# 第五章 数据通路及其控制

作业参考答案

**5.2** [10] <§5.4> Describe the effect that a single stuck-at-0 fault (i.e., regardless of what it should be, the signal is always 0) would have for the signals shown below, in the single-cycle datapath in Figure 5.17 on page 307. Which instructions, if any, will not work correctly? Explain why.

Consider each of the following faults separately:

a. RegWrite = 0
b. ALUop0 = 0
c. ALUop1 = 0
d. Branch = 0
e. MemRead = 0
f. MemWrite = 0

参看图**5-18**可以很快得到结论！

答：若**RegWrite=0**，则所有的需要写结果到寄存器的指令（如：**R-Type**指令、**load**指令等）都不能正确执行，因为寄存器不发生写操作；

若**ALUop0=0**， 则除**add**外的**R-type**指令不能正确执行（参看图**5-12**中**ALUop**的定义）

若**ALUop1=0**， 则**Branch**指令不能正确执行（参看图**5-12**中**ALUop**的定义）

若**Branch=0**，则**Branch**指令可能出错，因为永远不会发生转移；

若**MemRead=0**，则**Load**指令不能正确执行，因为存储器不能读出所需数据；

若**MemWrite=0**，则**Store**指令不能正确执行，因为存储器不能写入所需数据；

此外，

若**Regdst=0**，则所有**R-Type**指令都不能正确执行，因为目的寄存器指定错误；

若**ALUSrc=0**，则所有**I-Type**指令（除**Branch**）都不能正确执行，因为第二个操作数不是立即数扩展；

若**MemtoReg=0**，则所有的**Load**指令执行错误，因为寄存器写入的是**ALU**输出 ……

**5.3** [5] <§5.4> This exercise is similar to Exercise 5.2, but this time consider stuck-at-1 faults (the signal is always 1).

答：若**RegWrite=1**，则所有不需要写结果到寄存器的指令（如：**Store**指令、**Branch**指令等）都不能正确执行，因为寄存器发生了不需要的写操作；

若**ALUop0=1**， 则**Load/Store**和**Branch**指令可能不正确（参看图**5-12**中**ALUop**的定义）

若**ALUop1=1**， 则**Load/Store**和**R-type**指令可能不正确（参看图**5-12**中**ALUop**的定义）

若**Branch=1**，则除**Branch**指令指令外的其他指令可能不正确

若**MemRead=1**，则除**Load**指令外的其他指令可能不正确

若**MemWrite=1**，则除**Store**指令外的其他指令可能不正确

此外，

若**Regdst=1**，则**Load**指令和**I-Type**指令都不能正确执行，因为目的寄存器指定错误；

若**ALUSrc=1**，则所有**R-Type**指令和**Branch**指令都不能正确执行，因为第二个操作数为立即数扩展，而不是寄存器数据；

若**MemtoReg=1**，则所有**R-Type**指令执行错误，因为寄存器写入的是存储器读出的数据

……

**5.8** [15] <§5.4> We wish to add the instruction `jr` (jump register) to the single-cycle datapath described in this chapter. Add any necessary datapaths and control signals to the single-cycle datapath of Figure 5.17 on page 307 and show the necessary additions to Figure 5.18 on page 308. You can photocopy these figures to make it faster to show the additions.

答： jr指令为R-Type格式：

| 0 | 31 | 0 | 0 | 0 | 8 |
|---|---|---|---|---|---|

其功能是读出31号寄存器送PC，指令中的Rt和Rd都是0，因为0号寄存器可以读出，但不能改变其值，所以，写入信号对0号寄存器不起作用，因而，可以把jr指令看成是结果送PC的加法指令。即： ($31)+($0)->$0，同时($31)+($0)->PC

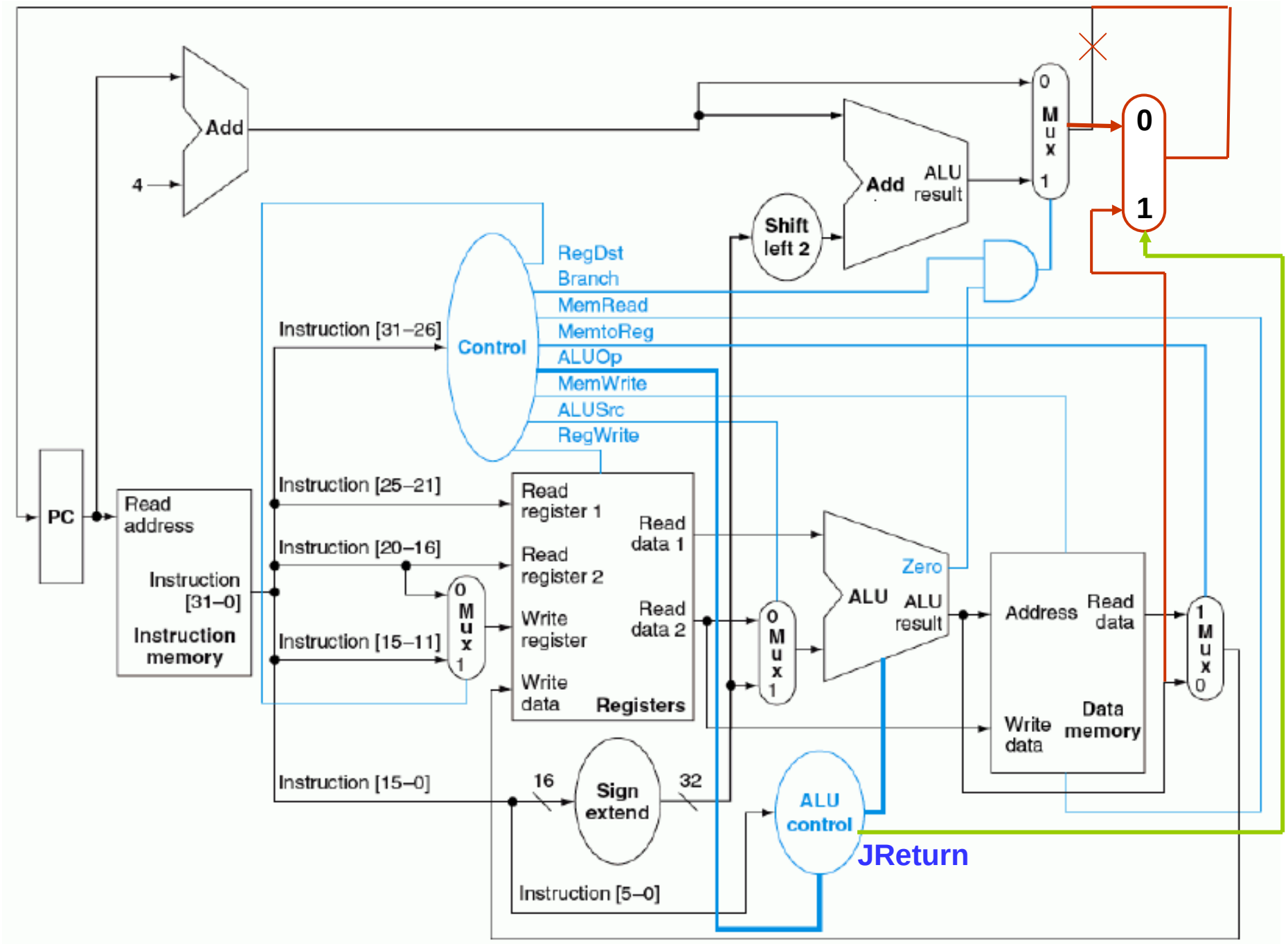
因此，原R-Type指令的数据通路不需要改动，而只要增加ALU结果送PC的数据通路，并对控制信号作相应修改即可。具体如下：

（1）增加一个PC的来源：ALU输出->PC，因此再加一个MUX

（2）新加一个控制信号Jreturn，用于对新加MUX的控制

（3）控制器的设计中要考虑R-Type指令的func=001000的情况，即在图5-12和5-13的表中加入相关的组合情况，以反映到控制器的设计中

（思考：ALU控制器的逻辑如何修改？主控制逻辑要修改吗？）


Add
4
M
u
x
0
1
Add
ALU
result
Shift
left 2
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
PC
Read
address
Instruction
[31–0]
Instruction
memory
Read
register 1
Read
register 2
Write
register
Write
data
Read
data 1
Read
data 2
Registers
ALU
Zero
ALU
result
Address
Read
data
Write
data
Data
memory
16
Sign
extend
32
ALU
control
JReturn

根据书中图**5-12**可以得到以下图**5-13**，在此基础上再加上**Jr**指令的控制信号

| ALUOp | | Funct field | | | | | | Operation | JReturn |
|---|---|---|---|---|---|---|---|---|---|
| ALUOp1 | ALUOp0 | F5 | F4 | F3 | F2 | F1 | F0 | | |
| 0 | 0 | X | X | X | X | X | X | 0010 | 0 |
| X | 1 | X | X | X | X | X | X | 0110 | |
| 1 | X | X | X | 0 | 0 | 0 | 0 | 0010 | |
| 1 | X | X | X | 0 | 0 | 1 | 0 | 0110 | |
| 1 | X | X | X | 0 | 1 | 0 | 0 | 0000 | |
| 1 | X | X | X | 0 | 1 | 0 | 1 | 0001 | |
| 1 | X | X | X | 1 | 0 | 1 | 0 | 0111 | |
| **1** | **X** | **0** | **0** | **1** | **0** | **0** | **0** | **0010** | 1 |

上述的做法只修改**ALU**控制逻辑而不修改主控制逻辑，所以无需对图**5-18**进行修改。

也可以采用修改主控制逻辑的方式，此时，主控制逻辑必须同时也对**func**字段进行译码，因而修改工作变得更复杂。

**(**思考：如何对主控制逻辑进行修改，以实现**Jr**指令的功能？**)**

**5.12** [5] <§5.4> Explain why it is not possible to modify the single-cycle implementation to implement the load with increment instruction described in Exercise 5.12 without modifying the register file.

