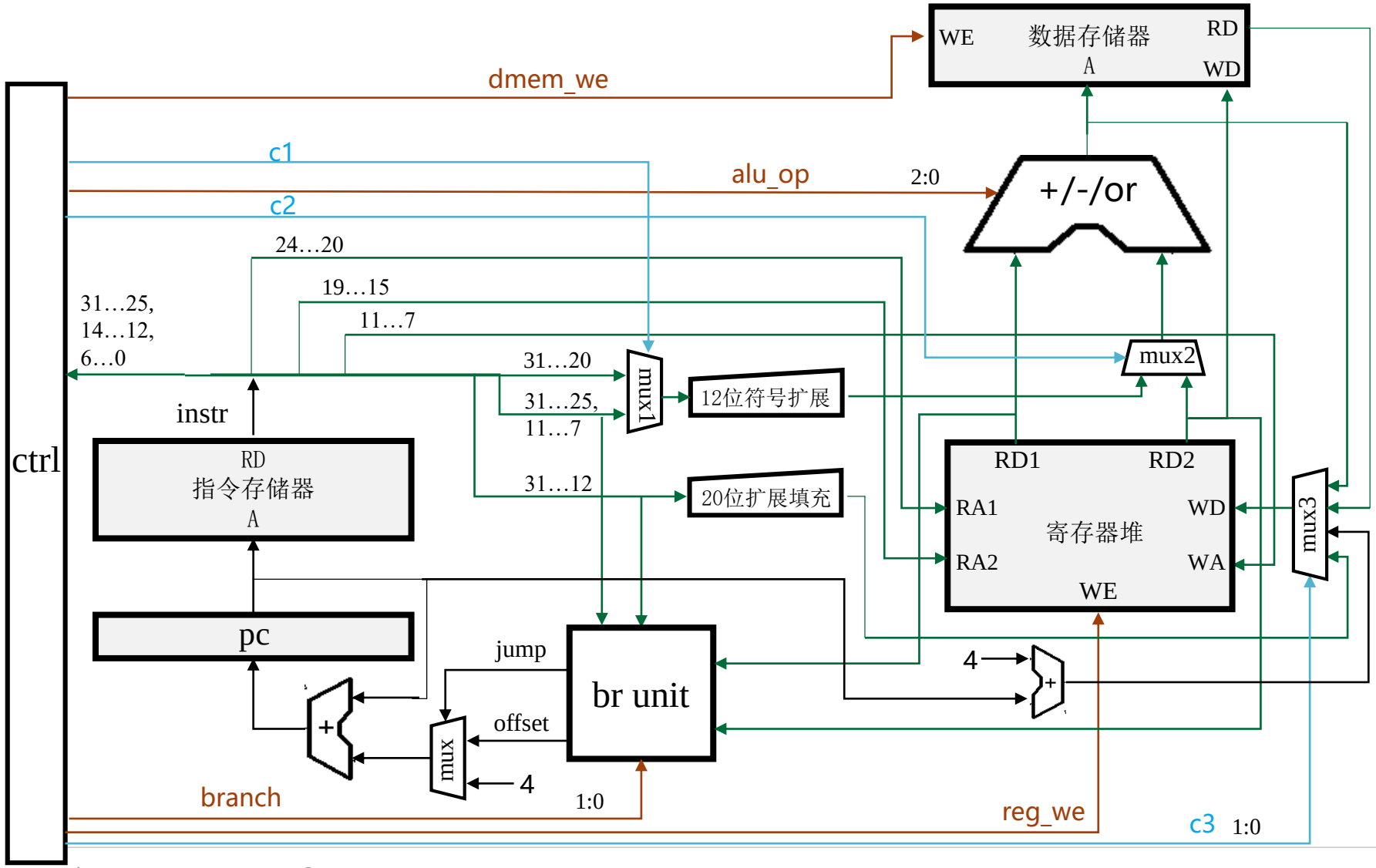
31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[20|10:1|11			19:12]		rd	11011	11



功能描述：无条件跳转，转移目标由offset指定（最低位总是置0）并进行有符号扩展的值加上当前指令的PC计算得到，同时将当前指令的下一条指令所对应的PC（即当前PC+4）写入目标寄存器rd。

sub t3, t0, t1
add t2, t0, t1
jal t4, 0x04 ← PC:start+36
ori t4, zero, 0x50
beq t2, t3, 0x04
lw t3, 0(t4)
sw t2, 0(t4)
add t2, t1, t0
lui t1, 0x40
ori t0, zero, 0x30
sub a3, a0, a1
add a2, a0, a1

最终设计结构



控制信号	位宽	控制对象
dmem_we	1	dmem
reg_we	1	regfile
alu_op	3	alu
branch	2	br unit
c1	1	mux1
c2	1	mux2
c3	2	mux3

控制模块设计

指令	类型	opcode	funct3	funct7
ADD	R	0110011	000	0000000
SUB	R	0110011	000	0100000
ORI	I	0010011	110	\
LUI	U	0110111	\	\
SW	S	0100011	010	\
LW	I	0000011	010	\
BEQ	B	1100011	000	\
JAL	J	1101111	\	\

