# 计算机组织与系统结构

## 设计多周期控制器
## 微程序和中断

## Designing a Multiple Cycle Controller
## Microprogramming and Exception

(第十讲)

程 旭

2020.12.3



北京大学微处理器研究开发中心

# 多周期实现概述


- 单周期处理器的问题根源:
  - 对于最慢的指令，周期时间必须足够长
- 解决方案:
  - 将指令处理分为更小的步骤
  - 每个周期执行一步（而不是整个指令）
    - 周期时间： 执行最长步所需的时间
    - 使所有的步骤尽量具有相同的长度
  - 这是多周期处理器的本质所在
- 多周期处理器的优点:
  - 周期时间非常短
  - 不同的指令需要不同的周期数来完成
    - 装入需要 5 个周期
    - 跳转仅仅需要3个周期
  - 允许每条指令多次使用同一个功能部件

# 控制概述

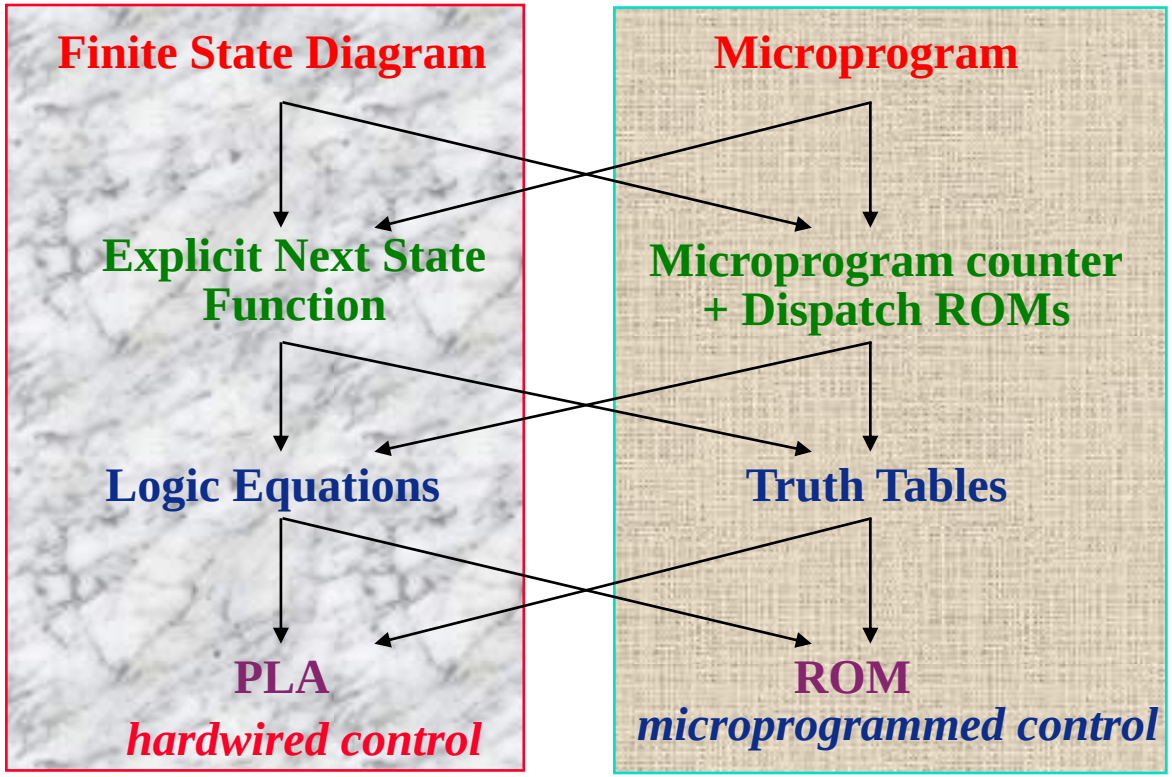

- 可以采用不同的初始表示来设计控制。然后，可以独立地择取序列控制，和如何表示逻辑功能；最后，采用结构化逻辑技术，从多种方法中，选取一种来实现控制功能。

初始表示
Initial Representation

定序控制
Sequencing Control

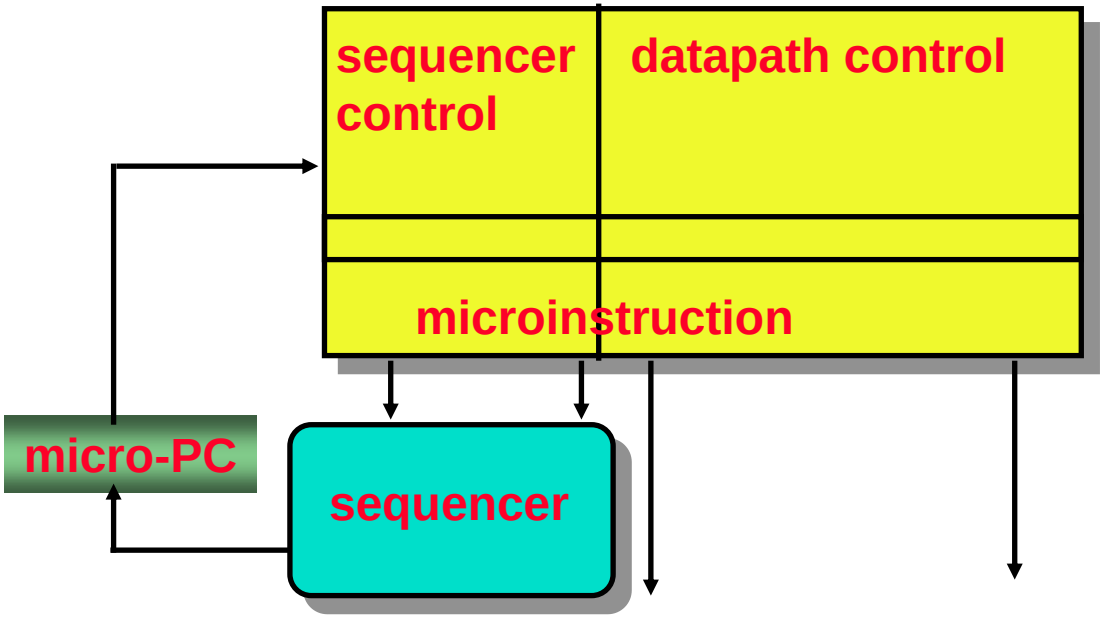
逻辑表示
Logic Representation

实现技术
Implementation Technique



# 控制器设计


- 用于定义具体指令系统控制器的状态图是高度结构化的
- 利用这一结构，可以构建一个简单的微序列发生器
- 控制就变成为：对这个非常简单的设备进行编程
  - 微程序编程

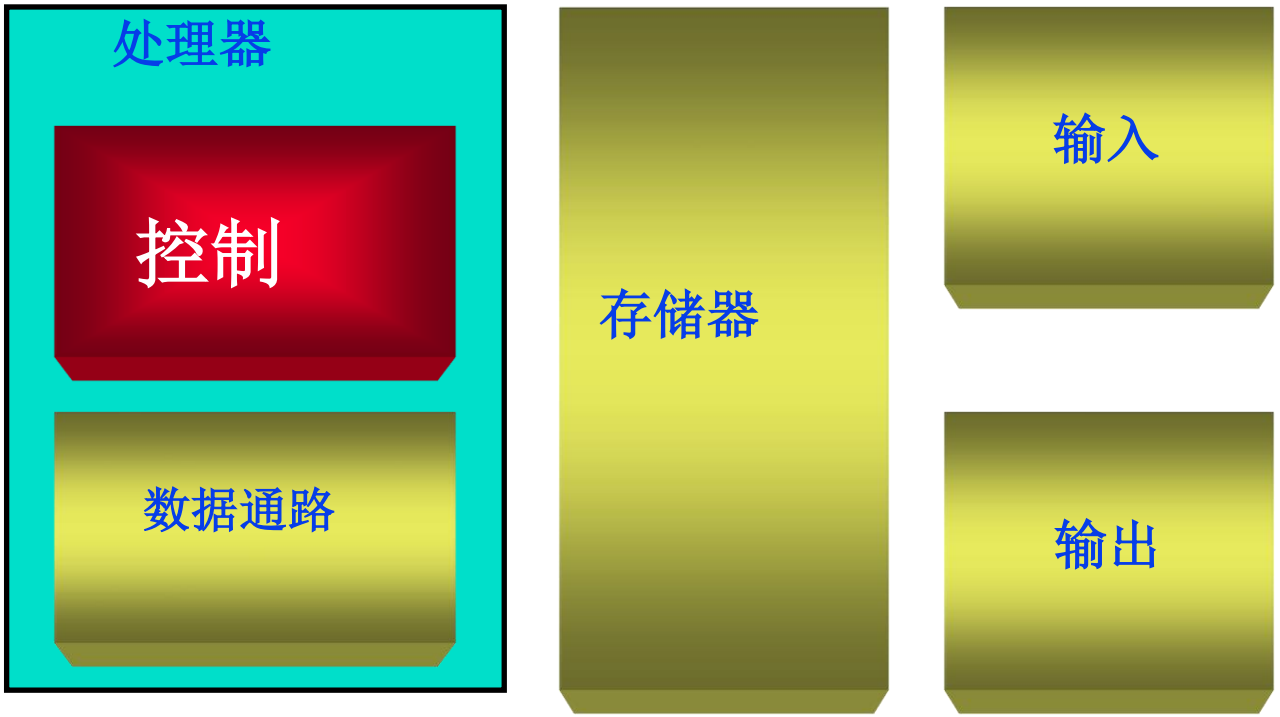


# 微程序控制描述


| | PC | Taken | Next | IR | PC en sel | Ops A B | Exec Ex Sr ALU S | Mem R W M | Write-Back M-R Wr Dst |
|---|---|---|---|---|---|---|---|---|---|
| | 0000 | ? | inc | 1 | | | | | |
| | 0001 | 0 | load | | | | | | |
| | 0001 | 1 | inc | | | | | | |
| BEQ | 0010 | x | zero | | 1 1 | | | | |
| | 0011 | x | zero | | 1 0 | | | | |
| R: | 0100 | x | inc | | | | 1 fun 1 | | |
| | 0101 | x | zero | | 1 0 | | | | 0 1 1 |
| ORi: | 0110 | x | inc | | | | 0 0 or 1 | | |
| | 0111 | x | zero | | 1 0 | | | | 0 1 0 |
| LW: | 1000 | x | inc | | | | 1 0 add 1 | | |
| | 1001 | x | inc | | | | | 1 0 1 | |
| | 1010 | x | zero | | 1 0 | | | | 1 1 0 |
| SW: | 1011 | x | inc | | | | 1 0 add 1 | | |
| | 1100 | x | zero | | 1 0 | | | 0 1 | |