答：该题是指用单周期数据通路实现**5.11** 中的自增装入指令（或**5.14**中的交换指令）时，要修改寄存器堆，否则，无法实现。解释为什么？

自增装入指令的功能：

```
lw $rs, L($rt)
addi $rt, $rt, 1
```

交换指令的功能：

使用额外寄存器的情况：

```
add $rtemp, $rs, $zero
add $rs, $rt, $zero
add $rt, $rtemp, $zero
```

不使用额外寄存器的情况：

```
xor $rs, $rs, $rt
xor $rt, $rs, $rt
xor $rs, $rs, $rt
```

可以看出，用硬件实现自增装入指令和交换指令，在一个周期内需至少写两次寄存器。

原单周期数据通路中，一条指令在一个时钟内完成，数据总是在时钟的下降沿被写入到寄存器堆，即本条指令执行的结果总是下条指令开始（即下个时钟到来）时，才被写到寄存器堆中，因此一个周期只能写一次寄存器。

所以，若不修改原来的寄存器堆，则单周期数据通路无法实现自增装入指令和交换指令。

**5.13** [7] <§5.4> Consider the single-cycle datapath in Figure 5.17. A friend is proposing to modify this single-cycle datapath by eliminating the control signal MemtoReg. The multiplexor that has MemtoReg as an input will instead use either the ALUSrc or the MemRead control signal. Will your friend's modification work? Can one of the two signals (MemRead and ALUSrc) substitute for the other? Explain.

答： 朋友的建议是有效的。

从图5-18的表中，可以看出， MemtoReg和MemRead、ALUSrc的取值在R-Type和lw指令时一样，在sw和beq指令时MemtoReg取值任意。

所以控制信号MemtoReg可以用MemRead或ALUSrc来代替。

因为MemRead和ALUSrc的取值在sw指令时取值不同，所以不能相互替代。

| Instruction | RegDst | ALUSrc | Memto-Reg | Reg Write | Mem Read | Mem Write | Branch | ALUOp1 | ALUOp0 |
|---|---|---|---|---|---|---|---|---|---|
| R-format | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| sw | X | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 |
| beq | X | 0 | X | 0 | 0 | 0 | 1 | 0 | 1 |

**5.14** [10] <§5.4> MIPS chooses to simplify the structure of its instructions. The way we implement complex instructions through the use of MIPS instructions is to decompose such complex instructions into multiple simpler MIPS ones. Show how MIPS can implement the instruction `swap $rs, $rt`, which swaps the contents of registers `$rs` and `$rt`. Consider the case in which there is an available register that may be destroyed as well as the care in which no such register exists.

If the implementation of this instruction in hardware will increase the clock period of a single-instruction implementation by 10%, what percentage of swap operations in the instruction mix would recommend implementing it in hardware?

答：使用额外寄存器的情况：

```
add $rtemp, $rs, $zero
add $rs, $rt, $zero
add $rt, $rtemp, $zero
```

不使用额外寄存器的情况：

```
xor $rs, $rs, $rt
xor $rt, $rs, $rt
xor $rs, $rs, $rt
```

假定该指令占**x%**，其他指令占**(1-x)%**
则用硬件实现该指令时，程序执行时间为原来的
**1.1*(x+1-x)=1.1** 倍
用软件实现该指令时，程序执行时间为原来的
**3x+1-x=**（**2x+1**）倍
当**1.1 < 2x+1** 时，硬件实现才有意义
由此可知，**x > 5%**

**5.29** [5] <§5.5> This exercise is similar to Exercise 5.2, but this time consider the effect that the stuck-at-0 faults would have on the *multiple-cycle* datapath in Figure 5.27. Consider each of the following faults:

a. RegWrite = 0
b. MemRead = 0
c. MemWrite = 0
d. IRWrite = 0
e. PCWrite = 0
f. PCWriteCond = 0.

答：若**RegWrite=0**，则所有的需要写结果到寄存器的指令（如：**R-Type**指令、**load**指令等）都不能正确执行，因为寄存器不发生写操作

若**MemRead=0**，则所有指令不能正确执行，因为指令不能读出

若**MemWrite=0**，则**Store**指令不能正确执行，因为存储器不能写入数据

若**IRWrite=0**，则所有指令都不能正确执行，因为**IR**中不能写入新的指令

若**PCWrite=0**，则所有指令都不正确，因为取指令阶段**PC+4**不能写入**PC**

若**PCWriteCond=0**，则**Brabch**指令不能正确执行，因为不能写入转移目标地址到**PC**

此外，

若**Regdst=0**，则所有**R-Type**指令都不能正确执行，因为目的寄存器指定错误；

若**ALUSrcA=0**，则所有**R-Type**、**I-Type**指令、**load/Stroe**指令都不能正确执行，因为第一个操作数总是**PC**的值，而不是寄存器**rs**中的值

若**MemtoReg=0**，则所有的**Load**指令执行错误，因为寄存器写入的是**ALU**输出

……

**5.30** [5] <§5.5> This exercise is similar to Exercise 5.29, but this time consider stuck-at-1 faults (the signal is always 1).

答：若**RegWrite=1**，则所有不需要写结果到寄存器的指令（如：**Store**指令、**Branch**指令等）都不能正确执行，因为寄存器发生了不需要的写操作；

若**MemRead=1**，则除**Load**指令外的其他指令可能出错，读出的错误数据可能影响结果

若**MemWrite=1**，则除**Store**指令外的其他指令不能正确执行，存储器发生了数据写入

若**IRWrite=1**，则所有指令都不能正确执行，因为**IR**中写入的可能不是指令

若**PCWrite=1**，则所有指令都不正确，每个阶段都会写入**PC**，使下条指令地址发生错误

若**PCWriteCond=1**，则除**Brabch**指令外的所有指令都可能出错，因为**PC**中可能会写入转移目标地址

此外，

若**Regdst=1**，则**Load**指令和**I-Type**指令都不能正确执行，因为目的寄存器指定错误

若**ALUSrcA=1**，则所有**R-Type**指令和**Branch**指令都不能正确执行，因为第二个操作数为立即数扩展，而不是寄存器数据；

若**MemtoReg=1**，则所有**R-Type**指令执行错误，因为寄存器写入的是存储器读出的数据

……

**5.32** 15] <§5.5> We wish to add the instruction `lui` (load upper immediate) described in Chapter 3 to the multicycle datapath described in this chapter. Use the same structure of the multicycle datapath of Figure 5.28 on page 323 and show the necessary modifications to the finite state machine of Figure 5.38 on page 339. You may find it helpful to examine the execution steps shown on pages 325 through 329 and consider the steps that will need to be performed to execute the new instruction. How many cycles are required to implement this instruction?

答： lui指令为I-Type格式：

| 001111 | 00000 | Rt | Imm16 |
|---|---|---|---|

其功能是把16位立即数送到寄存器Rt的高16位，低16位为0。只要对原扩展器稍加修改使输出为Imm16x$2^{16}$，然后和$zero相加，结果送Rt。具体修改如下：