· 模块输入：

opcode字段7位（instr[31:26]）
funct3字段3位（instr[14:12]）
funct7字段7位（instr[6:0]）

· 模块输出：

c1——I-type imm（instr[31:20]）和S-type imm（instr[31:25, 11:7]）之间2选1作为12位立即数
c2——12位立即数和rs2的值之间2选1作为ALU的第二个操作数
c3——20位立即数、当前PC+4、ALU运算结果、数据存储器读出数据4选1作为写回数据
alu_op——ALU运算使能（加/减/逻辑或，同一时间点最多只能某一个运算使能置位）
branch——分支处理单元跳转使能（条件分支/无条件跳转，同一时间点最多只能某一个跳转使能置位）
dmem_we——数据存储器写使能
reg_we——寄存器堆写使能

注：2选1　0-前者/1-后者
4选1　按顺序分别为00, 01, 10, 11
使能位　001-加/010-减/100-逻辑或，
01-条件分支/10-无条件跳转

指令	C1	C2	C3	alu_op	branch	dmem_we	reg_we
ADD	0	1	10	001	00	0	1
SUB	0	1	10	010	00	0	1
ORI	0	0	10	100	00	0	1
LUI	0	0	00	000	00	0	1
SW	1	0	00	001	00	1	0
LW	0	0	11	001	00	0	1
BEQ	0	0	00	000	01	0	0
JAL	0	0	01	000	10	0	1

· 根据输入输出列出真值表画、卡诺图，设计布尔表达式

控制模块代码 写法1

E: > Tasks > 小学期助教 > control.v

```
1  `timescale 1ns / 1ps
2
3  module control(
4      input [5:0] opcode,
5      input [5:0] func,
6
7      output c1,
8      output c2,
9      output c3,
10     output c4,
11     output [5:0] cA,
12     output [1:0] cB,
13     output dmem_we,
14     output reg_we
15 )
16
17 wire [3:0] inst_type; // 0-addi,1-sub,2-and,3-or,4-slt,5-sll,6-sw,7-lw,8-beq,9-j,10-reserved
18
19 assign inst_type = (opcode==6'b001000)? 0:
20                  (opcode==6'b101011)? 6:
21                  (opcode==6'b100011)? 7:
22                  (opcode==6'b000100)? 8:
23                  (opcode==6'b000010)? 9:
24                  (opcode==6'b000000)? (
25                      (func==6'b100010)? 1:
26                      (func==6'b100100)? 2:
27                      (func==6'b100101)? 3:
28                      (func==6'b101010)? 4:
29                      (func==6'b000000)? 5: 10): 10;
30
31 assign c1 = (inst_type==0||inst_type==6||inst_type==7)? 1: 0;
32 assign c2 = (inst_type==1||inst_type==2||inst_type==3||inst_type==4||inst_type==5)? 1: 0;
33 assign c3 = (inst_type==5)? 1: 0;
34 assign c4 = (inst_type==7)? 1: 0;
35 assign cA = (inst_type==0||inst_type==6||inst_type==7)? 6'b000001:
36           (inst_type==1)? 6'b000010:
37           (inst_type==2)? 6'b000100:
38           (inst_type==3)? 6'b001000:
39           (inst_type==4)? 6'b010000:
40           (inst_type==5)? 6'b100000: 6'b000000;
41 assign cB = (inst_type==8)? 2'b01:
42           (inst_type==9)? 2'b10: 2'b00;
43 assign dmem_we = (inst_type==6)? 1: 0;
44 assign reg_we = (inst_type==6||inst_type==8||inst_type==9||inst_type==10)? 0: 1;
45
46 endmodule
47
```