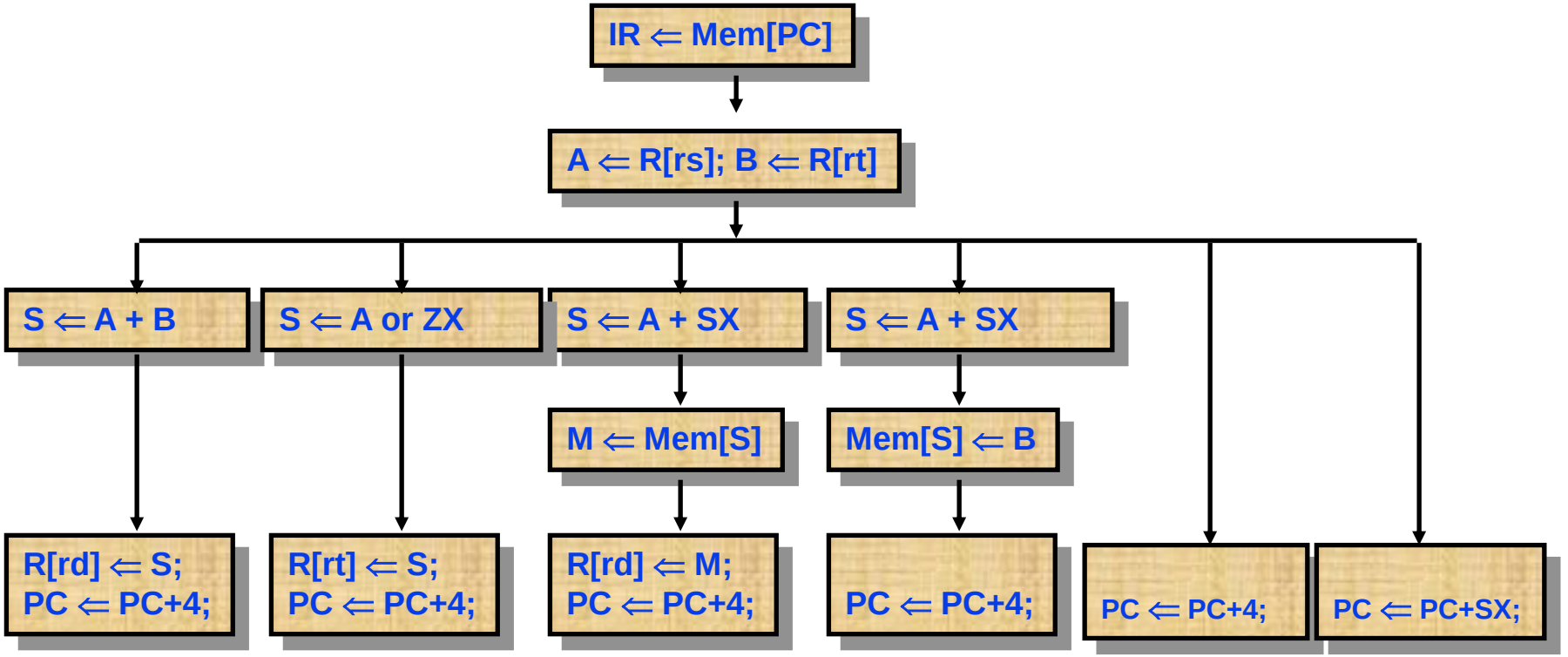
# 教学目标：已经掌握的内容

- 计算机的五个基本部件



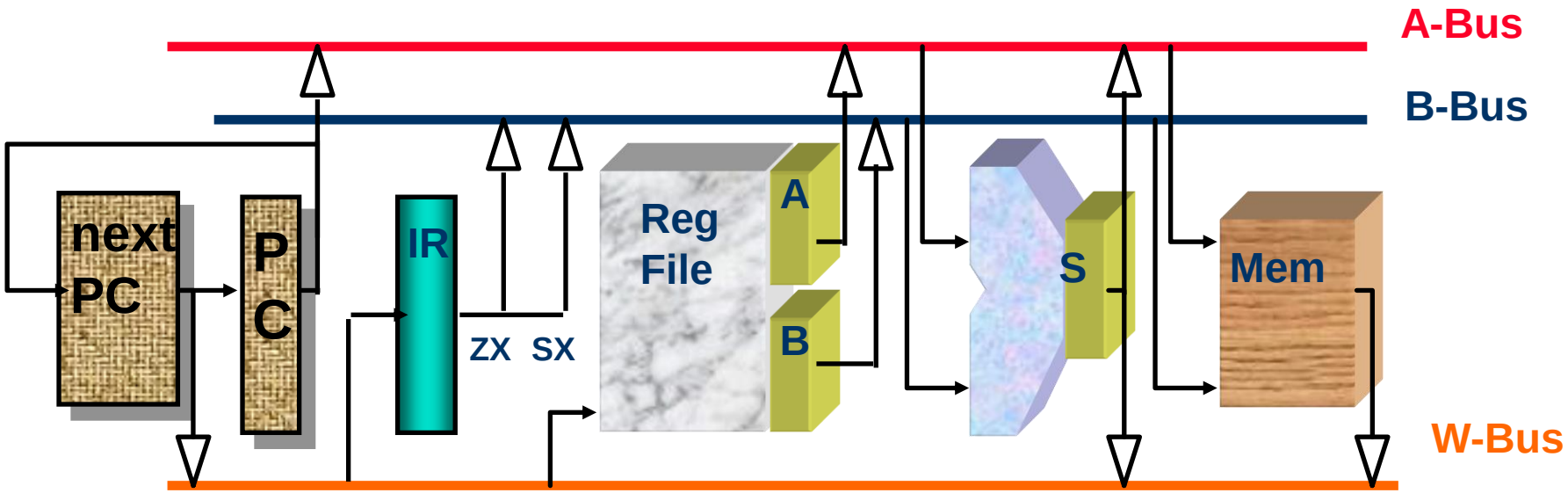

- 本讲主题：多周期数据通路设计

# 如何有效地利用我们的硬件?




- 例如: 存储器在不同的时间被使用了两次
  - 每条指令的平均存储器访问时间 = 1 + 装入指令的频度 + 存储指令的频度≈ 1.3
  - 如果CPI = 4.8, 指令存储器的利用率 = 1/4.8, 数据存储器的利用率 =0.3/4.8
- 在不降低性能的前提下,我们可以减少硬件
  - 额外的控制