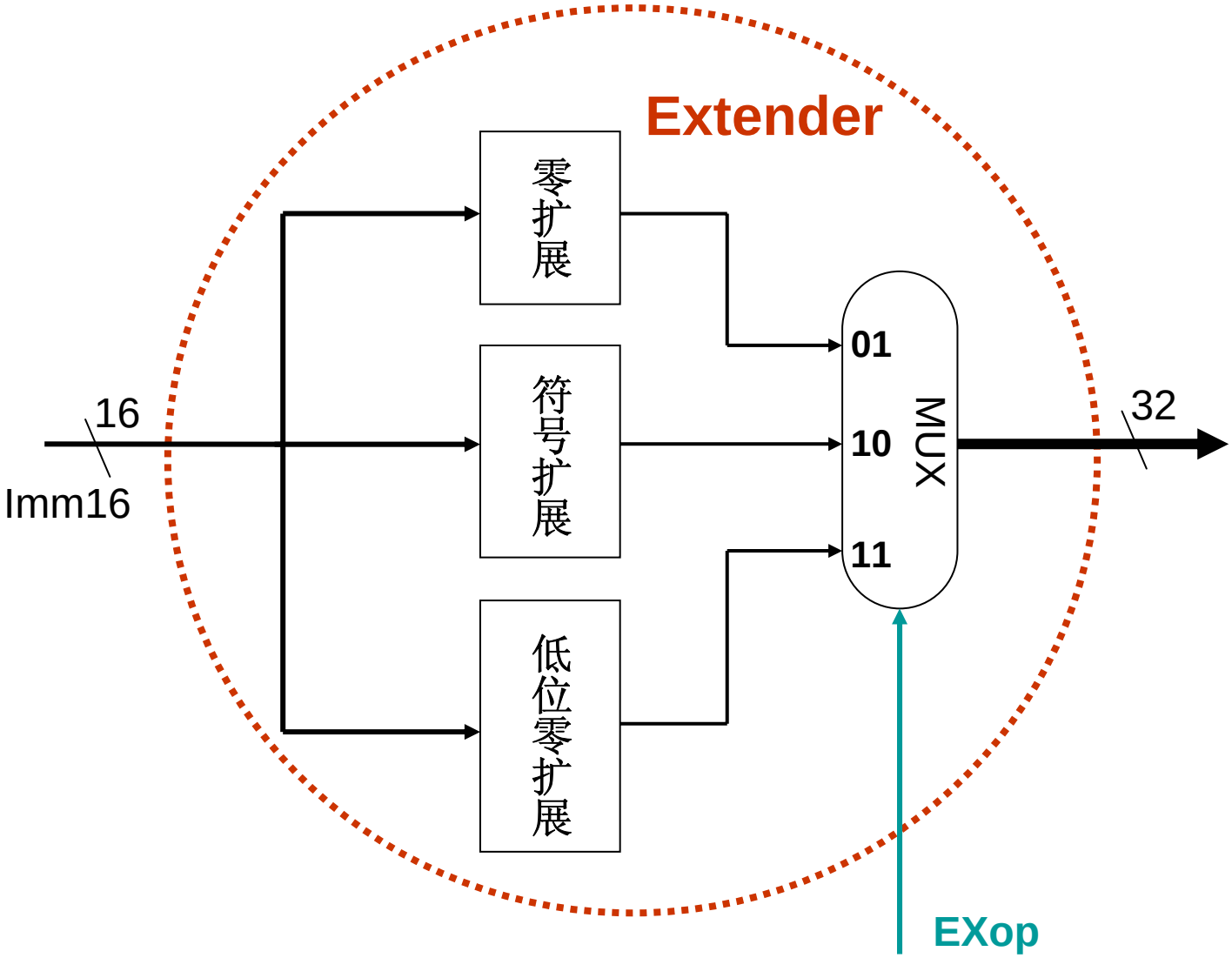
（1）在原来的符号扩展器上进行修改，使其具有三种扩展功能(见下页)；

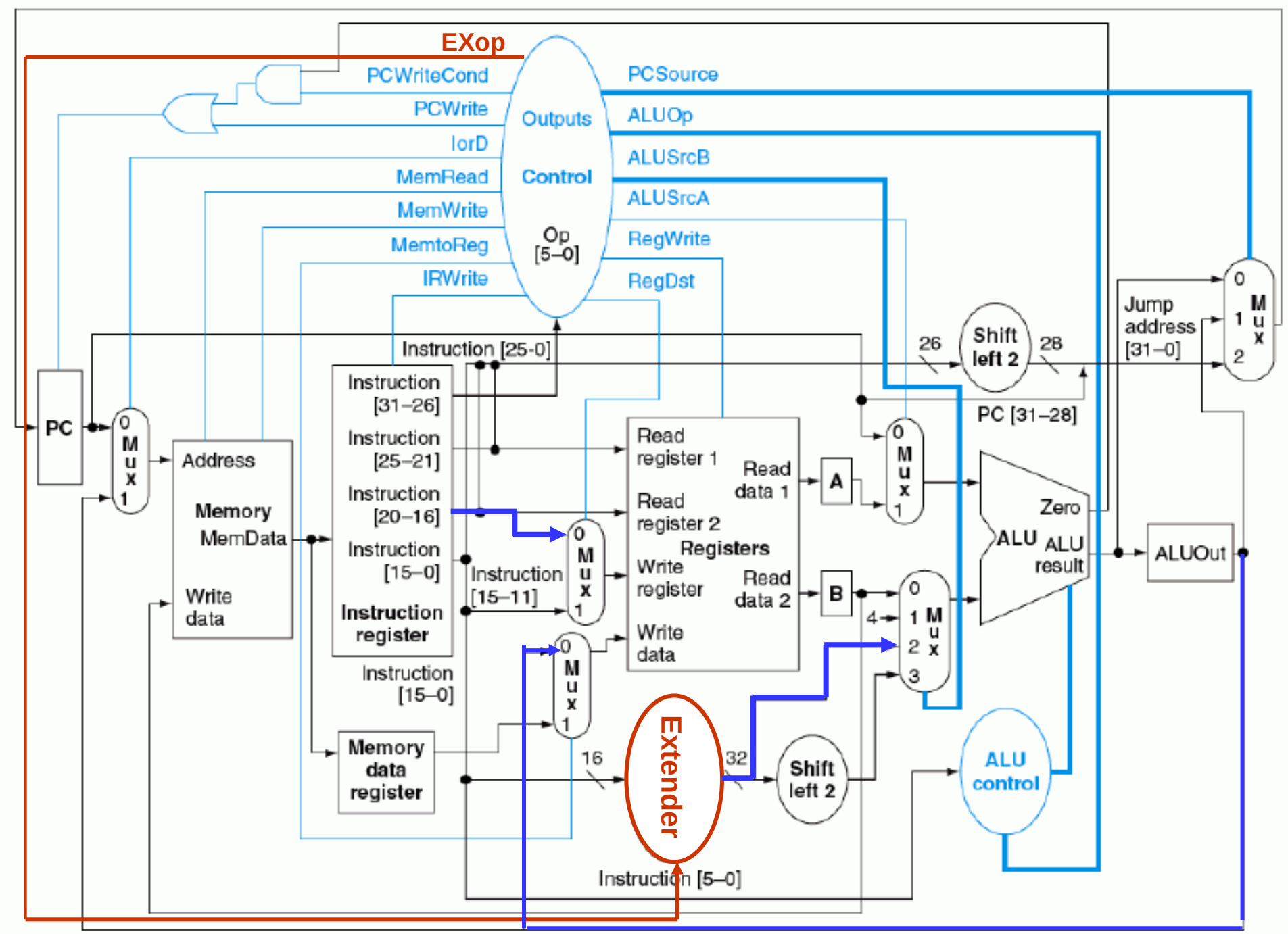
（2）增加一个控制信号EXop，扩展器能根据控制信号EXop的值进行扩展操作

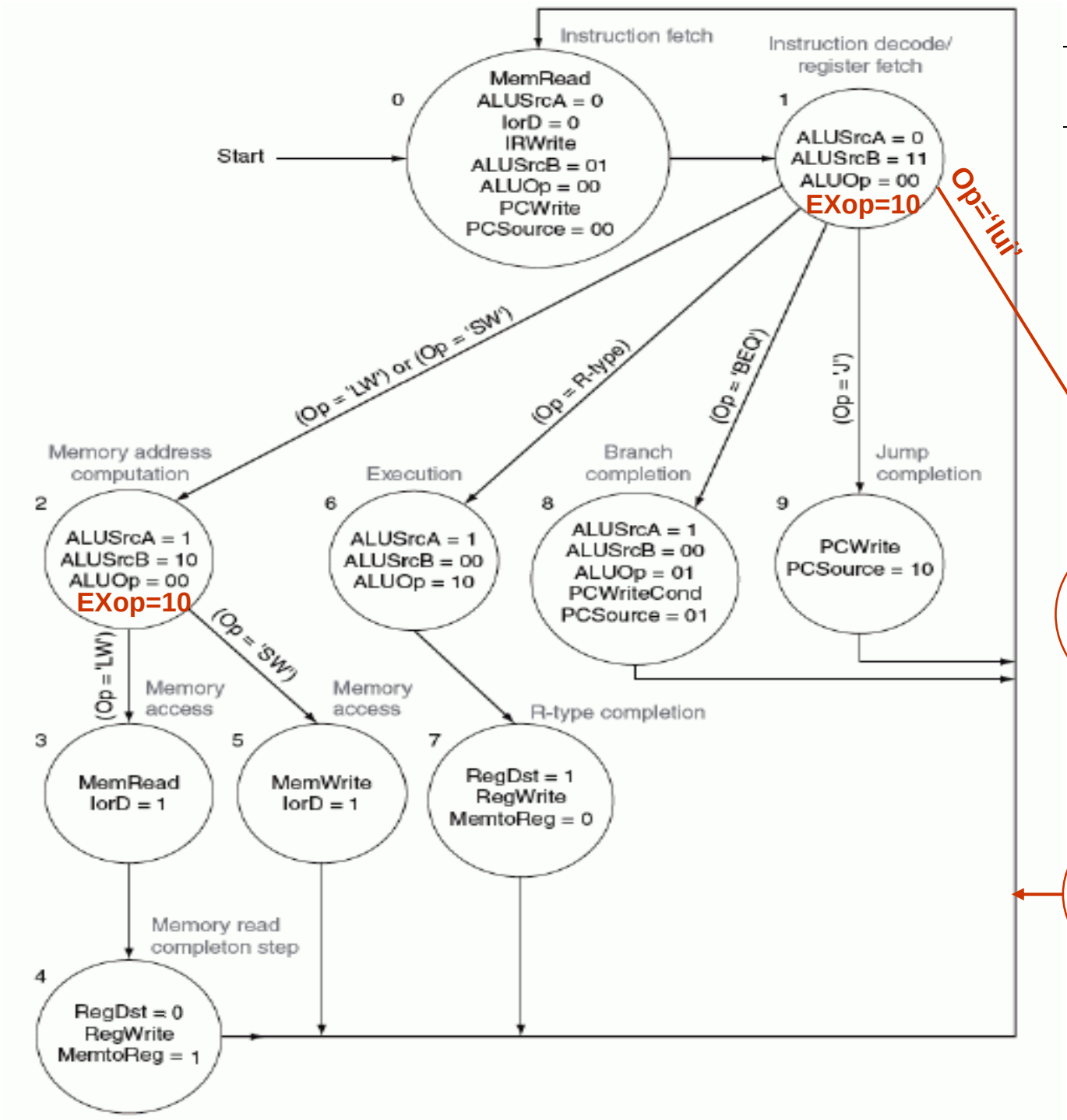
EXop=01: 零扩展；EXop=10: 符号扩展；EXop=11: 低位零扩展；

（3）控制信号的取值除EXop之外，其他控制信号类似于R-Type指令；

（4）由于扩展器进行了修改，原来ALUSrcB=10和ALUSrcB=11的地方要增加一个EXop控制信号，并设置正确的值。


Extender
零扩展
符号扩展
低位零扩展
01
10
11
MUX
16
Imm16
32
EXop


EXop
PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
Instruction [25-0]
26
Shift
left 2
28
PC [31–28]
Jump
address
[31–0]
PC
Address
Memory
MemData
Write
data
Instruction
[31–26]
Instruction
[25–21]
Instruction
[20–16]
Instruction
[15–0]
Instruction
register
Instruction
[15–11]
Instruction
[15–0]
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
Zero
ALU
ALU
result
ALUOut
Memory
data
register
16
Extender
32
Shift
left 2
ALU
control
Instruction [5–0]




实现该指令需要
4个时钟周期！

**5.35** [15] <§5.5> Consider a change to the multiple-cycle implementation that alters the register file so that it has only one read port. Describe (via a diagram) any additional changes that will need to be made to the datapath in order to support this modification. Modify the finite state machine to indicate how the instructions will work, given your new datapath.