输入、输出声明

因为指令少，或者你还可以声明对应长度的reg变量直接将各指令需要的控制信号存入寄存器，执行那条指令就读出对应的寄存器

根据op、func判断指令类型

根据真值表给出各控制信号逻辑（也可以事先设计好布尔表达式再用与或非门组合的方式）

控制模块代码 写法2

```
22  `include "macro.vh"
23
24  module control(
25  //    input           clk,
26  //    input           rst,
27
28      input   [6:0]   opcode,
29      input   [2:0]   funct3,
30      input   [6:0]   funct7,
31
32      output          c1,         //
33      output          c2,         //
34      output  [1:0]   c3,         //
35      output  [2:0]   alu_op,     //
36      output  [1:0]   branch,
37      output          dmem_we,
38      output          reg_we
39  );
40
41      wire [3:0] inst_id = get_inst_id(opcode, funct3, funct7);
42      function [3:0] get_inst_id(input [6:0] opcode,
43                                 input [2:0] funct3, input [6:0] funct7);
44      begin
45          case (opcode)
46              `OPCODE_R   : begin
47                  case (funct3)
48                      `FUNCT3_ADDSUB  : begin
49                          case (funct7)
50                              `FUNCT7_ADD : get_inst_id = `ID_ADD;
51                              `FUNCT7_SUB : get_inst_id = `ID_SUB;
52                              default     : get_inst_id = `ID_NULL;
53                          endcase
54                      end
55                      default     : get_inst_id = `ID_NULL;
56                  endcase
57              end
58              `OPCODE_I   : begin
59                  case (funct3)
60                      `FUNCT3_ORI : get_inst_id = `ID_ORI;
61                      default     : get_inst_id = `ID_NULL;
62                  endcase
63              `OPCODE_L   : begin
64                  case (funct3)
65                      `FUNCT3_LW  : get_inst_id = `ID_LW;
66                      default     : get_inst_id = `ID_NULL;
67                  endcase
68              end
69              `OPCODE_S   : begin
70                  case (funct3)
71                      `FUNCT3_SW  : get_inst_id = `ID_SW;
72                      default     : get_inst_id = `ID_NULL;
73                  endcase
74              end
75              `OPCODE_B   : begin
76                  case (funct3)
77                      `FUNCT3_BEQ : get_inst_id = `ID_BEQ;
78                      default     : get_inst_id = `ID_NULL;
79                  endcase
80              end
81              `OPCODE_JAL : get_inst_id = `ID_JAL;
82              `OPCODE_LUI : get_inst_id = `ID_LUI;
83              default     : get_inst_id = `ID_NULL;
84          endcase
85      end
86      endfunction
87
88      assign c1 = (inst_id == `ID_SW ) ? 1 : 0;
89
90      reg [8:0] mask_c2 = 9'b000000110;
91      assign c2 = mask_c2[inst_id];
92
93      assign c3 = get_c3(inst_id);
94      function [1:0] get_c3(input [3:0] inst_id);
95      begin
96          case (inst_id)
97              `ID_LUI, `ID_SW, `ID_BEQ  : get_c3 = 2'b00;
98              `ID_JAL                   : get_c3 = 2'b01;
99              `ID_ADD, `ID_SUB, `ID_ORI : get_c3 = 2'b10;
100             `ID_LW                    : get_c3 = 2'b11;
101             default                   : get_c3 = 2'b00;
102         endcase
103     end
104     endfunction
105
106     assign alu_op = get_alu_op(inst_id);
107     function [3:0] get_alu_op(input [3:0] inst_id);
108     begin
109         case (inst_id)
110             `ID_SW, `ID_LW, `ID_ADD : get_alu_op = `ALU_ADD;
111             `ID_SUB                 : get_alu_op = `ALU_SUB;
112             `ID_ORI                 : get_alu_op = `ALU_OR;
113             default                 : get_alu_op = `ALU_NULL;
114         endcase
115     end
116     endfunction
117
118     assign branch = get_branch(inst_id);
119     function [1:0] get_branch(input [3:0] inst_id);
120     begin
121         case (inst_id)
122             `ID_BEQ                 : get_branch = `BR_BEQ;
123             `ID_JAL                 : get_branch = `BR_JAL;
124             default                 : get_branch = `BR_NULL;
125         endcase
126     end
127     endfunction
128
129     assign dmem_we = (inst_id == `ID_SW ) ? 1 : 0;
130
131     reg [8:0] mask_reg_we = 9'b101011110;
132     assign reg_we = mask_reg_we[inst_id];
133
134 endmodule
135
```

控制模块代码 写法2 macro.vh

```
// opcode
`define OPCODE_R    7'b0110011
`define OPCODE_I    7'b0010011
`define OPCODE_L    7'b0000011
`define OPCODE_S    7'b0100011
`define OPCODE_B    7'b1100011
`define OPCODE_JAL  7'b1101111
`define OPCODE_LUI  7'b0110111

// funct3
`define FUNCT3_ADDSUB   3'b000
`define FUNCT3_ORI      3'b110
`define FUNCT3_LW       3'b010
`define FUNCT3_SW       3'b010
`define FUNCT3_BGE      3'b101

// funct7
`define FUNCT7_ADD  7'b0000000
`define FUNCT7_SUB  7'b0100000
```

```
// inst_id
`define ID_NULL     0
`define ID_ADD      1
`define ID_SUB      2
`define ID_ORI      3
`define ID_LUI      4
`define ID_SW       5
`define ID_LW       6
`define ID_BGE      7
`define ID_JAL      8

// alu_op
`define ALU_NULL    3'b000
`define ALU_ADD     3'b001
`define ALU_SUB     3'b010
`define ALU_OR      3'b100

// branch
`define BR_NULL     2'b00
`define BR_BGE      2'b01
`define BR_JAL      2'b10
```

指令/数据存储器

```
1   `timescale 1ns / 1ps
2
3   module iMem(//只读
4       input clk,
5       input rst,
6
7       input [31:0] _addr,//要读出的指令地址
8       output [31:0] idata//读出的指令数据
9       );
10
11      reg [7:0] imem[255:0];
12      //存储程序
13      initial begin
14      $readmemb("F:/sw_wrongaddr.txt",imem);
15      end
16
17      wire [7:0] addr=_addr[7:0];
18      assign idata={imem[addr+3],imem[addr+2],imem[addr+1],imem[addr]};
19
20  endmodule
21
```

结合Rars生成指令机器码

申请reg作为指令存储器。Verilog系统函数初始化空间内容（仅适用于仿真）。

读出指令

```
1   `timescale 1ns / 1ps
2
3   module dMem(
4       input clk,
5       input rst,
6       input we, //we=0只读，we=1可读可写
7
8       input [31:0] _addr,
9       input [31:0] wdata, //写入的数据
10
11      output [31:0] rdata
12      );
13
14
15      reg [7:0] dmem [255:0]; //载入初始化数据
16      initial begin
17      $readmemb("F:/add_overflow_data.txt",dmem);
18      end
19
20      wire [7:0] addr=_addr[7:0];
21      assign rdata={dmem[addr+3],dmem[addr+2],dmem[addr+1],dmem[addr]};
22
23      always @(posedge clk) begin
24          if(we) begin
25              dmem[addr]<=wdata[7:0];
26              dmem[addr+1]<=wdata[15:8];
27              dmem[addr+2]<=wdata[23:16];
28              dmem[addr+3]<=wdata[31:24];
29          end
30      end
31
32  endmodule
33
```