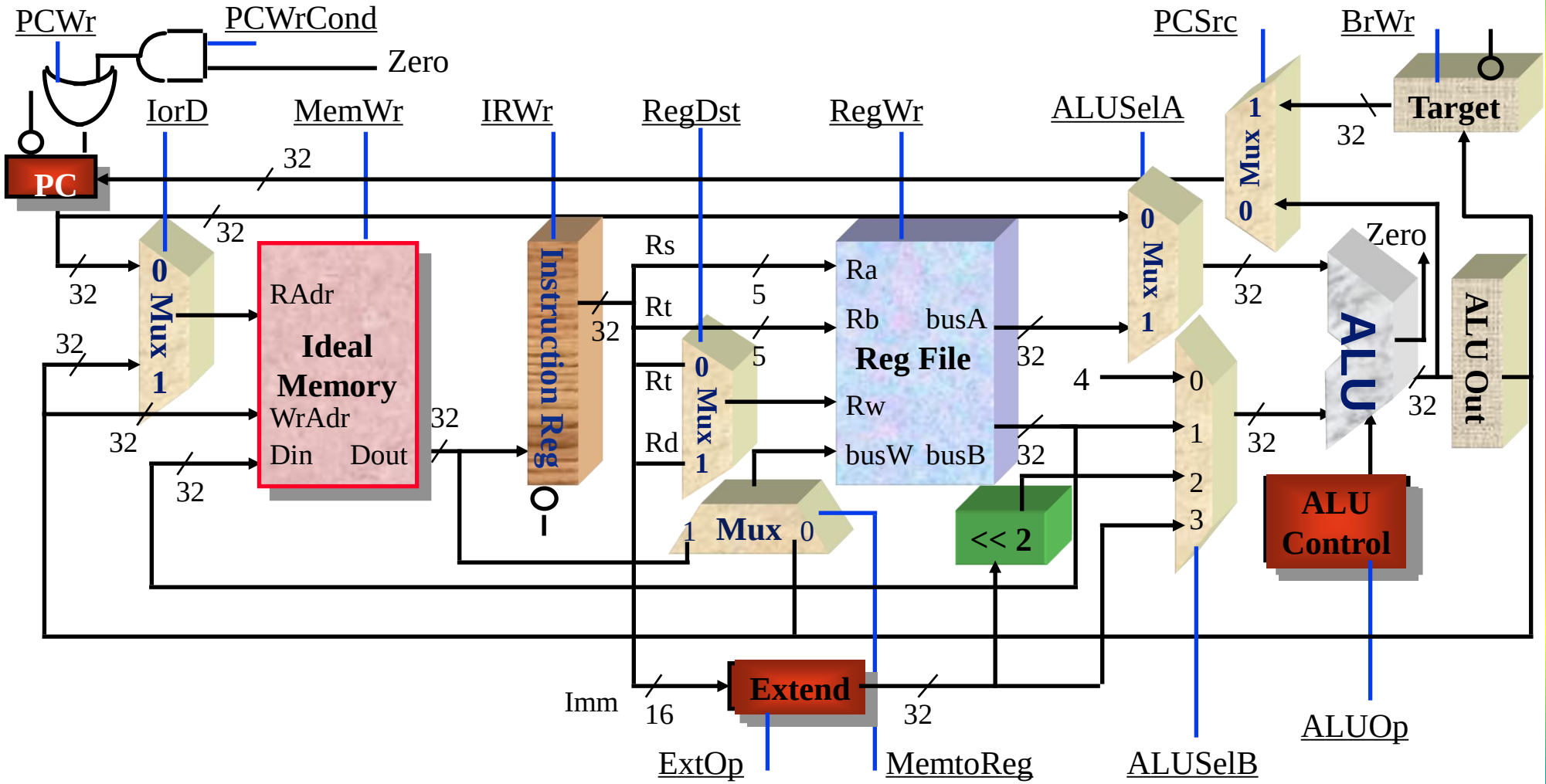
# Princeton组织



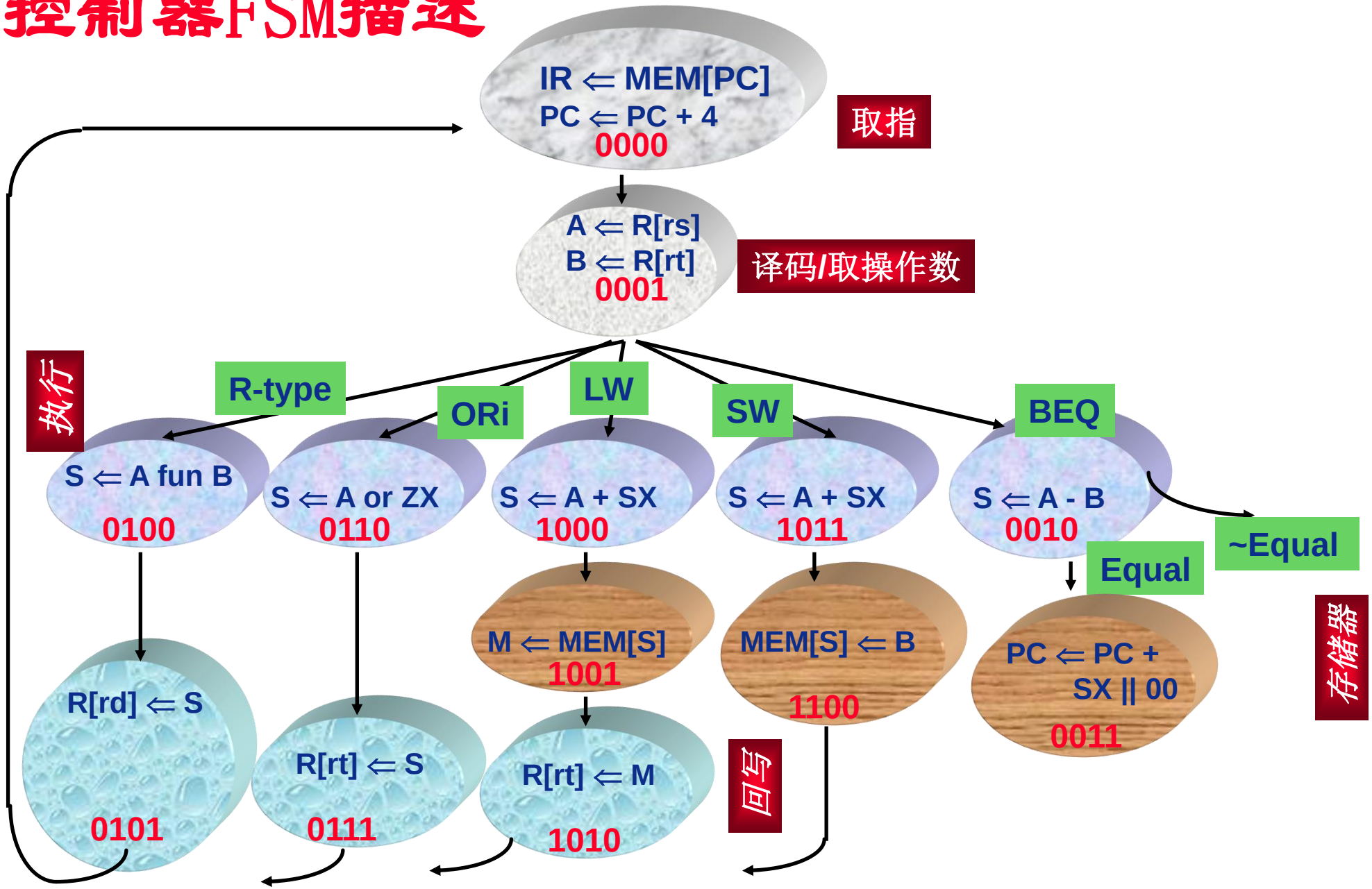

- 对指令和数据访问使用同一存储器
  - 存储器利用率 ⇒ 1.3/4.8
- 在这种情况下,状态图无需变化
  - 增加一些附加的控制信号
  - 必须保证在每一周期,只有唯一的一个源部件驱动总线

# 教科书中的数据通路：多周期数据通路


- 所需硬件最少：1个存储器，1个加法器



# 控制器FSM描述


IR ⇐ MEM[PC]
PC ⇐ PC + 4
0000
取指
A ⇐ R[rs]
B ⇐ R[rt]
0001
译码/取操作数
执行
R-type
ORi
LW
SW
BEQ
S ⇐ A fun B
0100
S ⇐ A or ZX
0110
S ⇐ A + SX
1000
S ⇐ A + SX
1011
S ⇐ A - B
0010
~Equal
Equal
M ⇐ MEM[S]
1001
MEM[S] ⇐ B
1100
PC ⇐ PC +
SX || 00
0011
存储器
R[rd] ⇐ S
0101
R[rt] ⇐ S
0111
R[rt] ⇐ M
1010
回写

# 控制概述

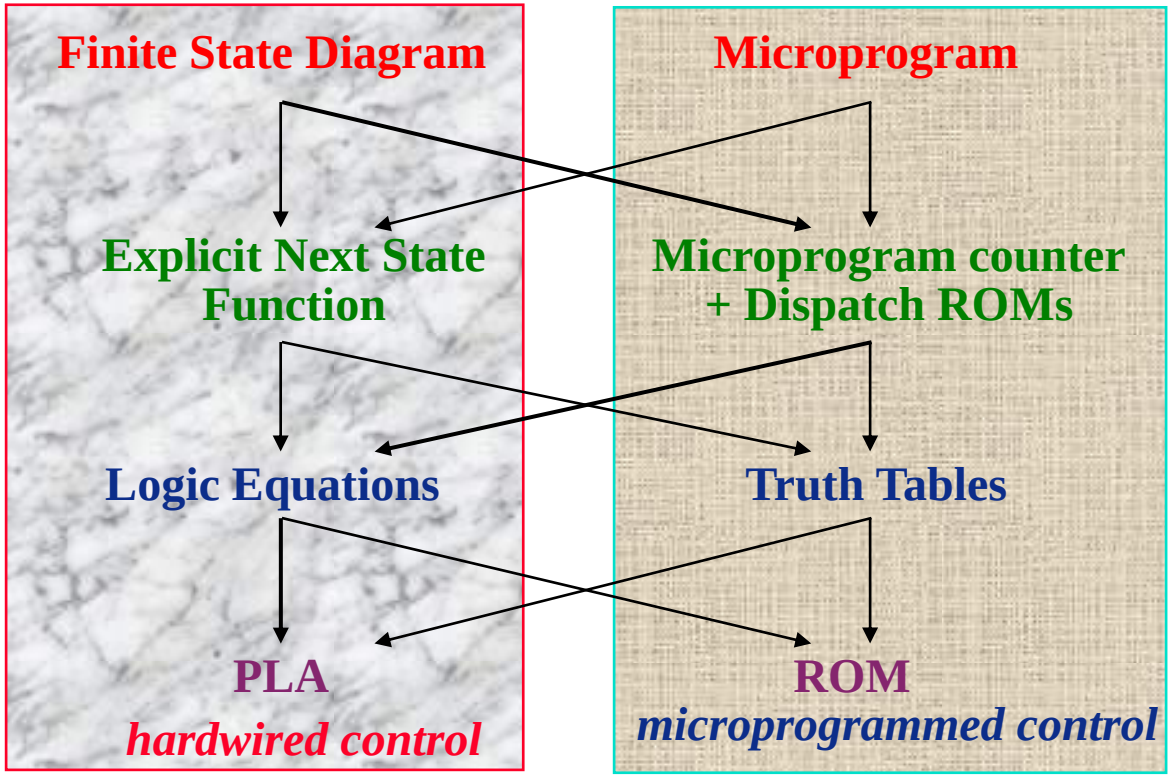
可以采用不同的初始表示来设计控制。然后，可以独立地择取序列控制，和如何表示逻辑功能；最后，采用结构化逻辑技术，从多种方法中，选取一种来实现控制功能。