分析如下：如果只有一个读口，那么原来A和B可以同时读，现在只能先读一个到A，然后再读一个到B。

A和B共用一个读地址端口，所以要加一个多路选择器，用于选择不同的寄存器号；读出的数据只有一个端口，可能要送到A，也可能要送到B，所以要加一个控制信号，以确定该写到A还是B中。

增加的两个控制信号为：

（1）**RegRead:** 用于控制多路选择器,为0则读口地址为Rs，为1则为Rt

（2）**AWrite:** 用于控制读出数据写入A还是B, 为1写入A，为0则写入B

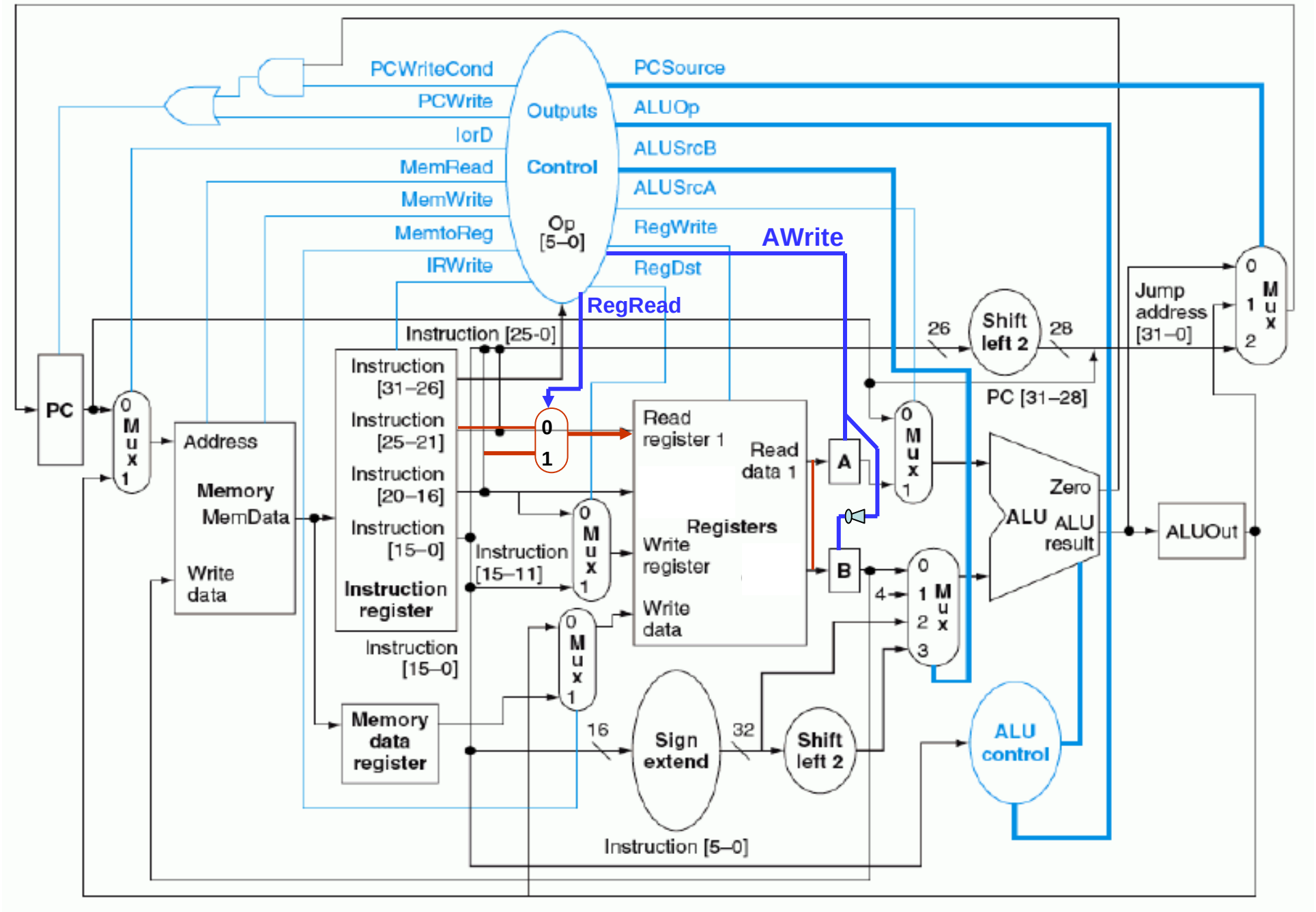
可以有三种做法：（在原有限状态机基础上考虑）

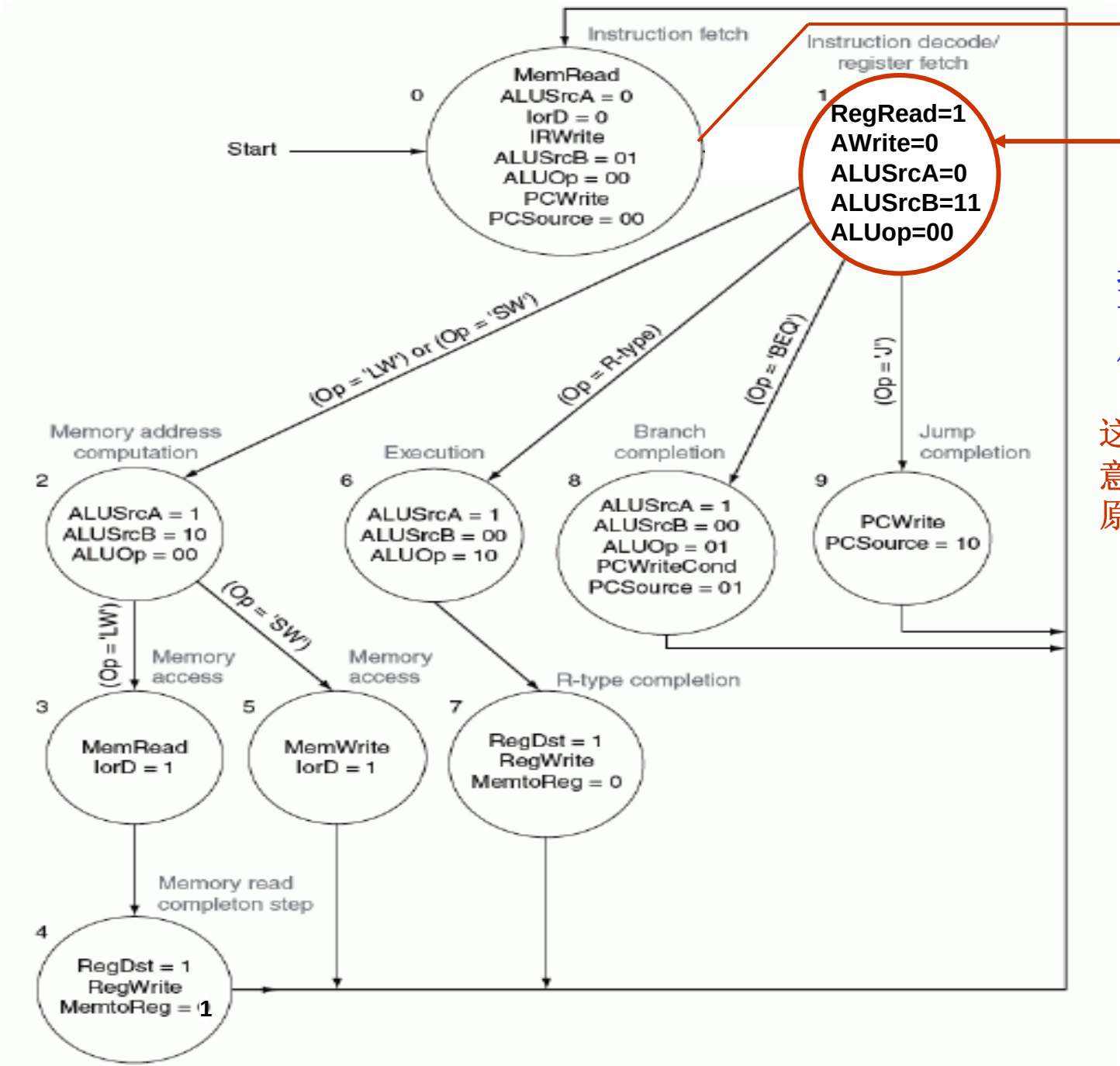
**(1)** 原来的周期1分成两个周期，分别读A和B，并在其中一个周期中完成转移地址计算