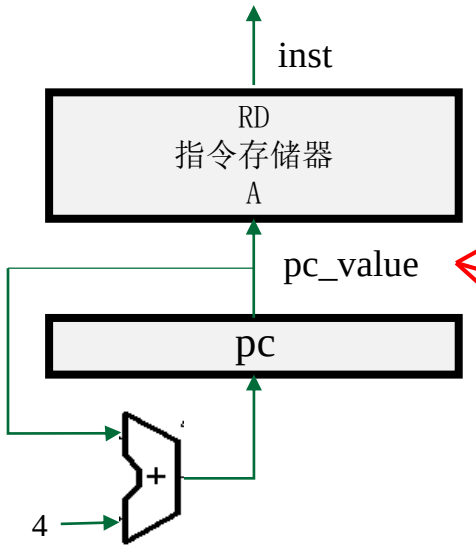
定义数据存储器空间并初始化。

读数据存储器

写数据存储器

CPU TOP模块

各个模块需要封装到一个顶层模块中组合成CPU，以下面CPU的局部情况为例：



经过中间数据通路

输出

输入

```
1  `timescale 1ns / 1ps
2
3  module top_cpu(
4      input clk,
5      input rst,
6      );
7
8      wire jmp;
9      wire [31:0] offset;
10     wire branch;
11     wire [31:0] target;
12     wire [31:0] pc_value;
13     wire [31:0] inst;
14
15     pc pc( // PC模块内置PC寄存器且内部处理mux5和mux6多路复用的逻辑
16         .clk(clk),
17         .rst(rst),
18
19         .jmp(jmp), //1-jmp to target
20         .branch(branch), // 1-branch to PC+offset
21         .offset(offset),
22         .target(target),
23
24         .pc_value(pc_value)
25         );
26
27     iMem iMem( //只读
28         .clk(clk),
29         .rst(rst),
30
31         ._addr(pc_value), //要读出的指令地址
32         .idata(inst) //读出的指令数据
33     );
34
35     ......
```

定义好中间线作为数据传输的通路

分支处理模块传来的数据通路（左图未画出）

模块元件例化

其它模块的例化及模块间数据通路请自行补充

testbench

仿真时缺少板上时钟和复位信号需要自行编写testbench给出相应的激励。

此时需要把top_cpu封装到testbench模块之下。

我们目前的CPU比较简单，只有clk和rst信号需要输入。

运行仿真查看仿真波形无误后可上板运行。

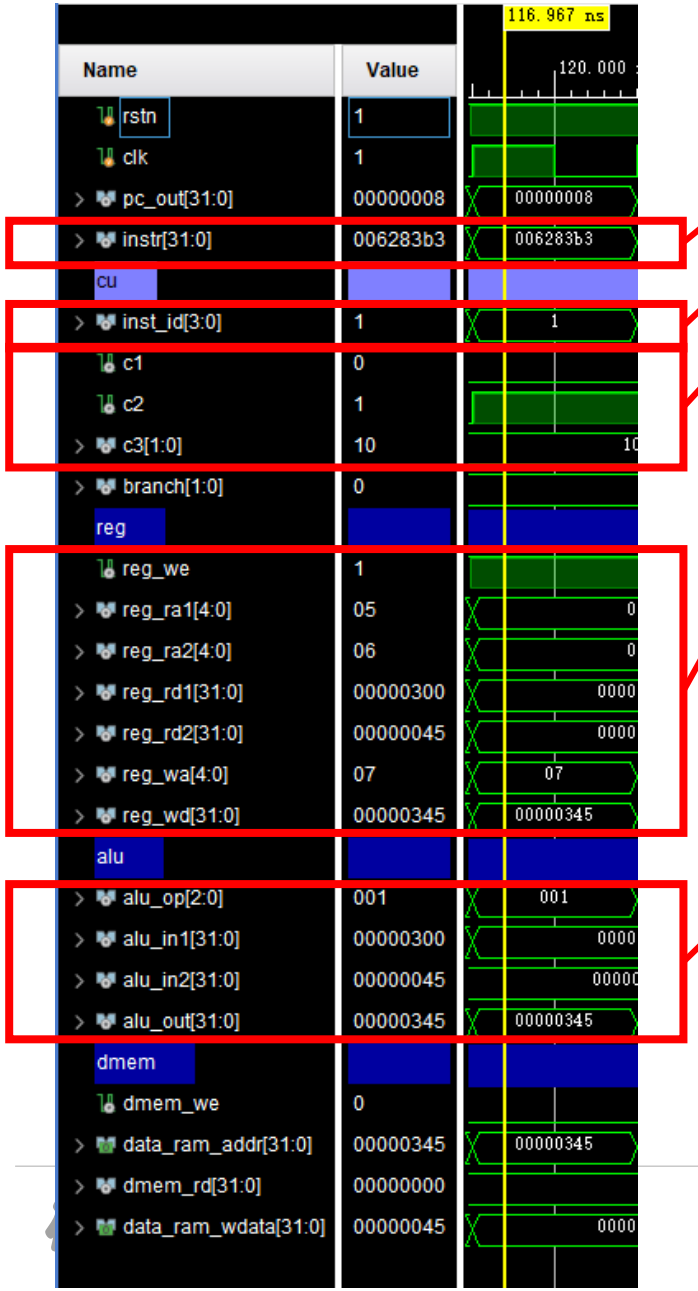
```
1   `timescale 1ns / 1ps
2
3   module testbench(
4       );
5
6       reg clk;
7       reg rst;
8       initial begin
9           clk=0;
10          rst=0;
11          #10
12          rst=1;
13      $display("running..."); // 系统打印函数
14      end
15      always #10 clk=~clk;
16
17      top_cpu top_cpu(
18      .clk(clk),
19      .rst(rst),
20      );
21
22  endmodule
23
```