初始表示
Initial Representation

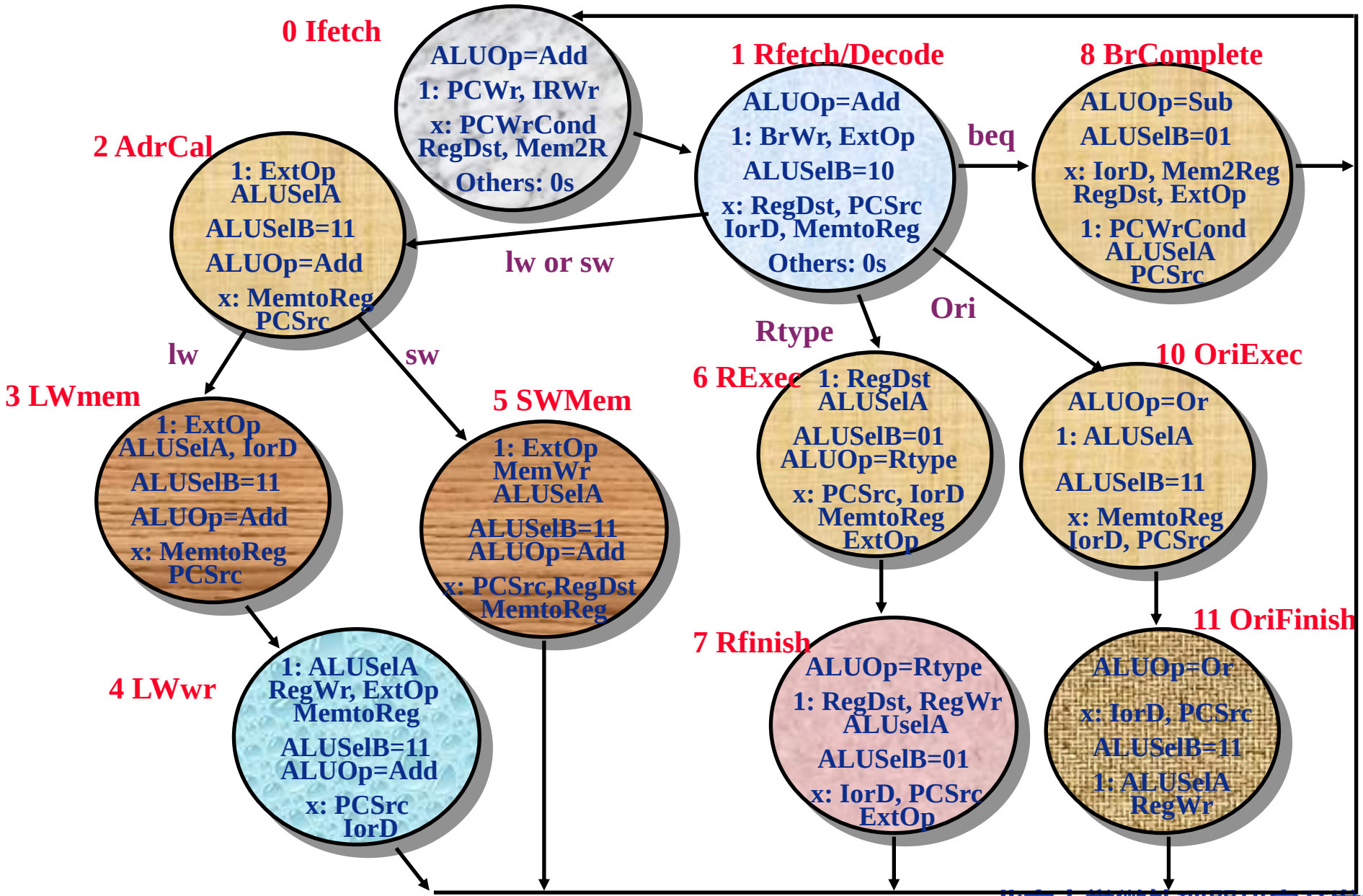
时序控制
Sequencing Control

逻辑表示
Logic Representation

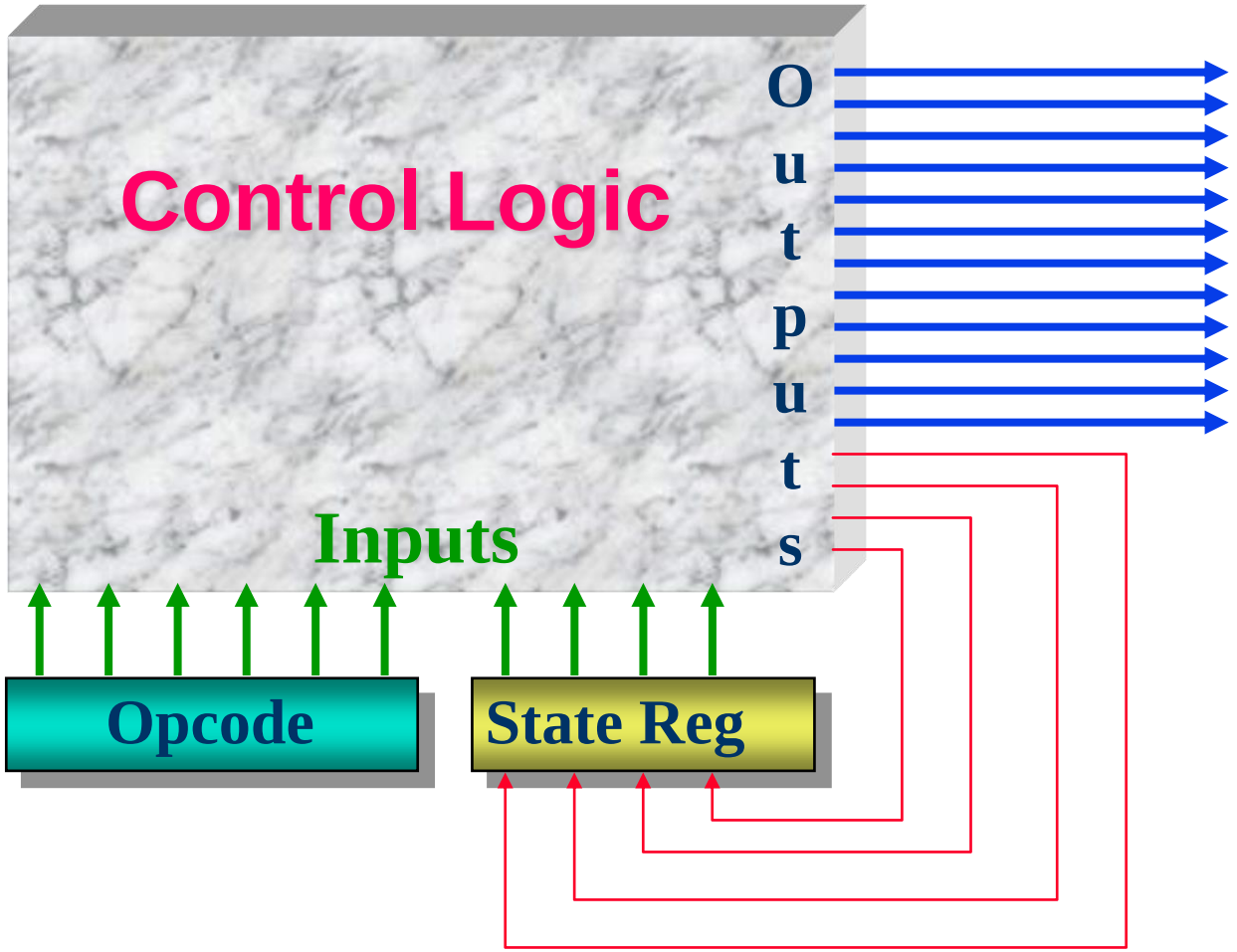
实现技术
Implementation Technique



# 初始表示：控制状态图


0 Ifetch
ALUOp=Add
1: PCWr, IRWr
x: PCWrCond
RegDst, Mem2R
Others: 0s
1 Rfetch/Decode
ALUOp=Add
1: BrWr, ExtOp
ALUSelB=10
x: RegDst, PCSrc
IorD, MemtoReg
Others: 0s
beq
8 BrComplete
ALUOp=Sub
ALUSelB=01
x: IorD, Mem2Reg
RegDst, ExtOp
1: PCWrCond
ALUSelA
PCSrc
2 AdrCal
1: ExtOp
ALUSelA
ALUSelB=11
ALUOp=Add
x: MemtoReg
PCSrc
lw or sw
Ori
Rtype
lw
sw
3 LWmem
1: ExtOp
ALUSelA, IorD
ALUSelB=11
ALUOp=Add
x: MemtoReg
PCSrc
5 SWMem
1: ExtOp
MemWr
ALUSelA
ALUSelB=11
ALUOp=Add
x: PCSrc,RegDst
MemtoReg
6 RExec
1: RegDst
ALUSelA
ALUSelB=01
ALUOp=Rtype
x: PCSrc, IorD
MemtoReg
ExtOp
10 OriExec
ALUOp=Or
1: ALUSelA
ALUSelB=11
x: MemtoReg
IorD, PCSrc
4 LWwr
1: ALUSelA
RegWr, ExtOp
MemtoReg
ALUSelB=11
ALUOp=Add
x: PCSrc
IorD
7 Rfinish
ALUOp=Rtype
1: RegDst, RegWr
ALUselA
ALUSelB=01
x: IorD, PCSrc
ExtOp
11 OriFinish
ALUOp=Or
x: IorD, PCSrc
ALUSelB=11
1: ALUSelA
RegWr

# 序列控制： 显式下一状态函数




- 下一状态号 的编码方式 与数据通路控制 类似

# 逻辑表示：逻辑方程式

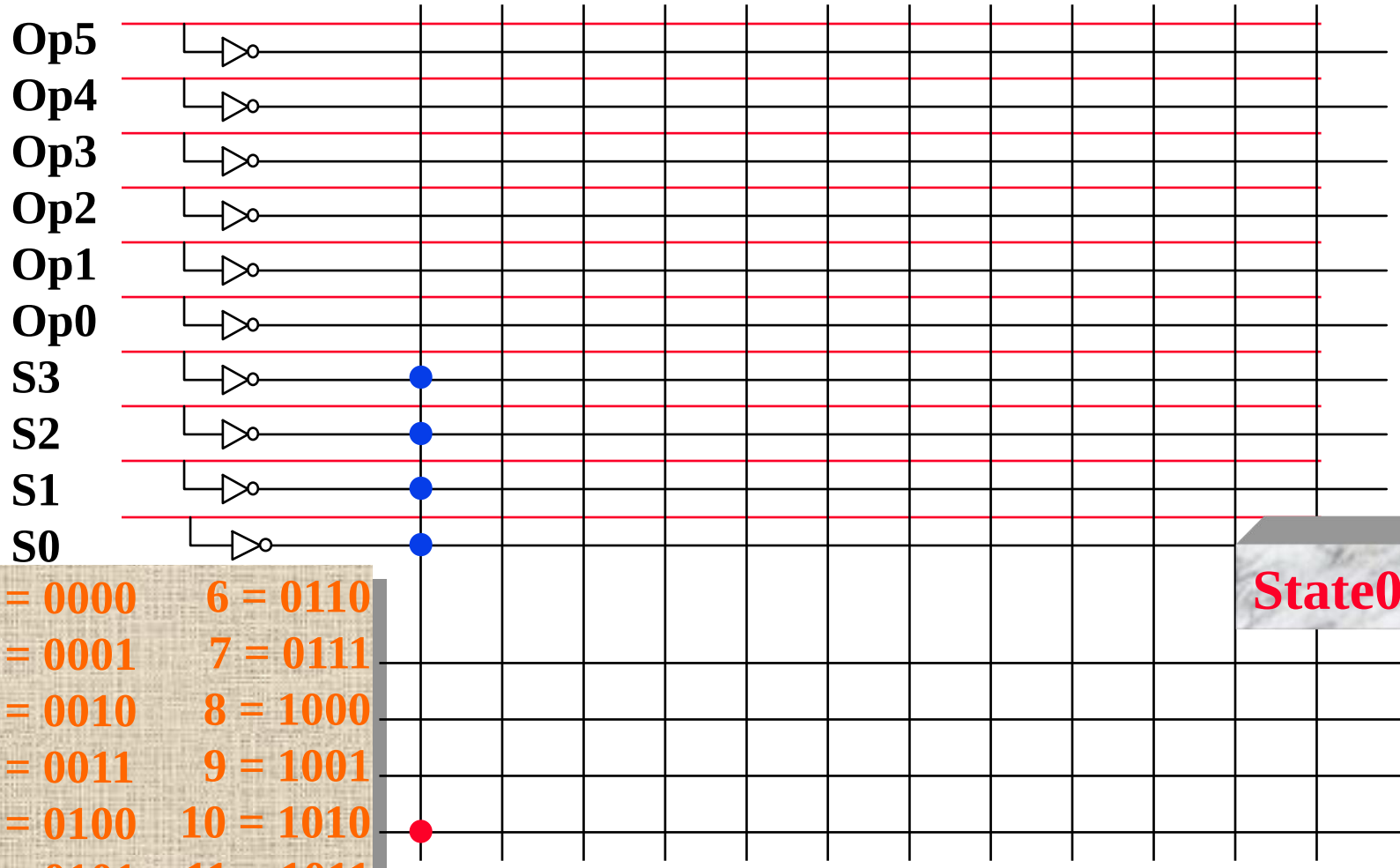
- 当前状态 ⇒下一状态
  - State 0 ⇒ State1
  - State 1 ⇒ S2, S6, S8, S10
  - State 2 ⇒ State3 &State5
  - State 3 ⇒ State4
  - State 4 ⇒ State 0
  - State 5 ⇒ State 0
  - State 6 ⇒ State 7
  - State 7 ⇒ State 0
  - State 8 ⇒ State 0
  - State 9 ⇒ State 0
  - State 10 ⇒ State 11
  - State 11 ⇒ State 0

- 对应情况, 前一状态&条件

| | | |
|---|---|---|
| S4, S5, S7, S8, S9, S11 | ⇒ | State0 |
| S0 | ⇒ | State 1 |
| S1&OP=LW\|OP=SW | ⇒ | State 2 |
| S2&OP=LW | ⇒ | State 3 |
| S3 | ⇒ | State 4 |
| State2 & op = sw | ⇒ | State 5 |
| S2&OP=Rtype | ⇒ | State 6 |
| State 6 | ⇒ | State 7 |
| S2&op=beq | ⇒ | State 8 |
| State2 & op = jmp | ⇒ | State 9 |
| S2&op=ori | ⇒ | State 10 |
| State 10 | ⇒ | State 11 |

# 实现技术：PLA(Programmed Logic Arrays)

- 每个输出线：输入信号或其反向信号 的 逻辑与 的 逻辑或: AND minterms在上部AND模板（plane）指定，OR sums 在底部 OR模板（plane）指定