**(2)** 在原来的周期1中先读A，这样在R-Type和Branch指令中要增加一个周期来读B

**(3)** 在原来的周期1中先读B，这样，load/store和Ori指令都要增加一个周期重新读A，R-Type和Branch指令中也要增加一个周期来读A

不同做法得到的CPI不同！　　哪个CPI最大？哪个最小？　　做法(2)的CPI最小！


PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
AWrite
RegRead
Instruction [25-0]
26
Shift
left 2
28
Jump
address
[31–0]
PC [31–28]
PC
Address
Memory
MemData
Write
data
Instruction
[31–26]
Instruction
[25–21]
Instruction
[20–16]
Instruction
[15–0]
Instruction
register
Instruction
[15–0]
Instruction
[15–11]
Read
register 1
Read
data 1
Registers
Write
register
Write
data
A
B
4
Zero
ALU
ALU
result
ALUOut
Memory
data
register
16
Sign
extend
32
Shift
left 2
ALU
control
Instruction [5–0]




投机计算转移地址可以在两个周期的任意一个中进行。

这是第(1)种做法的示意图，每条指令都比原来增加一个周期！

**5.36** [15] <§5.5> Two important parameters control the performance of a processor: cycle time and cycles per instruction. There is an enduring trade-off between these two parameters in the design process of microprocessors. While some designers prefer to increase the processor frequency at the expense of large CPI, other designers follow a different school of thought in which reducing the CPI comes at the expense of lower processor frequency.

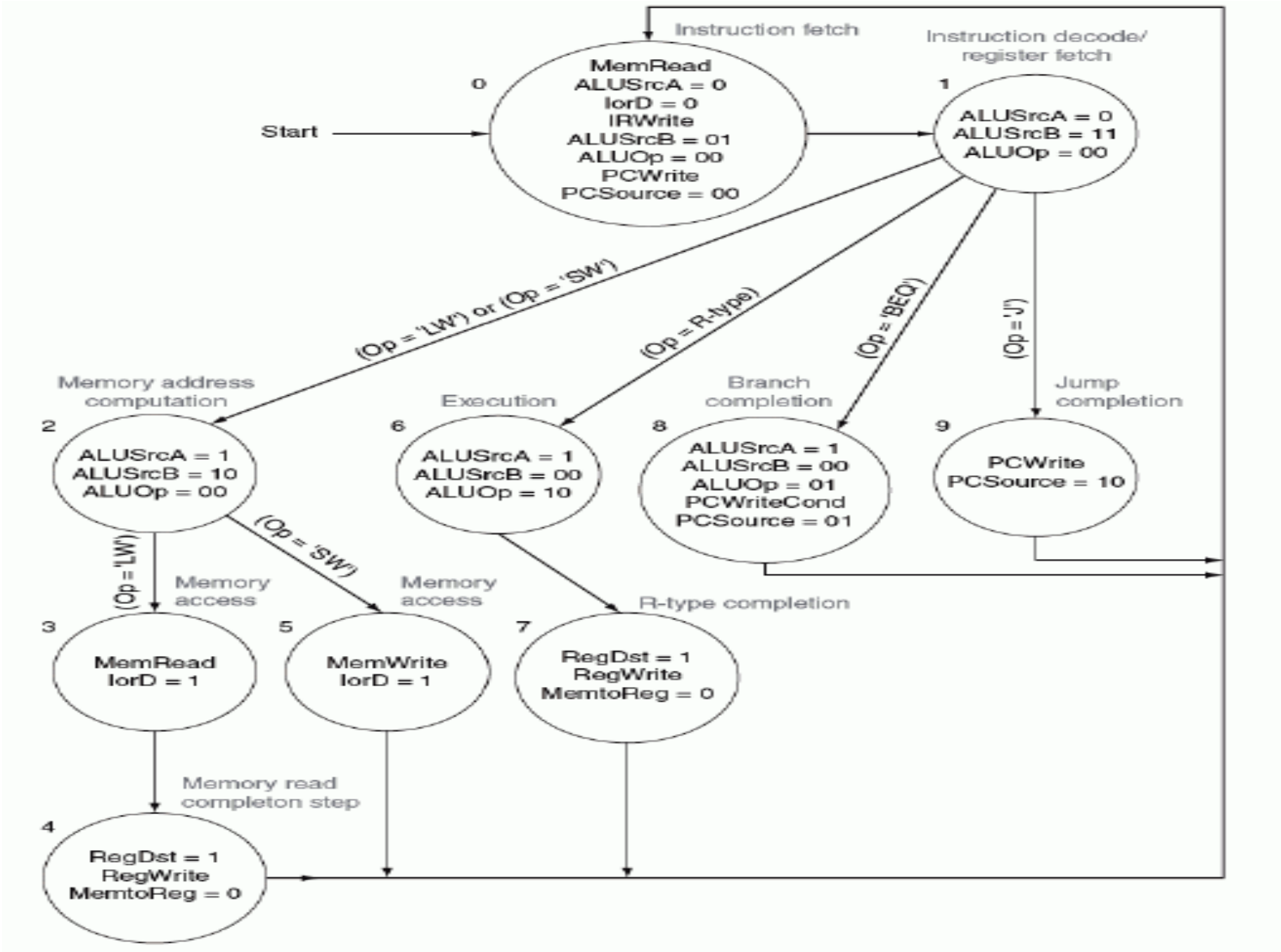
Consider the following machines, and compare their performance using the SPEC CPUint 2000 data from Figure 3.26 on page 228.

M1: The multicycle datapath of Chapter 5 with a 1 GHz clock.

M2: A machine like the multicycle datapath of Chapter 5, except that register updates are done in the same clock cycle as a memory read or ALU operation. Thus in Figure 5.38 on page 339, states 6 and 7 and states 3 and 4 are combined. This machine has an 3.2 GHz clock, since the register update increases the length of the critical path.

M3: A machine like M2 except that effective address calculations are done in the same clock cycle as a memory access. Thus states 2, 3, and 4 can be combined, as can 2 and 5, as well as 6 and 7. This machine has a 2.8 GHz clock because of the long cycle created by combining address calculation and memory access.

Find out which of the machines is fastest. Are there instruction mixes that would make another machine faster, and if so, what are they?


Instruction fetch
Instruction decode/
register fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address
computation
Execution
Branch
completion
Jump
completion
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
Memory
access
R-type completion
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
7
RegDst = 1
RegWrite
MemtoReg = 0
Memory read
completon step
4
RegDst = 1
RegWrite
MemtoReg = 0

答：图3-26所示的SPECINT2000混合指令频率为：

| | | | |
|---|---|---|---|
| **Load: 25%** | **5** | **4** | **3** |
| **Store:10%** | **4** | **4** | **3** |
| **Bran: 11%** | **3** | **3** | **3** |
| **Jump:2%** | **3** | **3** | **3** |
| **ALU: 52%** | **4** | **3** | **3** |

$CPI_{M1}$=25%x5+10%x4+11%x3+2%x3+52%x4=4.12

$CPI_{M2}$=25%x4+10%x4+11%x3+2%x3+52%x3=3.35

$CPI_{M3}$=25%x3+10%x3+11%x3+2%x3+52%x3=3

从有限状态图分析，得知：

**M1中上述各类指令的CPI分别为5、4、3、3、4**

**M2中上述各类指令的CPI分别为4、4、3、3、3**

**M3中上述各类指令的CPI分别为3、3、3、3、3**

$MIPS_{M1}$=1G / 4.12 = 242.7

$MIPS_{M2}$=3.2 G / 3.35 =955.2

$MIPS_{M3}$=2.8 G / 3 = 933.3

**M2和M3对M1作了不同的改变，M2的做法效果更好，速度最快！**

**但当所有指令都是load/store指令时，M3的速度最快！**

**5.37** [20] <§5.5> Your friends at $C^3$ (Creative Computer Corporation) have determined that the critical path that sets the clock cycle length of the multicycle datapath is memory access for loads and stores (not for fetching instructions). This has caused their newest implementation of the MIPS 30000 to run at a clock rate of 4.8 GHz rather than the target clock rate of 5.6 GHz. However, Clara at $C^3$ has a solution. If all the cycles that access memory are broken into two clock cycles, then the machine can run at its target clock rate.

Using the SPEC CPUint 2000 mixes shown in Chapter 3 (Figure 3.26 on page 228), determine how much faster the machine with the two-cycle memory accesses is compared with the 4.8 GHz machine with single-cycle memory access. Assume that all jumps and branches take the same number of cycles and that the set instructions and arithmetic immediate instructions are implemented as R-type instructions. Would you consider the further step of splitting instruction fetch into two cycles if it would raise the clock rate up to 6.4 GHz? Why?