指令波形

检查顺序：

- PC & instr
- reg： reg_ra1, reg_ra2, reg_rd1, reg_rd2, reg_we, reg_wa, reg_wd
- cu: inst_id, c1, c2, c3
- alu: alu_op, alu_in1, alu_in2, alu_out
- dmem: dmem_a, dmem_rd, dmem_we, dmem_wd
- br: branch, pc_in, offset, jump

ADD指令波形



当前指令：add t2, t0, t1

对应机器码：0x006283b3

对应指令序号：`ID_ADD = 1

对应控制信号：c1 = 0（非立即数指令，无效）

c2 = 1（rs2作为ALU第二操作数）

c3 = 2’ b10（ALU运算结果作为写回数据）

寄存器操作：reg_ra1 = 0x05（ALU第一操作数为t0）

reg_ra2 = 0x06（ALU第一操作数为t1）

reg_rd1、reg_rd2 为对应寄存器的值

reg_wa = 0x07（目标寄存器t2）

reg_we = 1（该指令将写入寄存器）

reg_wd = 0x00000345（同alu_out）

ALU操作：alu_op = 3’ b001（加法操作）

alu_in1 = 0x00000300（同reg_rd1）

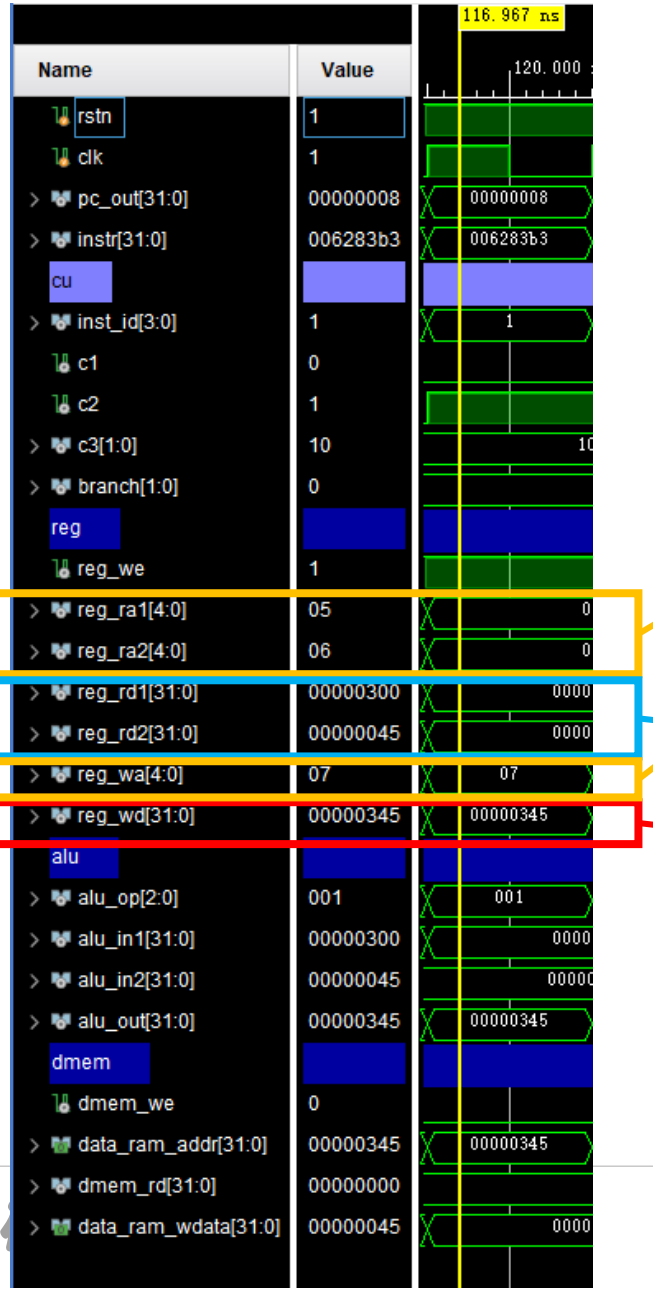
alu_in2 = 0x00000045（同reg_rd2）

alu_out = 0x00000345（运算结果）

dmem操作：无

分支跳转操作：无

ADD指令波形



	Code	Basic	Source
0x00400000	0x30006293	ori x5, x0, 0x00000300	5: ori t0, zero, 0x300
0x00400004	0x04506313	ori x6, x0, 0x00000045	6: ori t1, zero, 0x45
0x00400008	0x006283b3	add x7, x5, x6	7: add t2, t0, t1
0x0040000c	0x40628e33	sub x28, x5, x6	8: sub t3, t0, t1
0x00400010	0x00012eb7	lui x29, 18	9: lui t4, 0x12
0x00400014	0x007e8eb3	add x29, x29, x7	10: add t4, t4, t2
0x00400018	0x008000ef	jal x1, 0x00000008	11: jal label1
0x0040001c	0x00106193	ori x31, x0, 1	12: ori t6, zero, 0x1
0x00400020	0x00040537	lui x10, 0x00000040	14: lui a0, 0x40
0x00400024	0x01d52023	sw x29, 0 (x10)	15: sw t4, (a0)