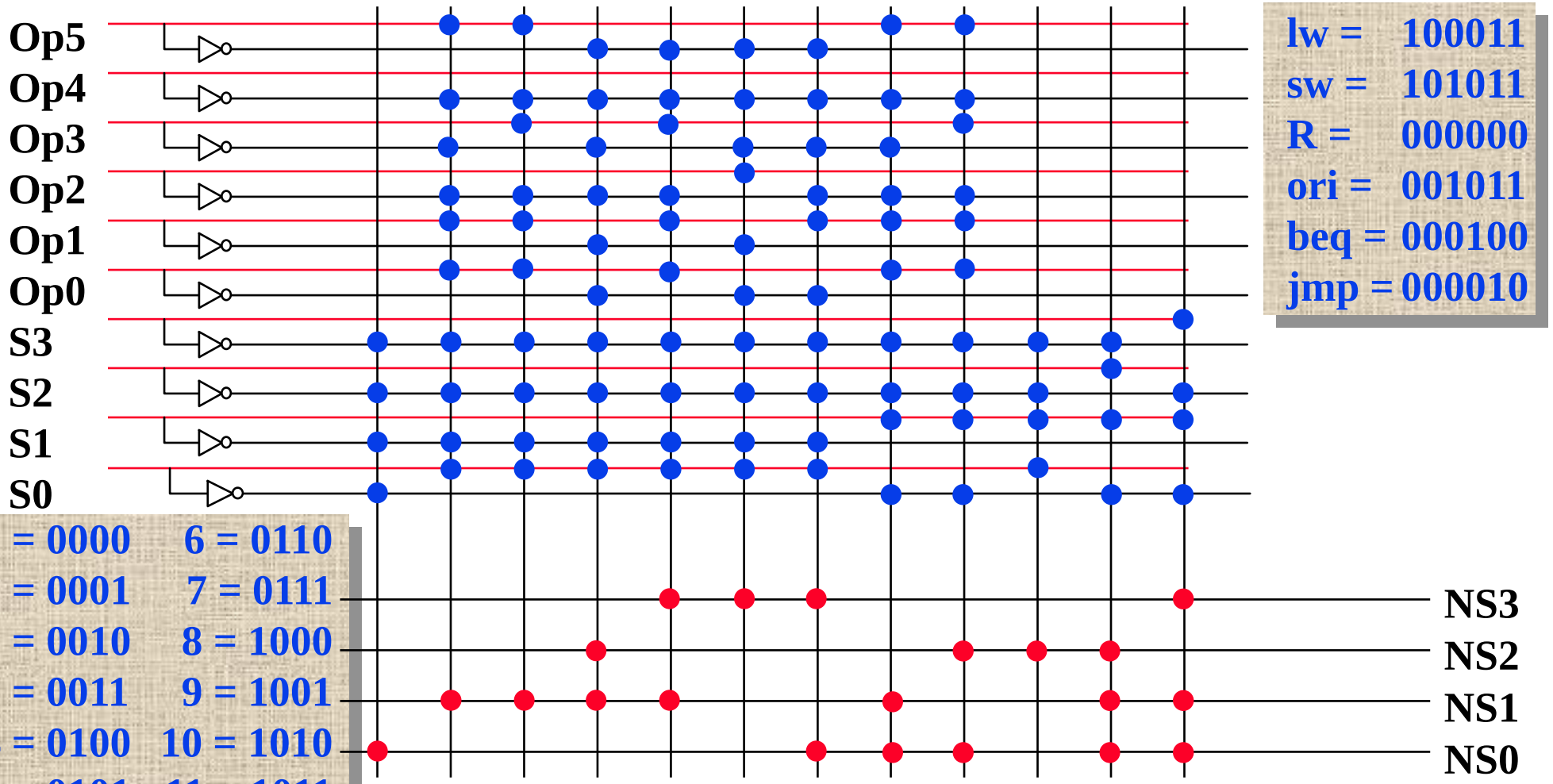

R = 000000
beq = 000100
lw = 100011
sw = 101011
ori = 001011
jmp = 000010

State0⇒State1

0 = 0000 6 = 0110
1 = 0001 7 = 0111
2 = 0010 8 = 1000
3 = 0011 9 = 1001
4 = 0100 10 = 1010
5 = 0101 11 = 1011

# 实现技术：PLA(Programmed Logic Arrays)

- 每个输出线：输入信号或其反向信号 的 逻辑与 的 逻辑或: AND minterms在上部 AND模板（plane）指定，OR sums 在底部 OR模板（plane）指定



# 多周期控制

- 给定 FSM 的状态号，就可以根据下一状态和 输入、当前状态的函数关系，确定下一状态
- 将这些函数变换为 关于下一状态控制线的每一位的布尔表达式
- 可以用PLA来轻易实现
- 如果具有很多状态、很多条件，那么情况如何？
- 如果需要增加一个状态，那么情况又会如何？

# 使用时序部件（Sequencer）产生下一状态

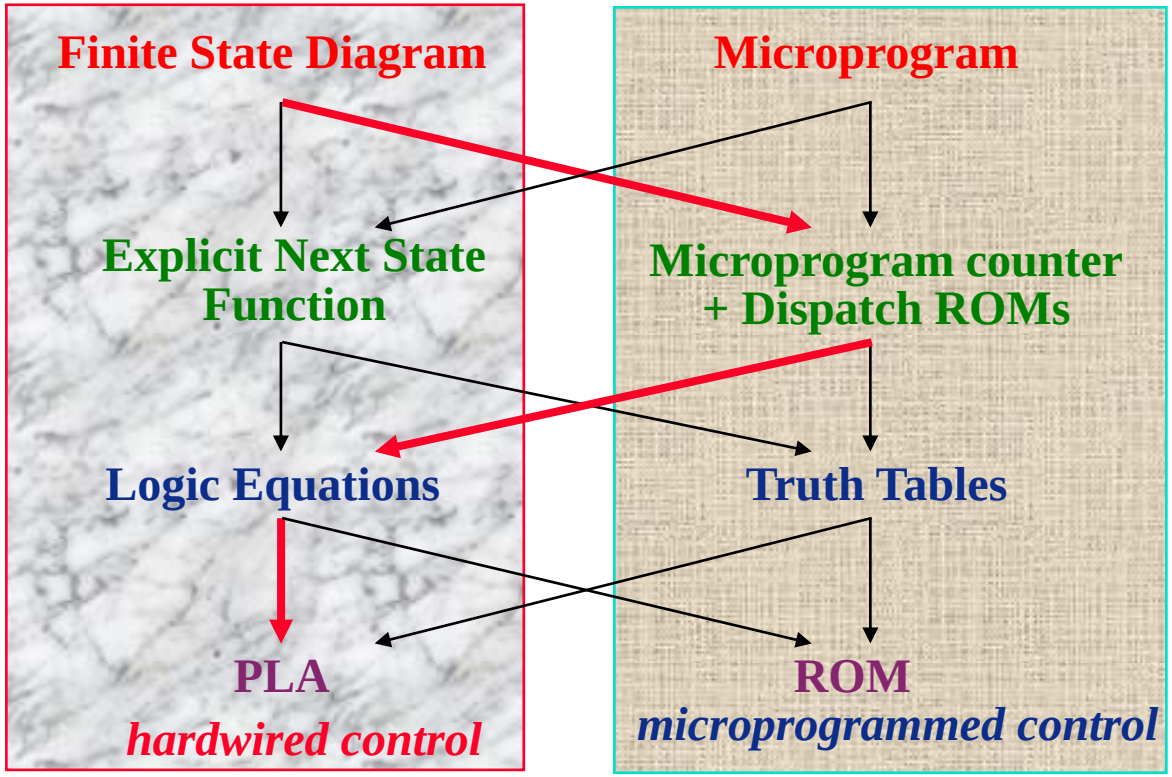
- 在“显式下一状态”之前: 试图从右边的框图中前进一步
- 在小规模FSM中有一些串行状态: 假设进行浮点加法
- 也需要进入一些非串行状态: 例如，state 1 => 2, 6, 8, 10