考虑思路同前面一题，也是先算出各种指令的**CPI**，然后根据机器的时钟频率算出**MIPS**数，对**MIPS**数进行比较。

只不过两题中引起机器性能变化的原因不同而已。

答：图3-26所示的SPECINT2000混合指令频率为：

| | | |
|---|---|---|
| Load: 25% | 6 | 5 |
| Store:10% | 5 | 4 |
| Bran: 11% | 3 | 3 |
| Jump:2% | 3 | 3 |
| ALU: 52% | 4 | 4 |

$CPI_{M1}$=25%x6+10%x5+11%x3+2%x3+52%x4=4.47

$CPI_{M2}$=25%x5+10%x4+11%x3+2%x3+52%x4=4.12

$MIPS_{M1}$=5.6G / 4.47 = 1253

$MIPS_{M2}$=4.8G / 4.12 =1165

数据存取为双周期的机器**M1**中上述各类指令的**CPI**分别为：**6、5、3、3、4**

数据存取为单周期的机器**M2**中上述各类指令的**CPI**分别为：**5、4、3、3、4**

由此可见，数据存取改为双周期的做法效果较好。

进一步把取指令分成两个周期的机器**M3**的各类指令的**CPI**分别为：**7、6、4、4、5**

**$CPI_{M3}$=25%x7+10%x6+11%x4+2%x4+52%x5=5.47**

（实际上就是每个指令都加一个时钟周期，所以**$CPI_{M3}=CPI_{M1}+1=5.47$**

$MIPS_{M3}$=6.4 G / 5.47 = 1170

由此可见，进一步把取指令改为双周期的做法使**MIPS**数变小了，所以不可取。

为什么两者都使时钟频率提高**0.8G**，但效果却不同？

因为数据存取只涉及到**load/Store**指令，而指令存取涉及到所有指令！

**5.38** [20] <§5.5> Suppose there were a MIPS instruction, called `bcmp`, that compares two blocks of words in two memory addresses. Assume that this instruction requires that the starting address of the first block is in register `$t1` and the starting address of the second block is in `$t2`, and that the number of words to compare is in `$t3` (which is `$t3≥0`). Assume the instruction can leave the result (the address of the first mismatch or zero if a complete match) in `$t1` and/or `$t2`. Furthermore, assume that the values of these registers as well as registers `$t4` and `t5` can be destroyed in executing this instruction (so that the registers can be used as temporaries to execute the instruction).

Write the MIPS assembly language program to implement (emulate the behavior of) block compare. How many instructions will be executed to compare two 100-word blocks? Using the CPI of the instructions in the multicycle implementation, how many cycles are needed for the 100-word block compare?

参考答案如下：

```
                beq   $t3, $zero, done
compare:        lw    $t4, 0($t1)
                lw    $t5, 0($t2)
                bne   $t4, $t5, done
                addi  $t1, $t1, 4
                addi  $t2, $t2, 4
                addi  $t3, $t3, -1
                bne   $t3, $zero, compare
                addi  $t2, $zero, 0
done:
```

假定比较次数为**100**，则所需的指令数为：**1+100x7+1=702**条指令

其中，**load**指令为：**100x2=200**条， 周期数为**5x200**条**=1000**

**branch**指令为：**1+2x100=201**条， 周期数为**3x201**条**=603**

**addi**指令为：**1+3x100=301**条， 周期数为**4x301**条**=1204**

所以，总周期数为**1000+603+1204=2807**

**5.50** [6] <§5.6> Exceptions occur when a control flow change is required to handle an unexpected event in the processor. How can the cause and the instruction that caused the exception, be represented by the hardware in a MIPS machine? Give two examples for conditions that a processor can handle by restarting execution of instructions after handling the exception, and two others for exceptions that lead to program termination.

参考答案：

（1）异常原因可以记录在**Cause**寄存器中

（2）断点（发生异常的指令或下条指令的地址）存放在**EPC**中

（3）异常有三种：故障、自陷、终止

故障：由正在执行的指令产生的使当前指令无法继续执行的“异常事件”。处理完后回到发生故障的指令重新执行（如：缺页）或终止程序执行（如：溢出、除数为0、非法操作码、保护错等）

自陷：人为在程序中先设定一条特殊的访管或自陷指令。如**80x86**中的指令**“INT n”**执行到这条指令时，**CPU**自动中止正在执行的程序，转到一个特定的内核管理程序去执行，执行完后，回到这条指令后面的一条指令（断点处）开始执行。

终止：发生不可恢复的硬件致命错误而使机器无法继续执行指令。如：硬件线路故障、电源掉电等。此时，系统被终止并重新启动操作系统。

**5.51** [6] <§5.6> Exception detection is an important aspect of exception handling. Try to identify the cycle in which the following exceptions can be detected for the multicycle datapath in Figure 5.28 on page 323.

Consider the following exceptions:

a. Divide by zero exception (suppose we use the same ALU for division in one cycle, and that it is recognized by the rest of the control)

c. Invalid instruction

d. External interrupt

e. Invalid instruction memory address

f. Invalid data memory address

参考答案：

**a. “除数为0”异常可以在取数/译码周期进行检测**

**b. “溢出”异常可以在R-Type指令的完成周期进行检测**

**C. “无效指令”异常可以在取数/译码周期进行检测**

**d. “外部中断”异常可以在每条指令的完成周期进行检测**

**e. “无效指令地址”异常可以在取指令周期检测**

**f. “无效数据地址”异常可以在load/store指令的地址计算周期检测**

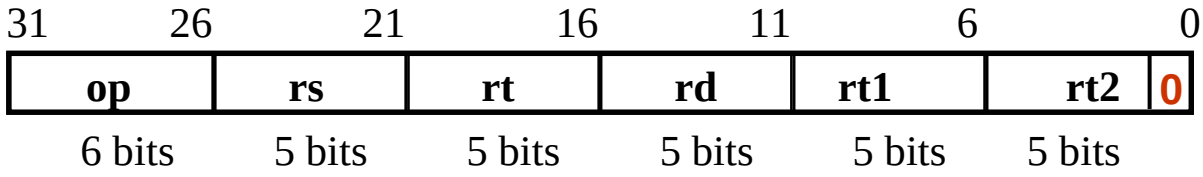
**5.53** [30] <§5.7> Microcode has been used to add more powerful instructions to an instruction set; let's explore the potential benefits of this approach. Devise a strategy for implementing the `bcmp` instruction described in Exercise 5.38 using the multicycle datapath and microcode. You will probably need to make some changes to the datapath in order to efficiently implement the `bcmp` instruction. Provide a description of your proposed changes and describe how the `bcmp` instruction will work. Are there any advantages that can be obtained by adding internal registers to the datapath to help support the `bcmp` instruction? Estimate the improvement in performance that you can achieve by implementing the instruction in hardware (as opposed to the software solution you obtained in Exercise 5.38) and explain where the performance increase comes from.

包括以下几个方面的设计：

- 指令的汇编形式、机器码格式（与原有指令兼容）
- 指令的功能（用流程图和**RTL**表示）
- 数据通路修改
- 控制信号增加或修改
- 有限状态机的修改

指令的汇编形式：**bcmp rs, rd, rc, rt1, rt2**

指令的机器码格式：




指令的功能：

比较个数由**rc**指出，如果**rc=0**则什么都不做，继续执行下条指令，否则：

**rs**和**rd**所指内存单元依次顺序比较其内容，直到发生以下情况：

（**1**）有一对数据不相等，此时，返回不相等数据对的地址在**rs**和**rd**中

（**2**）所有数据都相等，此时，返回**0**存放在**rs**中

**rt1**和**rt2**是临时寄存器，在指令执行过程中他们和**rs**、**rd**、**rc**都会被破坏。

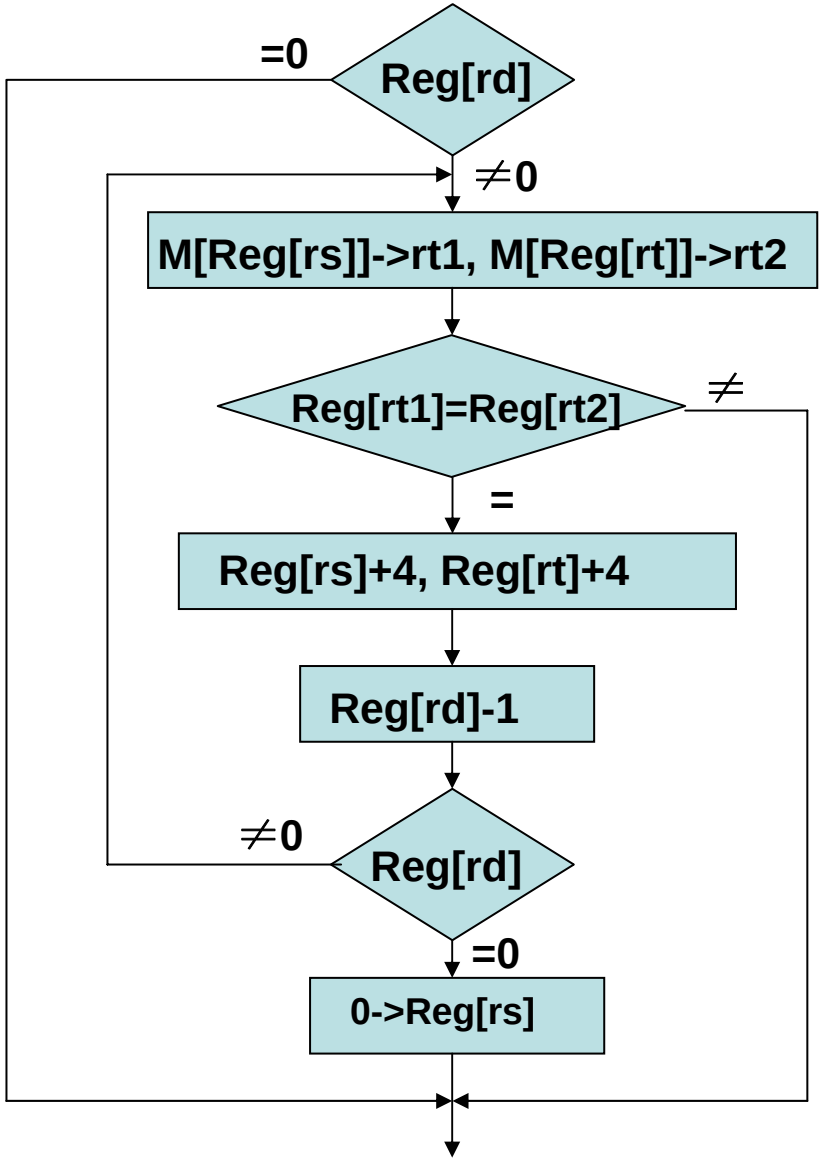
用一个程序段来描述为：

```
        beq   rd, $zero, done
comp:   lw    rt1, 0(rs)
        lw    rt2, 0(rt)
        bne   rt1, rt2, done
        addi  rs, rs, 4
        addi  rt, rt, 4
        addi  rd, rd, -1
        bne   rd, $zero, compare
        addi  rs, $zero, 0
done:
```