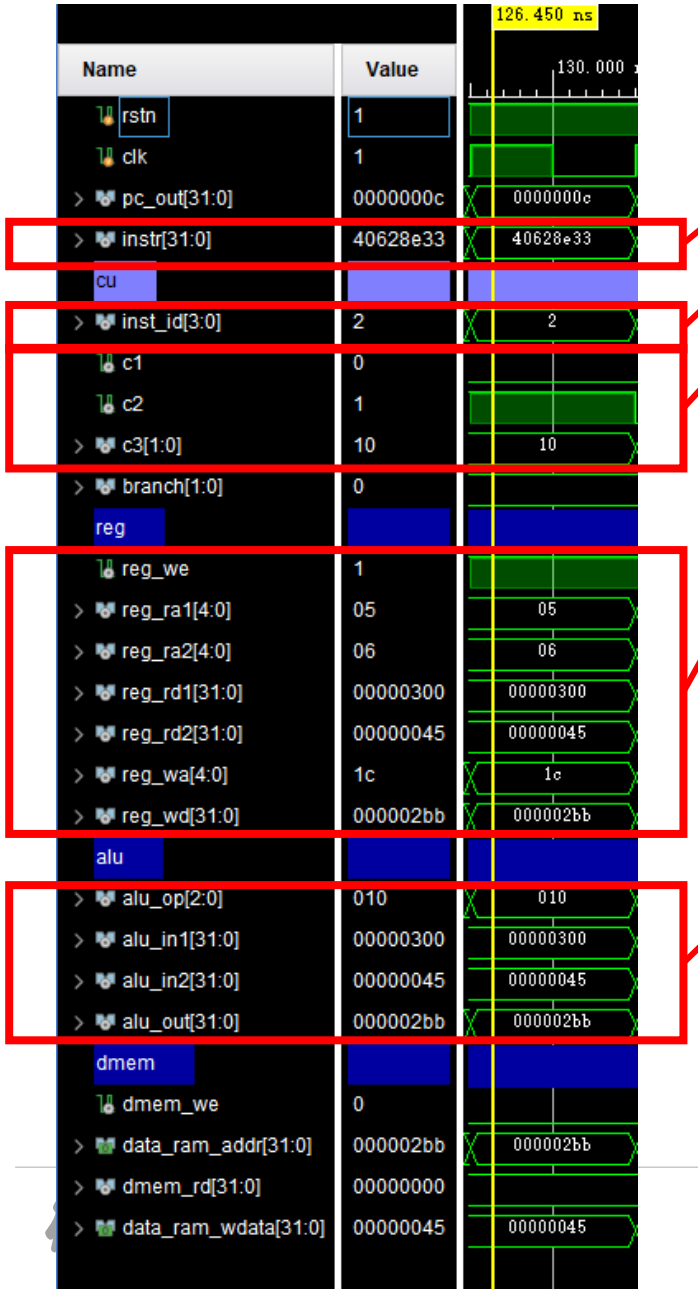
Registers | Floating Point | Control and Status

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x7fffeffc
gp	3	0x10008000
tp	4	0x00000000
t0	5	0x00000300
t1	6	0x00000045
t2	7	0x00000000
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x00000000
a1	11	0x00000000
a2	12	0x00000000
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000

Registers | Floating Point | Control and Status

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x7fffeffc
gp	3	0x10008000
tp	4	0x00000000
t0	5	0x00000300
t1	6	0x00000045
t2	7	0x00000345
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x00000000
a1	11	0x00000000
a2	12	0x00000000