初始表示
Initial Representation

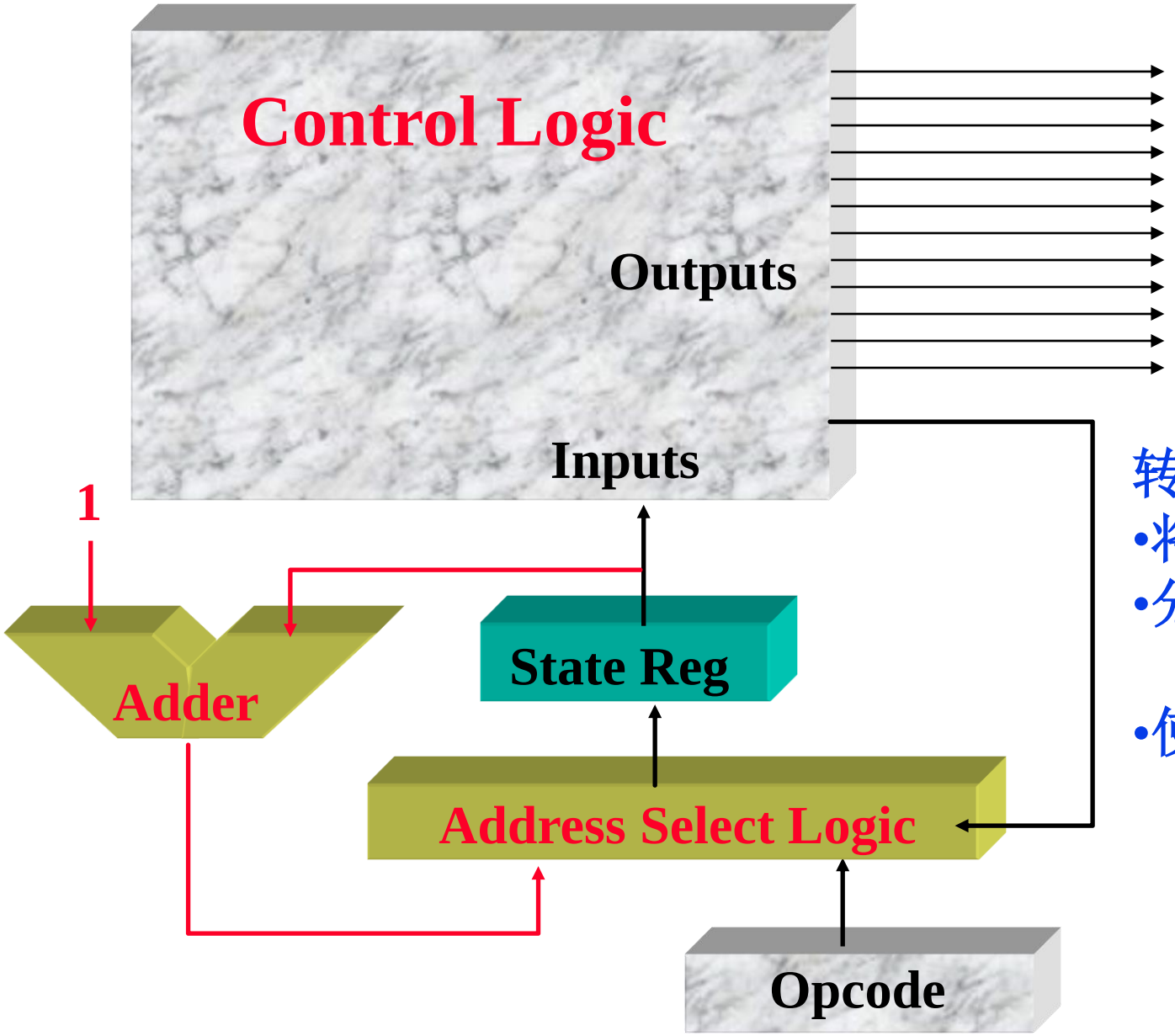
时序控制
Sequencing Control

逻辑表示
Logic Representation

实现技术
Implementation Technique



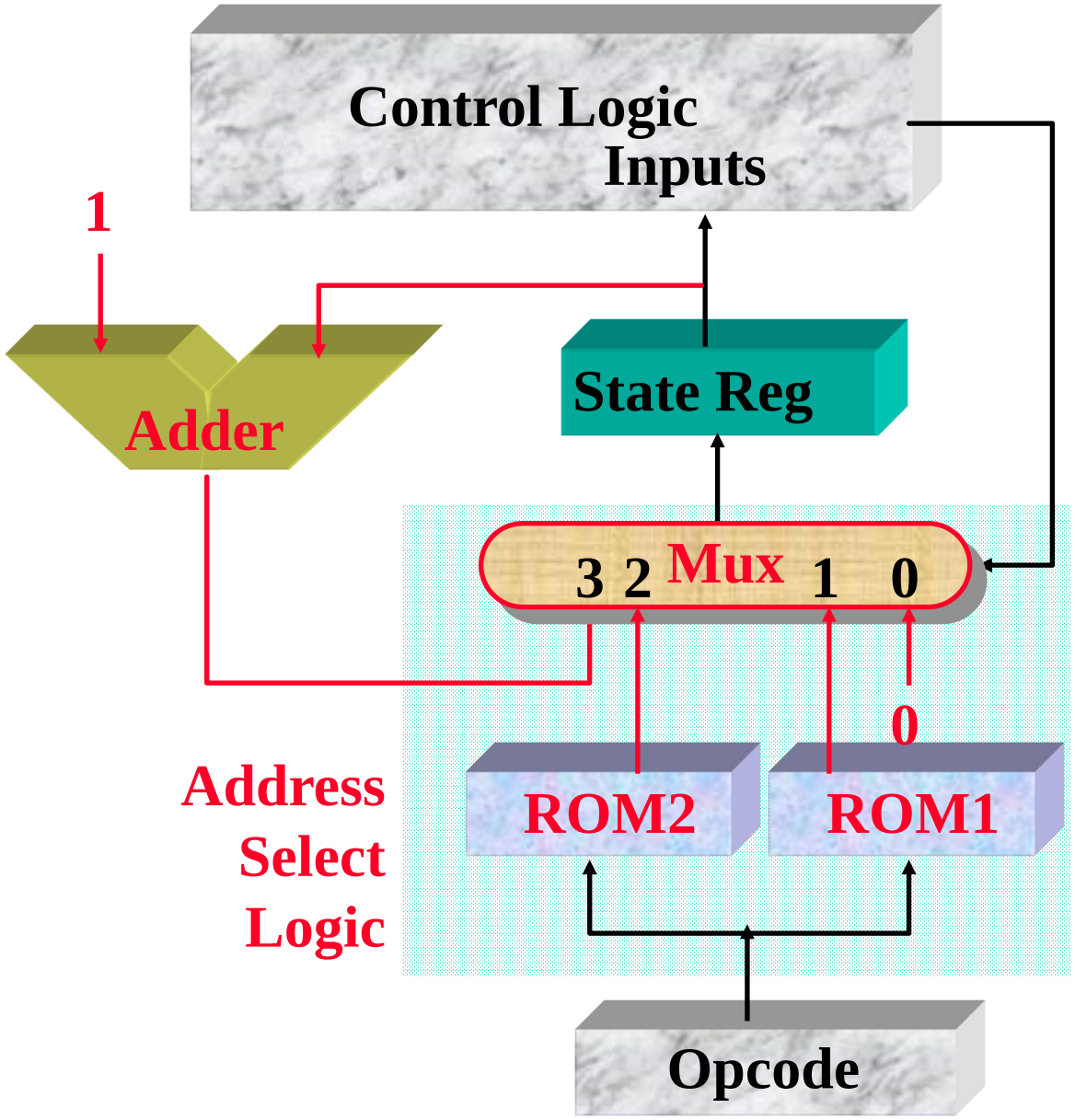
# 基于时序部件的控制单元




多周期
数据通路

转移的类型
- 将状态置为 0
- 分发（Dispatch）：
  状态1 & 2
- 使用递增的状态号

# 基于时序部件的控制单元探秘




*Dispatch ROM 1*

| *Op* | *Name* | *State* |
|---|---|---|
| 000000 | Rtype | 0110 |
| 000010 | jmp | 1001 |
| 000100 | beq | 1000 |
| 001011 | ori | 1010 |
| 100011 | lw | 0010 |
| 101011 | sw | 0010 |

*Dispatch ROM 2*

| *Op* | *Name* | *State* |
|---|---|---|
| 100011 | lw | 0011 |
| 101011 | sw | 0101 |

# 使用ROM实现控制

- 用ROM代替PLA, 每个状态一个控制字

| *状态号*<br>*State number* | *控制字位 18-2*<br>*Control Word Bits 18-2* | *控制字位 1-0*<br>*Control Word Bits 1-0* |
|---|---|---|
| 0 | 10010100000001000 | 11 |
| 1 | 00000000010011000 | 01 |
| 2 | 00000000000010100 | 10 |
| 3 | 00110000000010100 | 11 |
| 4 | 00110010000010110 | 00 |
| 5 | 00101000000010100 | 00 |
| 6 | 00000000001000100 | 11 |
| 7 | 00000000001000111 | 00 |
| 8 | 01000000100100100 | 00 |
| 9 | 10000001000000000 | 00 |
| 10 | ? | 11 |
| 11 | ? | 00 |

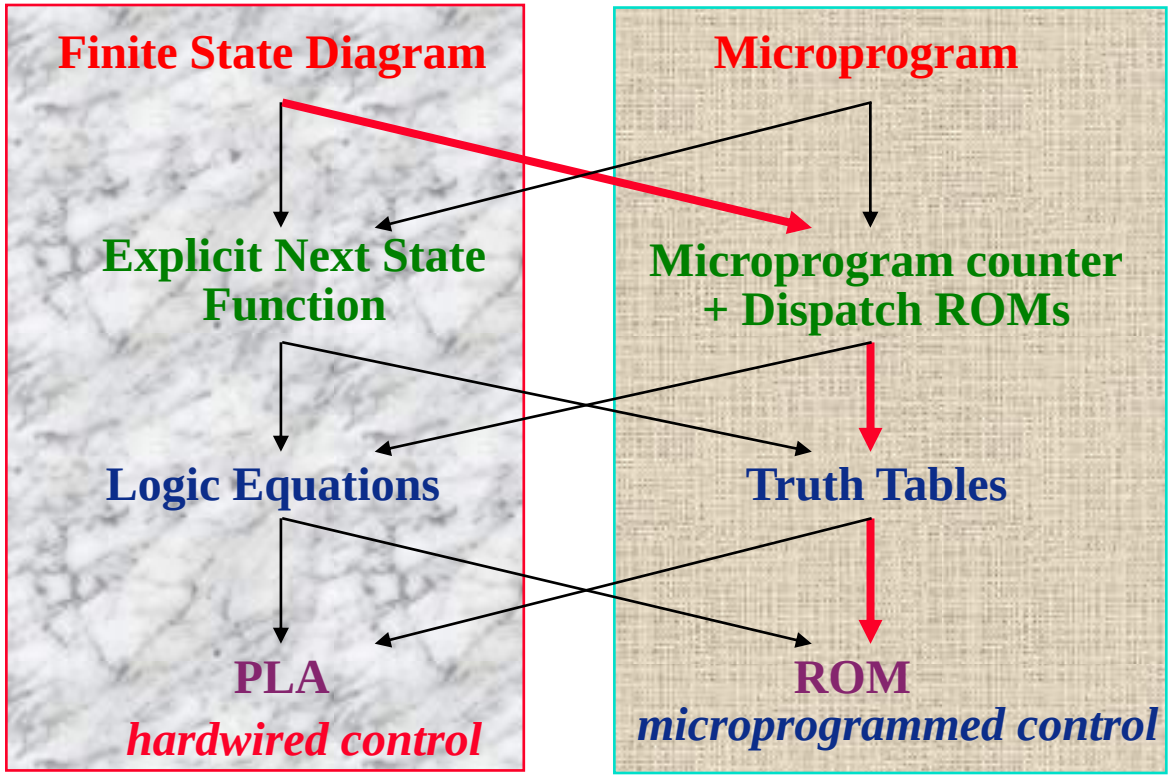
# 使用微程序表示

初始表示
Initial Representation

时序控制
Sequencing Control

逻辑表示
Logic Representation

实现技术
Implementation Technique




- ROM可以被看为是一个控制字的序列
- 控制字可以被看为是一条指令: 微指令（Microinstruction）
- 使用汇编语言，而不是二进制程序

# 微程序设计

- 控制是处理器设计的一个难点
  - 数据通路具有较好的规整性和很好的组织
  - 存储器具有很高的规整性
  - 控制不规整，并且涉及全局

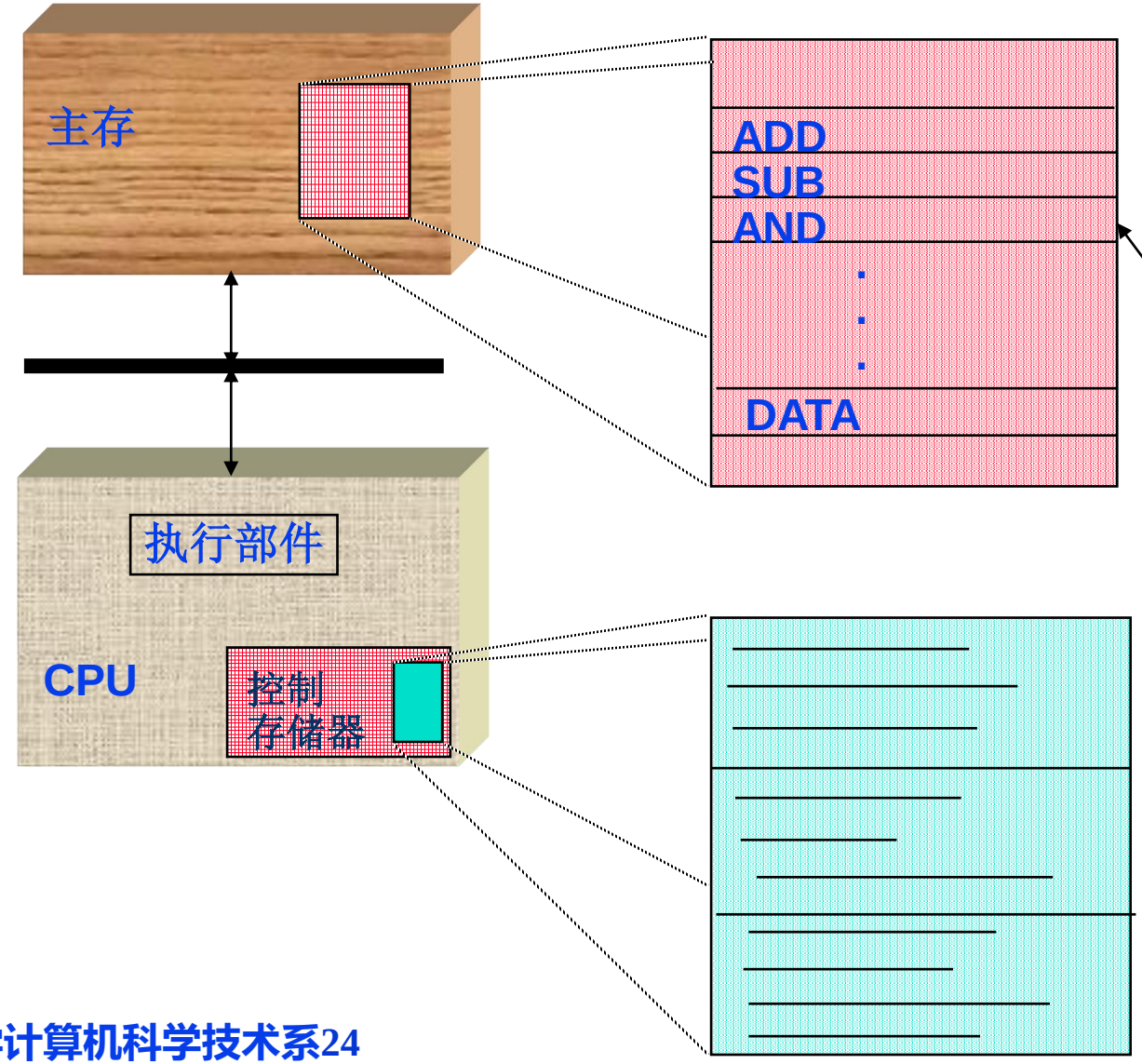
**微程序设计:**
一种特殊的实现处理器控制部件的策略，它在寄存器传输操作的级别进行“编程”

**微体系结构（Microarchitecture）:**
微程序编程人员所看到的硬件的逻辑结构和功能特性

*注释:*

- IBM 360系列首次把体系结构和组成的概念区别开来
- 同一指令系统在很大范围内采用不同的实现技术来实现，使其具有不同的性能价格比

# 宏指令解释执行




用户程序+数据

这一部分可以变化!