指令功能复杂，在给出RTL描述之前，先画出流程图：

指令功能用RTL描述为：（除公共操作外）

```
if Reg[rd]=0 then Exec-Next else
  loop:   Addr ← R[rs] , R[rt1] ← Mem[Addr]
          Addr ← R[rt] , R[rt2] ← Mem[Addr]
  if R[rt1]≠R[rt2] then Exec-Next else
    R[rs] ← R[rs]+4,  R[rt] ← R[rt]+4
    R[rd] ← R[rd]-1
      if R[rd]=0 then
        R[rs] ← 0, Exec_Next
      else
        go to loop
```




数据通路的修改：

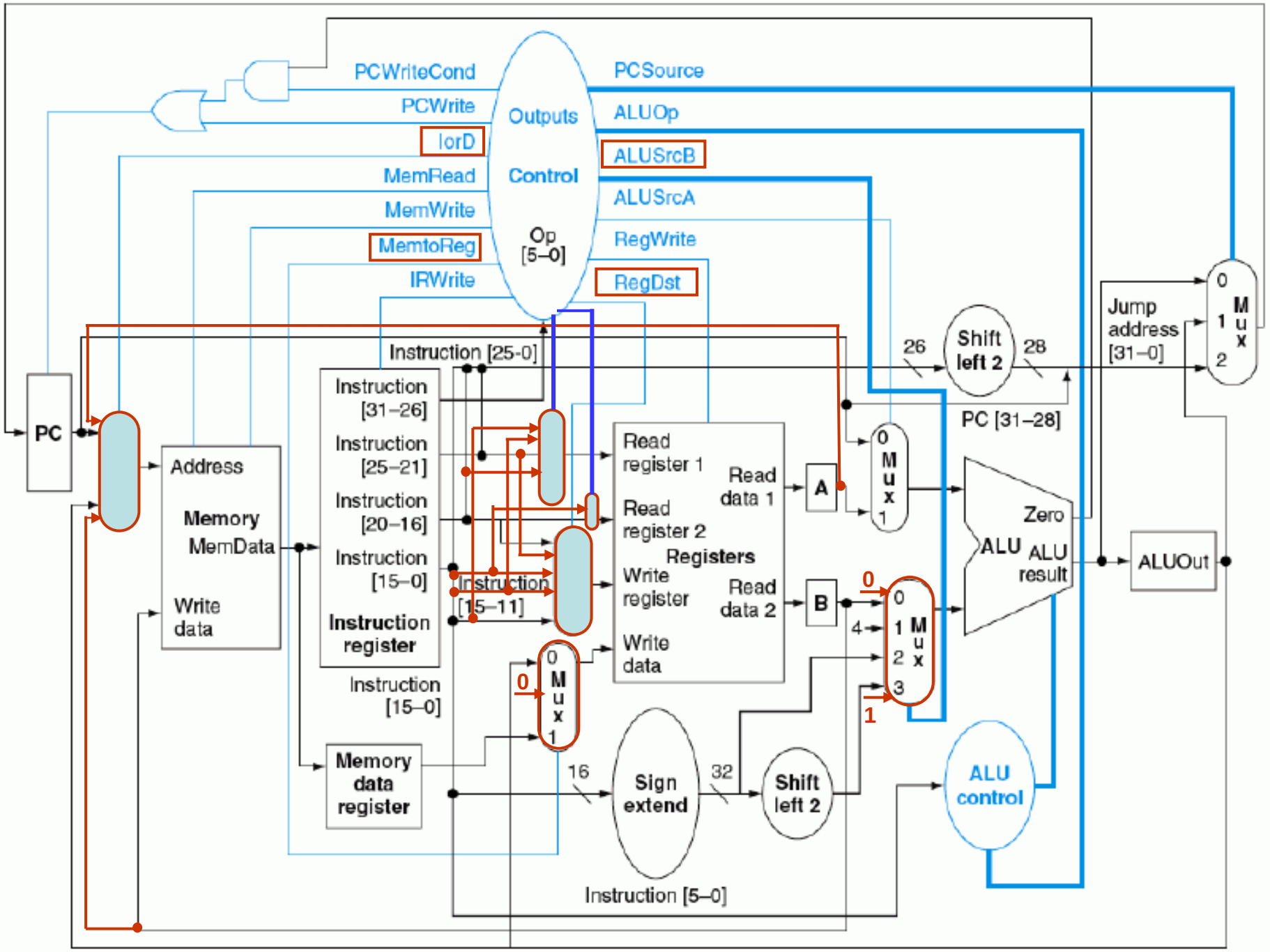
- 在Mem的Addr处改MUX以选择不同的取数地址
- 在Reg的WAddr处改MUX以选择不同的存数地址
- 在Reg的RAddr加MUX以选择不同的读数地址
- 在Reg的WData改MUX以选择不同的写数据
- 在ALUSrcB处加“1”和“0”，以执行“-1/-0”操作

控制信号的增加或修改：

- 对新加或更改的MUX的控制信号进行修改或增加

有限状态机的修改：

- 增加若干新状态，每个状态在一个周期内完成


PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
Instruction [25-0]
26
Shift
left 2
28
PC [31–28]
Jump
address
[31–0]
M
u
x
0
1
2
PC
Address
Memory
MemData
Write
data
Instruction
[31–26]
Instruction
[25–21]
Instruction
[20–16]
Instruction
[15–0]
Instruction
register
Instruction
[15–0]
Instruction
[15–11]
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
Zero
ALU
ALU
result
ALUOut
Memory
data
register
16
Sign
extend
32
Shift
left 2
ALU
control
Instruction [5–0]