SUB指令波形



当前指令：sub t3, t0, t1

对应机器码：0x40628e33

对应指令序号：`ID_SUB = 2

对应控制信号：c1 = 0（非立即数指令，无效）

c2 = 1（rs2作为ALU第二操作数）

c3 = 2’ b10（ALU运算结果作为写回数据）

寄存器操作：reg_ra1 = 0x05（ALU第一操作数为t0）

reg_ra2 = 0x06（ALU第一操作数为t1）

reg_rd1、reg_rd2 为对应寄存器的值

reg_wa = 0x1c（目标寄存器t3）

reg_we = 1（该指令将写入寄存器）

reg_wd = 0x000002bb（同alu_out）

ALU操作：alu_op = 3’ b010（减法操作）

alu_in1 = 0x00000300（同reg_rd1）

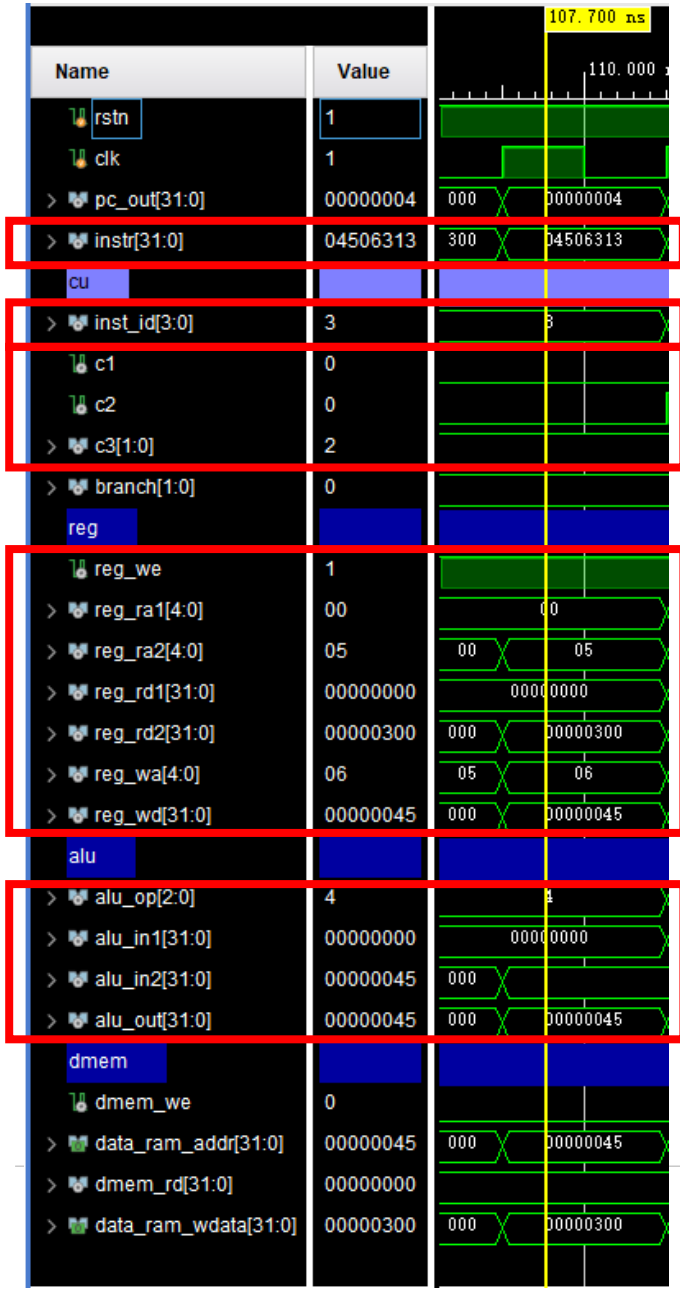
alu_in2 = 0x00000045（同reg_rd2）

alu_out = 0x000002bb（运算结果）

dmem操作：无

分支跳转操作：无

ORI指令波形



当前指令：ori t1, zero, 0x45

对应机器码：0x04506313

对应指令序号：`ID_ORI = 3

对应控制信号：c1 = 0（I-type 12位立即数）

c2 = 0（12位立即数作为ALU第二操作数）

c3 = 2’ b10（ALU运算结果作为写回数据）

寄存器操作：reg_ra1 = 0x00（ALU第一操作数为0号寄存器，即zero）

reg_ra2 = 0x05（非双操作数寄存器指令，无效）

reg_rd1、reg_rd2 为对应寄存器的值

reg_wa = 0x06（目标寄存器t1）

reg_we = 1（该指令将写入寄存器）

reg_wd = 0x00000045（同alu_out）

ALU操作：alu_op = 3’ b100（逻辑或操作）

alu_in1 = 0x00000000（同reg_rd1，0号寄存器，值总为0）

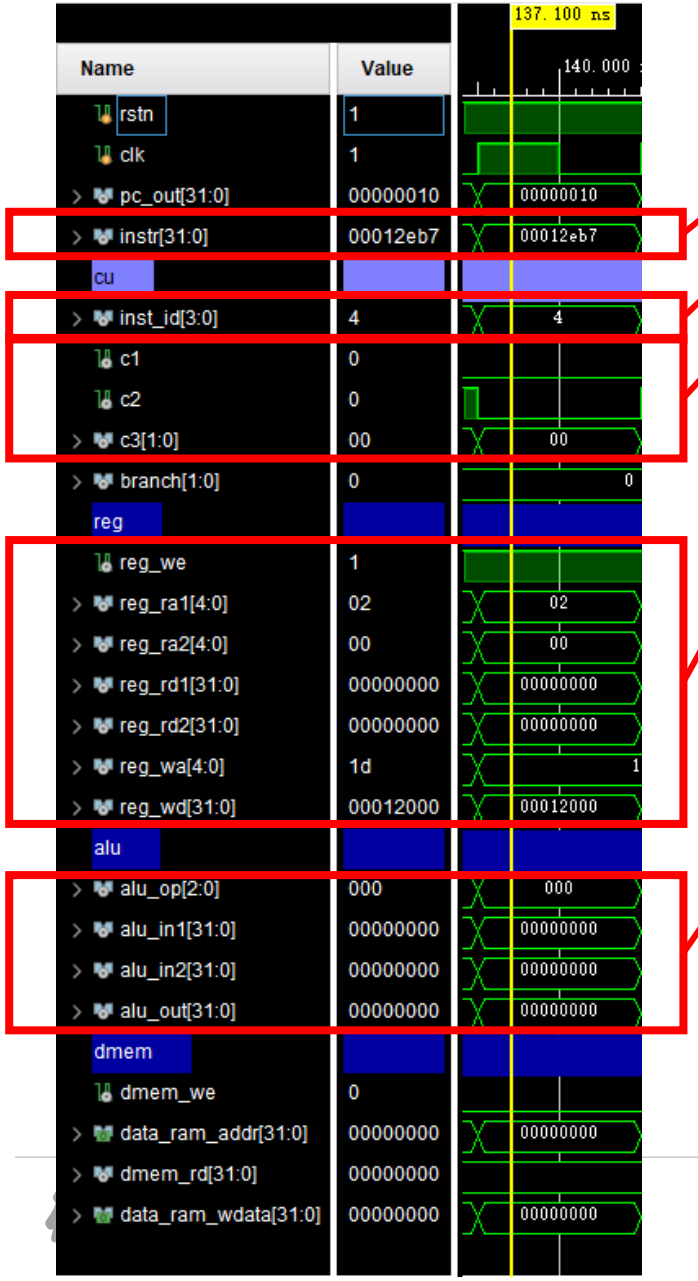
alu_in2 = 0x00000045（立即数）

alu_out = 0x00000045（运算结果）

dmem操作：无

分支跳转操作：无

LUI指令波形



当前指令：lui t4, 0x12

对应机器码：0x00012eb7

对应指令序号：`ID_LUI = 4

对应控制信号：c1 = 0（非12位立即数指令，无效）

c2 = 0（非双操作数指令，无效）

c3 = 2’ b00（20位立即数作为写回数据）

寄存器操作：reg_ra1、reg_ra2 无效

reg_rd1、reg_rd2 无效

reg_wa = 0x1d（目标寄存器t4）

reg_we = 1（该指令将写入寄存器）

reg_wd = 0x00012000（20位立即数移位扩展后的值）

ALU操作：alu_op = 3’ b000（无操作）

alu_in1、alu_in2、alu_out 无效

dmem操作：无

分支跳转操作：无

SW指令波形

当前指令：sw t4, (a1)

对应机器码：0x01d52023

对应指令序号：`ID_SW = 5

对应控制信号：c1 = 1（S-type 12位立即数）

c2 = 0（12位立即数作为ALU第二操作数）

c3 = 2’ b00（非写回寄存器指令，无效）

寄存器操作：reg_ra1 = 0x0b（ALU第一操作数为a1）

reg_rd1 为对应寄存器的值

reg_wa、reg_wd 非写回寄存器指令，无效

reg_we = 0（该指令不写入寄存器）