这两部分可以相互映射

AND 微序列

例如：
取指
计算操作数地址
取操作数
计算
存储结果

# 微程序设计的不同类型

水平微码

针对机器中每个控制点的控制场位

| Req | Addr | A-mux | B-mux | bus enables | register enables | |
|---|---|---|---|---|---|---|

垂直微码

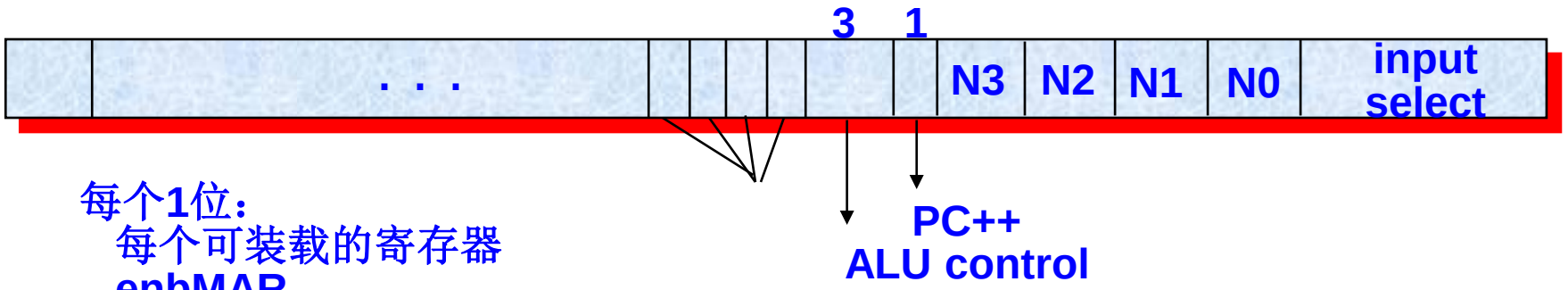
针对每一类微操作，压缩微指令格式

**branch:** **Req-op Add**

**execute:** **ALU-op A,B,R**

**memory:** **mem-op S, D**

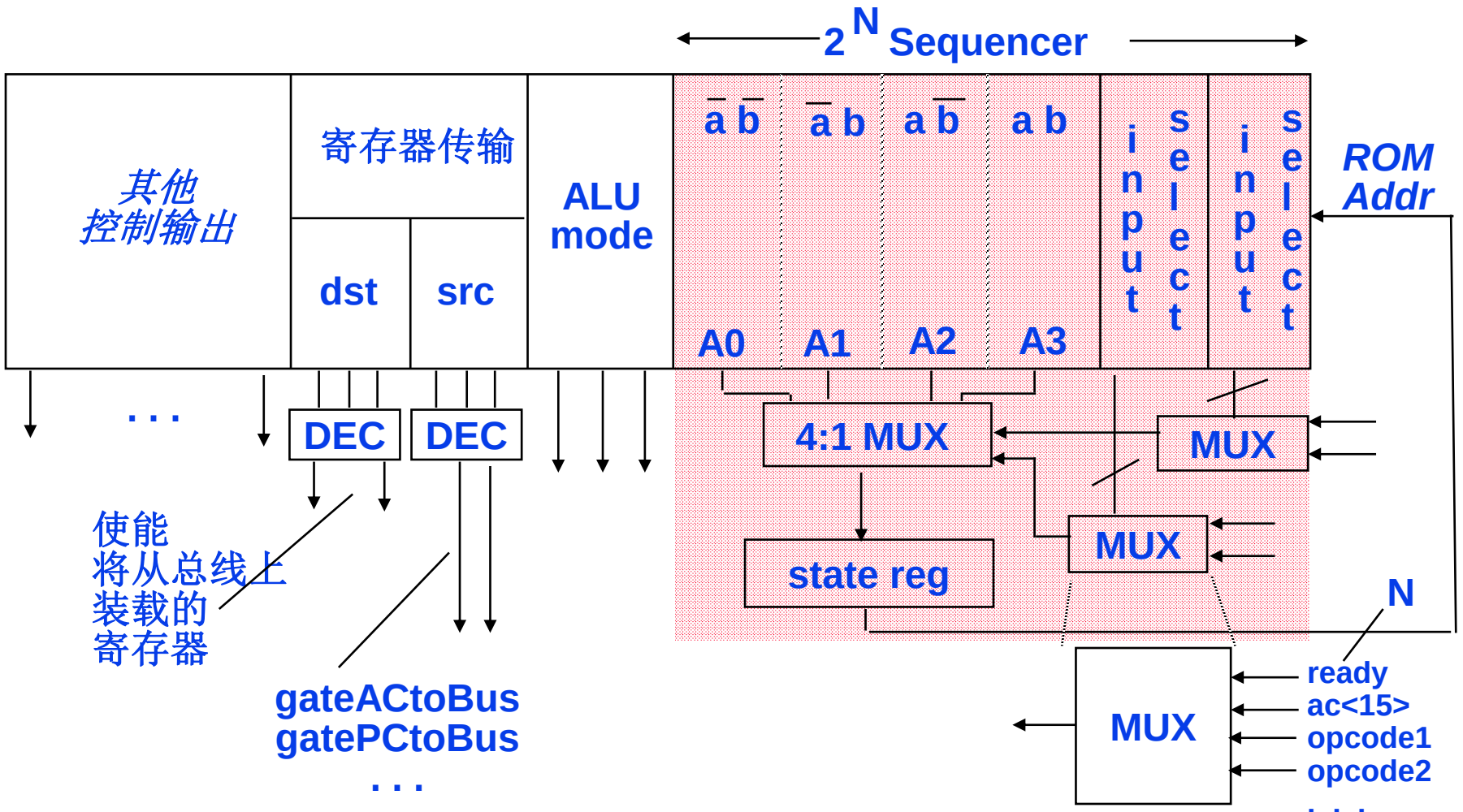

# 完全水平编码 (Extreme Horizontal)



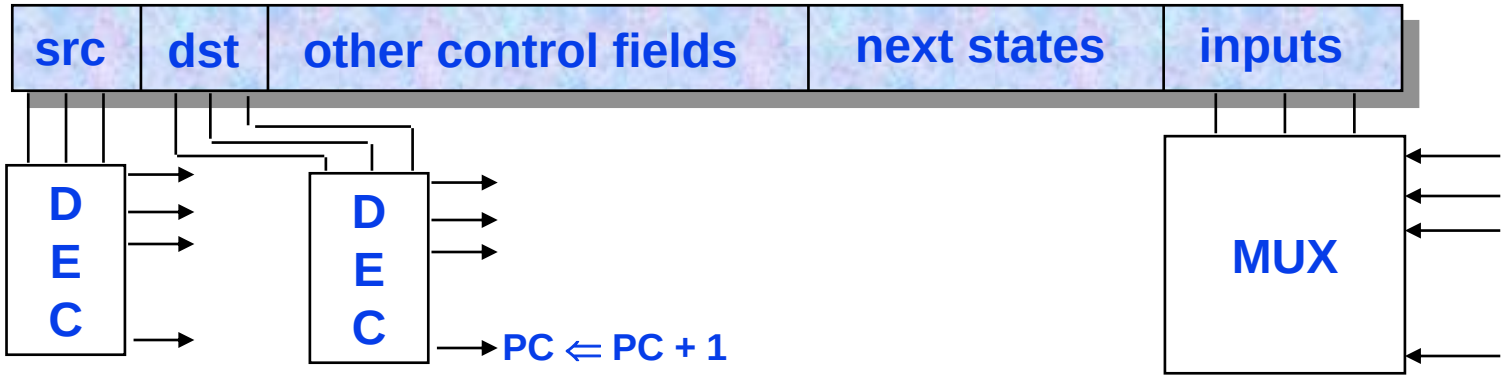

- 依赖于总线的组成，许多可能的控制组合在同一个时刻并不会发生
- 启示我们对场位进行编码，以节省ROM空间
- 例如：门 Rx 和门 Ry 不可能同时连接到同一总线上⇒ 编码在同一位
- 注释：编码应该刚好足以满足在单一微指令中可指明的、数据通路支持的并行动作的需求