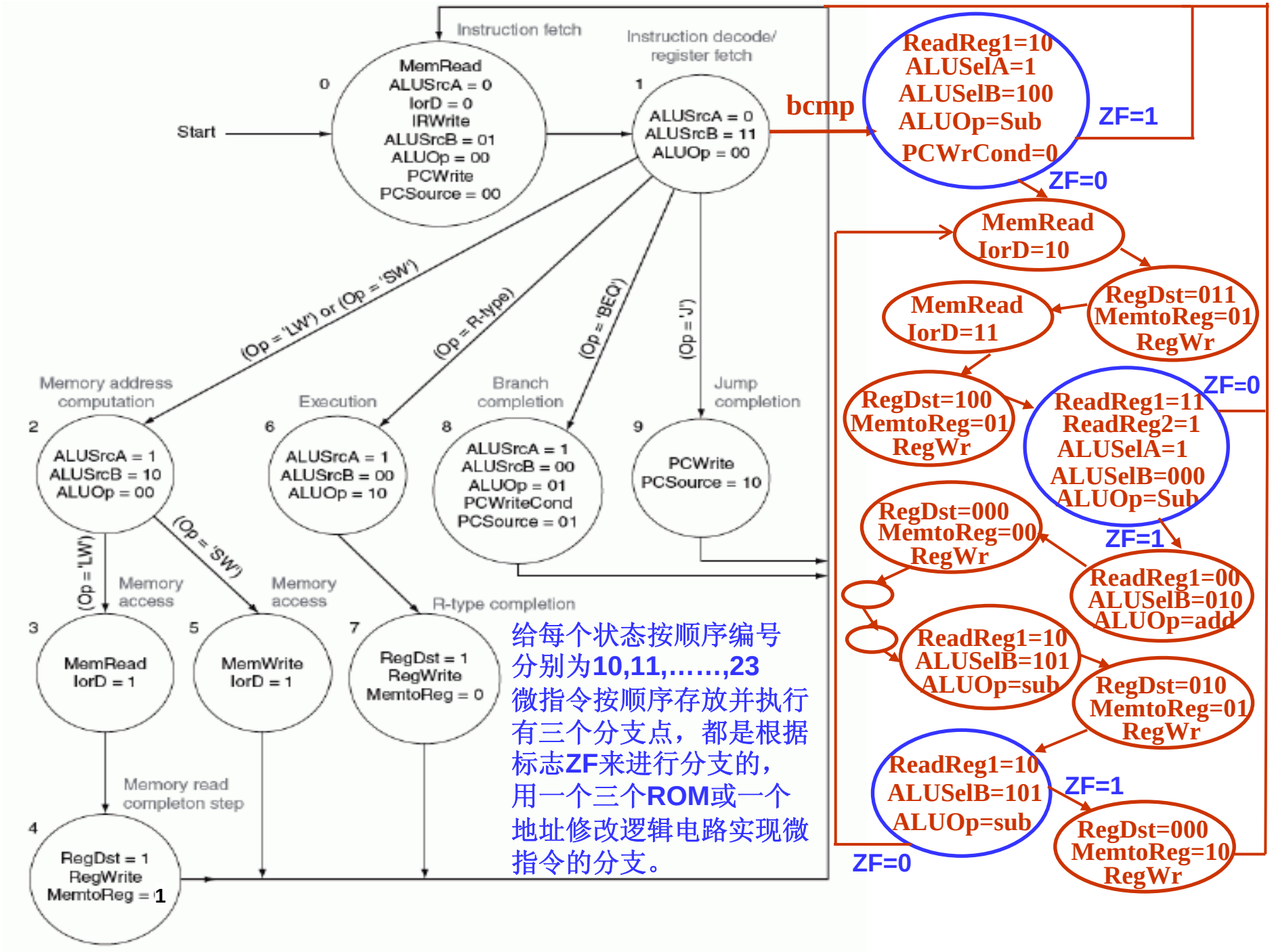
以下四个控制信号要修改

- **IorD**
  - 0: PC->Address
  - 1: ALUout->Address
  - 2: A->Address
  - 3: B->Address
- **RegDst**
  - 0: Inst[25-21]->WriteReg
  - 1: Inst[20-16]->WriteReg
  - 2: Inst[15-11]->WriteReg
  - 3: Inst[10- 6]->WriteReg
  - 4: Inst[ 5- 1]->WriteReg
- **ALUSrcB**
  - 0:
  - 1:
  - 2:
  - 3:

  } 定义不变
  - 4: 0->ALUSrcB
  - 5: 1->ALUSrcB
- **MemtoReg**
  - 0: Memory Data ->Reg'Write Data
  - 1: ALUout -> Reg'Write Data
  - 2: 0-> Reg'Write Data

新加两个控制信号

- **ReadReg1：控制Read Reg1的MUX**
  - 0: Inst[25-21]->ReadReg1
  - 1: Inst[20-16]-> ReadReg1
  - 2: Inst[15-11]-> ReadReg1
  - 3: Inst[10- 6]-> ReadReg1
- **ReadReg2：控制Read Reg2的MUX**
  - 0: Inst[20-16]-> ReadReg2
  - 1: Inst[5- 1]-> ReadReg2


Instruction fetch
Instruction decode/
register fetch
Start
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
bcmp
ReadReg1=10
ALUSelA=1
ALUSelB=100
ALUOp=Sub
PCWrCond=0
ZF=1
ZF=0
MemRead
IorD=10
RegDst=011
MemtoReg=01
RegWr
MemRead
IorD=11
RegDst=100
MemtoReg=01
RegWr
ZF=0
ReadReg1=11
ReadReg2=1
ALUSelA=1
ALUSelB=000
ALUOp=Sub
RegDst=000
MemtoReg=00
RegWr
ZF=1
ReadReg1=00
ALUSelB=010
ALUOp=add
ReadReg1=10
ALUSelB=101
ALUOp=sub
RegDst=010
MemtoReg=01
RegWr
ReadReg1=10
ALUSelB=101
ALUOp=sub
ZF=1
RegDst=000
MemtoReg=10
RegWr
ZF=0
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address
computation
Execution
Branch
completion
Jump
completion
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
Memory
access
R-type completion
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
7
RegDst = 1
RegWrite
MemtoReg = 0
Memory read
completon step
4
RegDst = 1
RegWrite
MemtoReg = 1
给每个状态按顺序编号
分别为10,11,……,23
微指令按顺序存放并执行
有三个分支点，都是根据
标志ZF来进行分支的，
用一个三个ROM或一个
地址修改逻辑电路实现微
指令的分支。

分支1处：ZF=0则Next μaddr = 0；ZF=1则Next μaddr = μaddr+1
分支2处：ZF=0则Next μaddr = μaddr+1；ZF=1则Next μaddr = 0
分支3处：ZF=0则Next μaddr = 01011；ZF=1则Next μaddr = μaddr+1

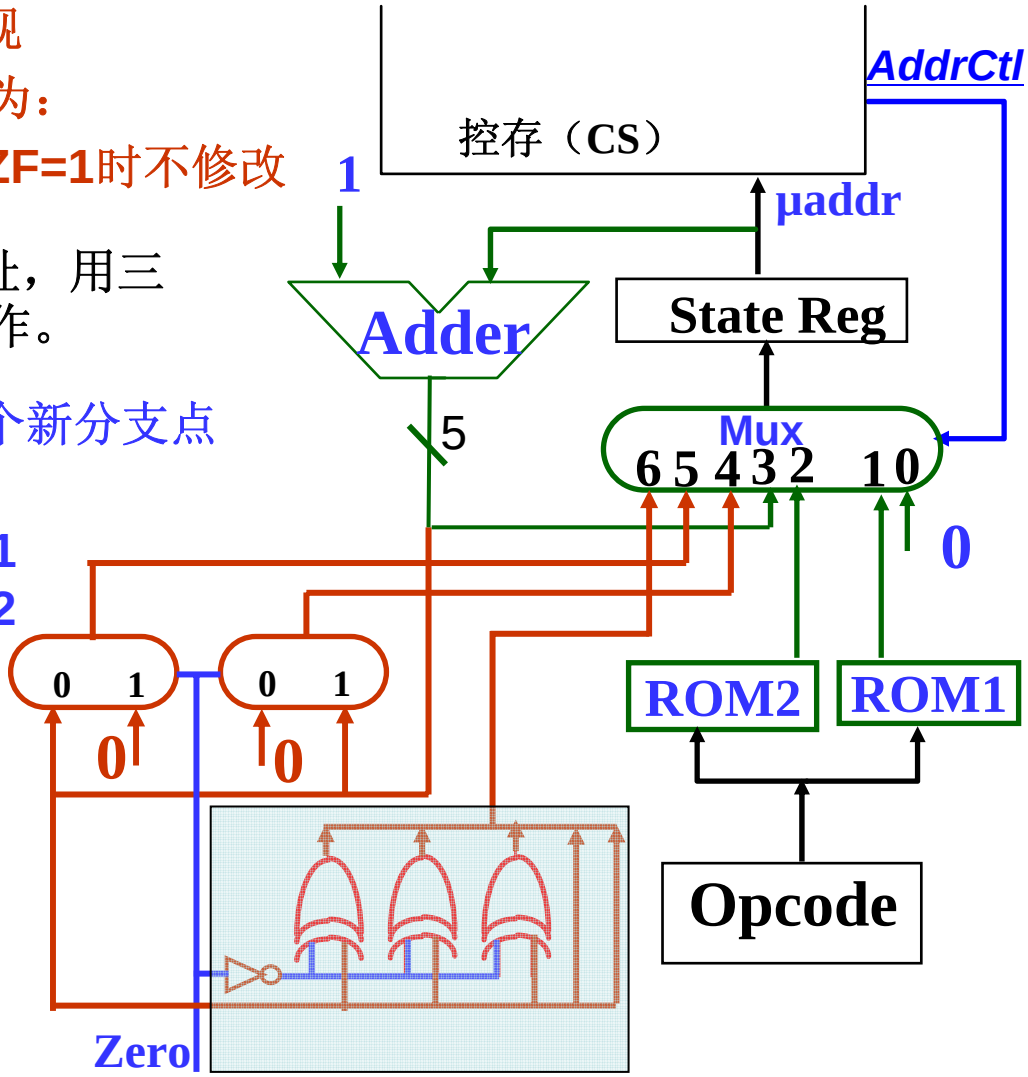
分支1和分支2处可以各用一个MUX实现
分支3处用一个地址修改逻辑，其功能为：
当ZF=0时，将输入10111改为01011, ZF=1时不修改

地址修改逻辑中要改变的是前三位地址，用三个“异或”门分别与“ZF”的反码进行操作。

对AddrCtl控制信号进行修改，实现三个新分支点

000 Next μaddr = 0
001 Next μaddr = dispatch ROM 1
010 Next μaddr = dispatch ROM 2
011 Next μaddr = μaddr + 1
100 Next μaddr = 分支1处输出
101 Next μaddr = 分支2处输出
110 Next μaddr = 分支3处输出

在有限状态图中每个状态里加上AddrCtl控制信号的取值即可。



假定比较次数为n≠0，根据有限状态机可知：

最坏的情况下，用硬件实现该指令所需的时钟周期数为2+1+12n+1=4+12n

最坏的情况下，用软件实现该指令所需的时钟周期数计算如下：

load指令为：2n条， 周期数为5x2n=10n

branch指令为：(1+2n)条， 周期数为3x(1+2n)=3+6n

addi指令为：(1+3n)条， 周期数为4x(1+3n)=4+12n

所以，总周期数为10n+3+6n+4+12n=7+28n

由此可知：用多周期数据通路硬件实现块比较指令比软件至少快一倍多！
原因是什么？

主要有两个原因：

（1）软件方式下，循环内每条指令都要取指令、译码/取数；而硬件实现时不需要

（2）软件方式下，每条指令保存结果，下条指令要用时再取；而硬件方式下，中间数据可以直接使用

思考1：如果不是用多周期数据通路，而是单周期数据通路，情况怎样？

用软件实现更合算，因为时钟周期宽度以最复杂指令为准！

思考2：如果rt1和rt2用内部寄存器的话，有没有好处？

可以简化数据通路、减少控制信号线、减少时钟数。给出整个解决方案。