ALU操作：alu_op = 3’ b001（加法操作，计算偏移后的内存地址）

alu_in1 = 0x00040004（同reg_rd1）

alu_in2 = 0x00000000（即12位立即数扩展）

alu_out = 0x00040004（运算结果）

dmem操作：dmem_a = 0x00040004（同alu_out）

dmem_wd = 0x00012345（要写入内存对应地址的内容）

dmem_we = 1（写入内存）

分支跳转操作：无

LW指令波形

当前指令：lw t5, (a0)

对应机器码：0x40628e33

对应指令序号：`ID_LW = 6

对应控制信号：c1 = 0（I-type 12位立即数）

c2 = 0（12位立即数作为ALU第二操作数）

c3 = 2’ b11（dmem读出数据作为写回数据）

寄存器操作：reg_ra1 = 0x0a（ALU第一操作数为t5）

reg_rd1 为对应寄存器的值

reg_wa = 0x1e（目标寄存器t5）

reg_we = 1（该指令将写入寄存器）

reg_wd = 0x00012345（同dmem_rd）

ALU操作：alu_op = 3’ b001（加法操作，计算偏移后的内存地址）

alu_in1 = 0x00040000（同reg_rd1）

alu_in2 = 0x00000000（即12位立即数扩展）

alu_out = 0x00040000（运算结果）

dmem操作：dmem_a = 0x00040000（同alu_out）

dmem_rd = 0x00012345（内存对应地址的内容）

dmem_we = 0（非写入）

分支跳转操作：无

BEQ指令波形

当前指令：beq t4, t5, 0x08

对应机器码：0x01ee8463

对应指令序号：`ID_BEQ = 7

对应控制信号：c1 = 1（S-type 12位立即数）

c2 = 0（12位立即数作为ALU第二操作数）

c3 = 2’ b00（非写回寄存器指令，无效）

寄存器操作：reg_ra1 = 0x1d（条件跳转第一操作数为t4）

reg_ra2 = 0x1e（条件跳转第一操作数为t5）

reg_rd1、reg_rd2 为对应寄存器的值

reg_we = 0（该指令不写入寄存器）

ALU操作：无

dmem操作：无

分支跳转操作：branch = 2’ b01（条件跳转）

offset = 0x0000008（即12位立即数位扩展后的值）

pc_in = 0x0000040（跳转目标指令的PC）

jump = 1（需要执行跳转）

JAL指令波形

当前指令：jal ra, 0x08

对应机器码：0x008000ef

对应指令序号：`ID_JAL = 8

对应控制信号：c1 = 0（非12位立即数指令，无效）

c2 = 0（非12位立即数指令，无效）

c3 = 2’ b01（当前PC+4作为写回数据）

寄存器操作：reg_ra1、reg_ra2、reg_rd1、reg_rd2 不读取寄存器，无效

reg_wa = 0x01（目标寄存器为ra）

reg_wd = 0x000001c（写回数据为当前PC+4）

reg_we = 1（该指令将写入寄存器）

ALU操作：无

dmem操作：无

分支跳转操作：branch = 2’ b10（无条件跳转）

offset = 0x0000008（即20位立即数位扩展后的值）

pc_in = 0x0000020（跳转目标指令的PC）

jump = 1（需要执行跳转）

北京理工大学
BEIJING INSTITUTE OF TECHNOLOGY

感谢各位

主讲人：陈康冰
计算机学院