# 微程序编程示例

- 使用 ROM/RAM,而不采用离散逻辑,来产生控制点
- 包括下一状态逻辑（微序列器）的控制

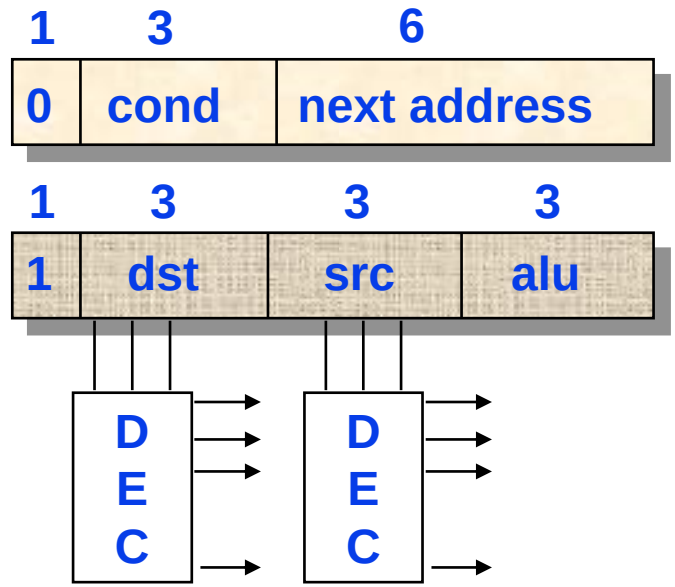


# 更加垂直的格式




*这些译码输出中的一些信号可能对寄存器不做任何操作！*

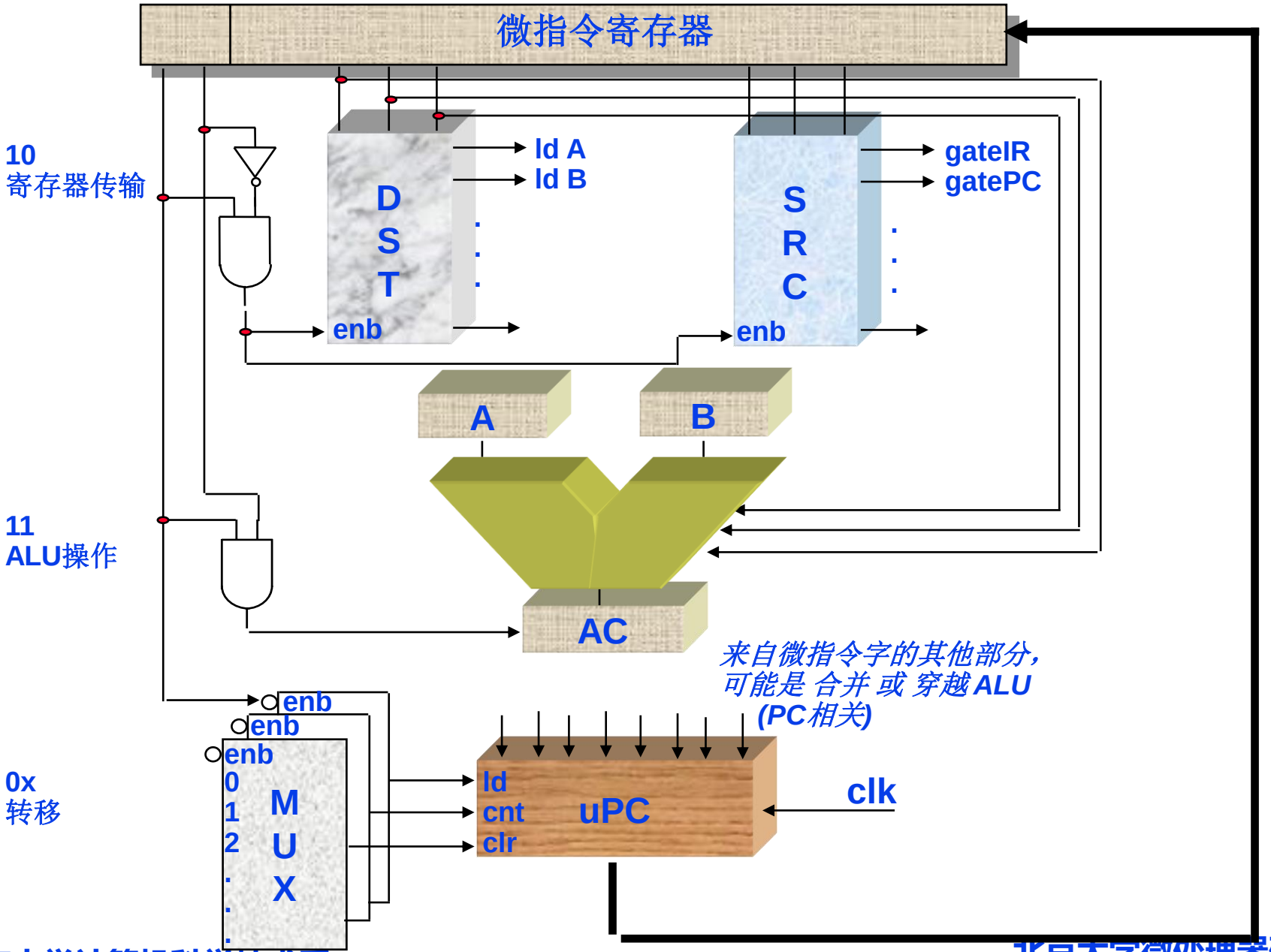
*多格式微码:*




转移、跳转

寄存器传输操作

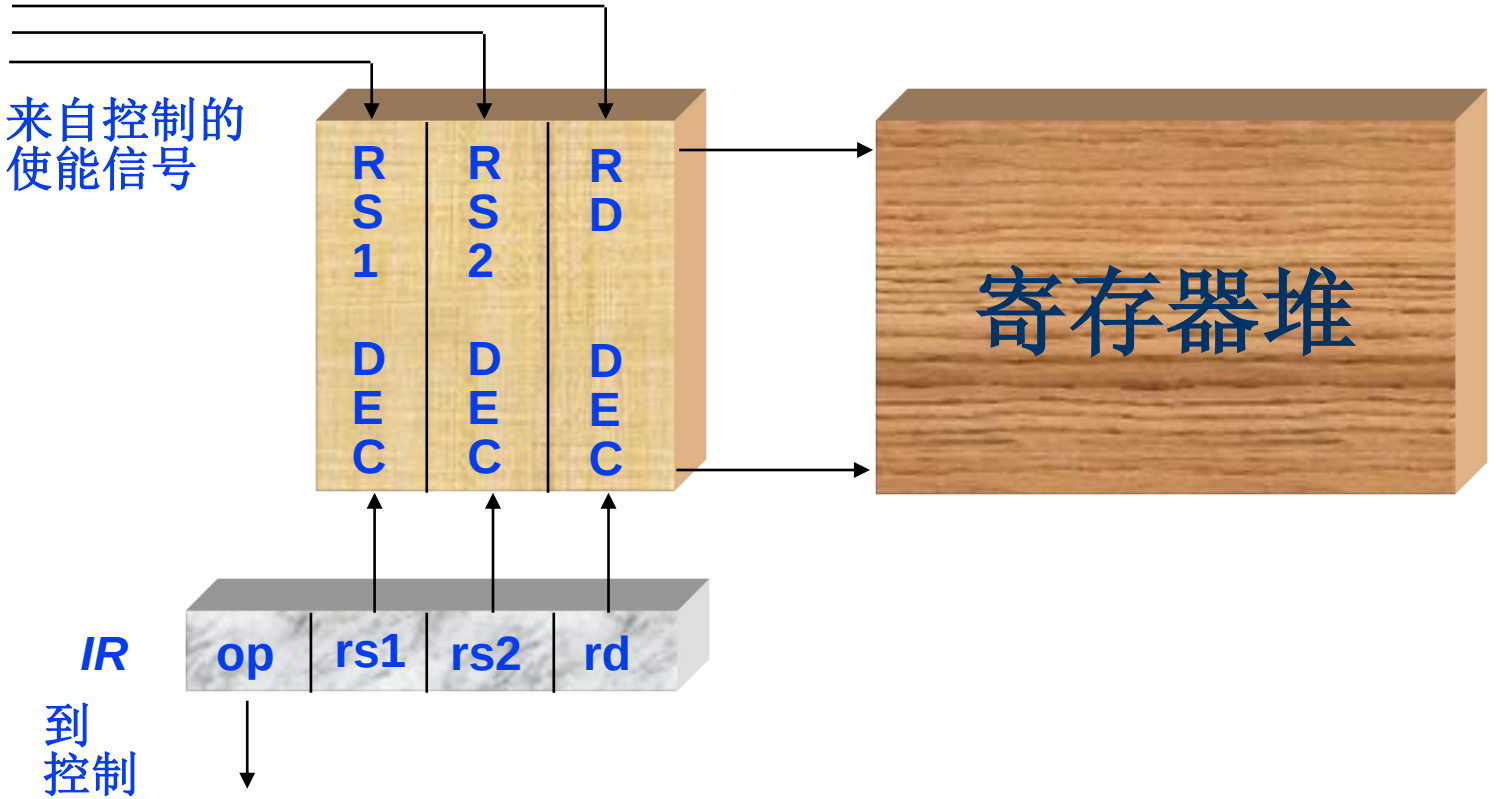
# 控制器实现


微指令寄存器
10
寄存器传输
D
S
T
enb
ld A
ld B
S
R
C
enb
gateIR
gatePC
A
B
11
ALU操作
AC
来自微指令字的其他部分，
可能是 合并 或 穿越 ALU
(PC相关)
0x
转移
enb
enb
enb
0
1
2
M
U
X
ld
cnt
clr
uPC
clk

# 混合控制

并非所有的关键控制信息都来自控制逻辑

例如， IR中也包括有用的控制信息, 如原寄存器、目标寄存器、操作码等等



# 水平微程序设计 与 垂直微程序设计

注释：上述组织 并非 是真正的水平微程序设计；
寄存器译码器 支持 编码后的 微操作

大多数基于微程序设计的控制器之间的差异表现为：

水平组织 (每个控制点一个控制位)

垂直组织 (在控制存储器存放编码后的场位，在发出控制信号之前必须首先译码)

*水平编码*

+ 对数据通路中操作的潜在并行性能够进行更好的控制

- 需要使用更多的控制存储

*垂直编码*

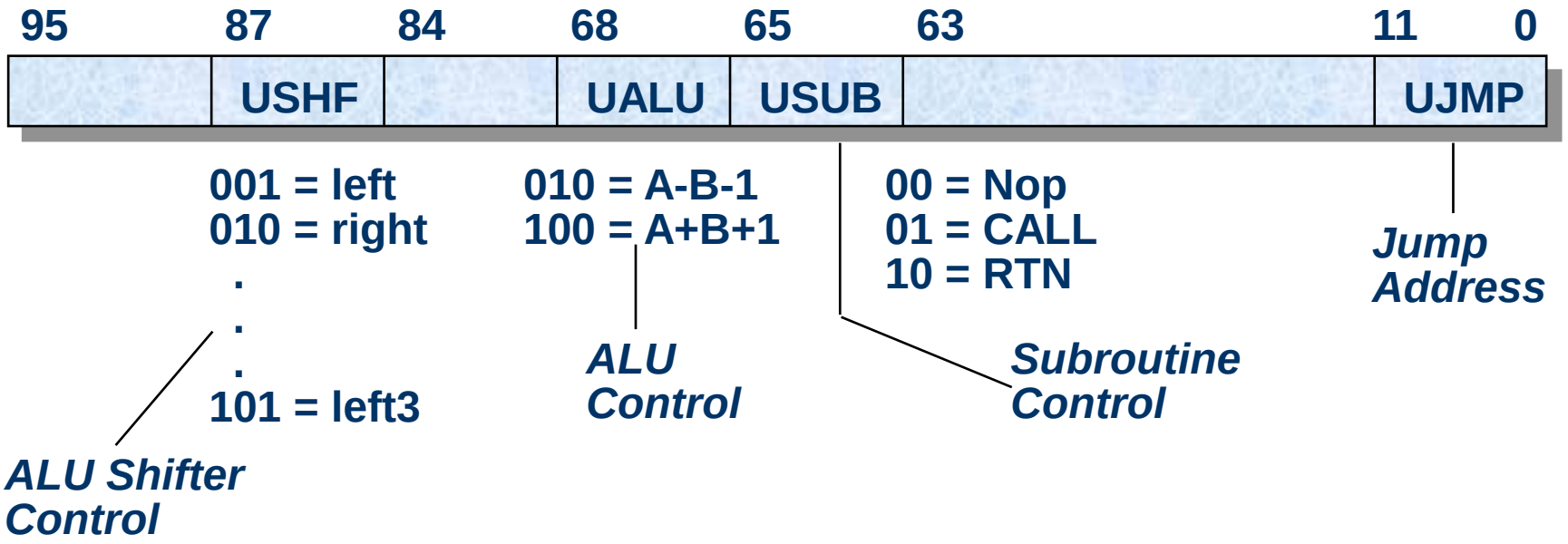
+ 易于编程，与使用汇编语言对一个RISC机器进行编程没有太大差别

- 额外级别的译码可能会降低机器的速度

# Vax 微指令

VAX 微体系结构:

96位的控制存储, 30个场位, VAX ISA的4096条微指令编码了可并发执行的 “微操作"



# 设计一个微指令系统

1) 从一系列控制信号开始设计

2) 根据这些信号的性质将它们分组: 称为“场位”

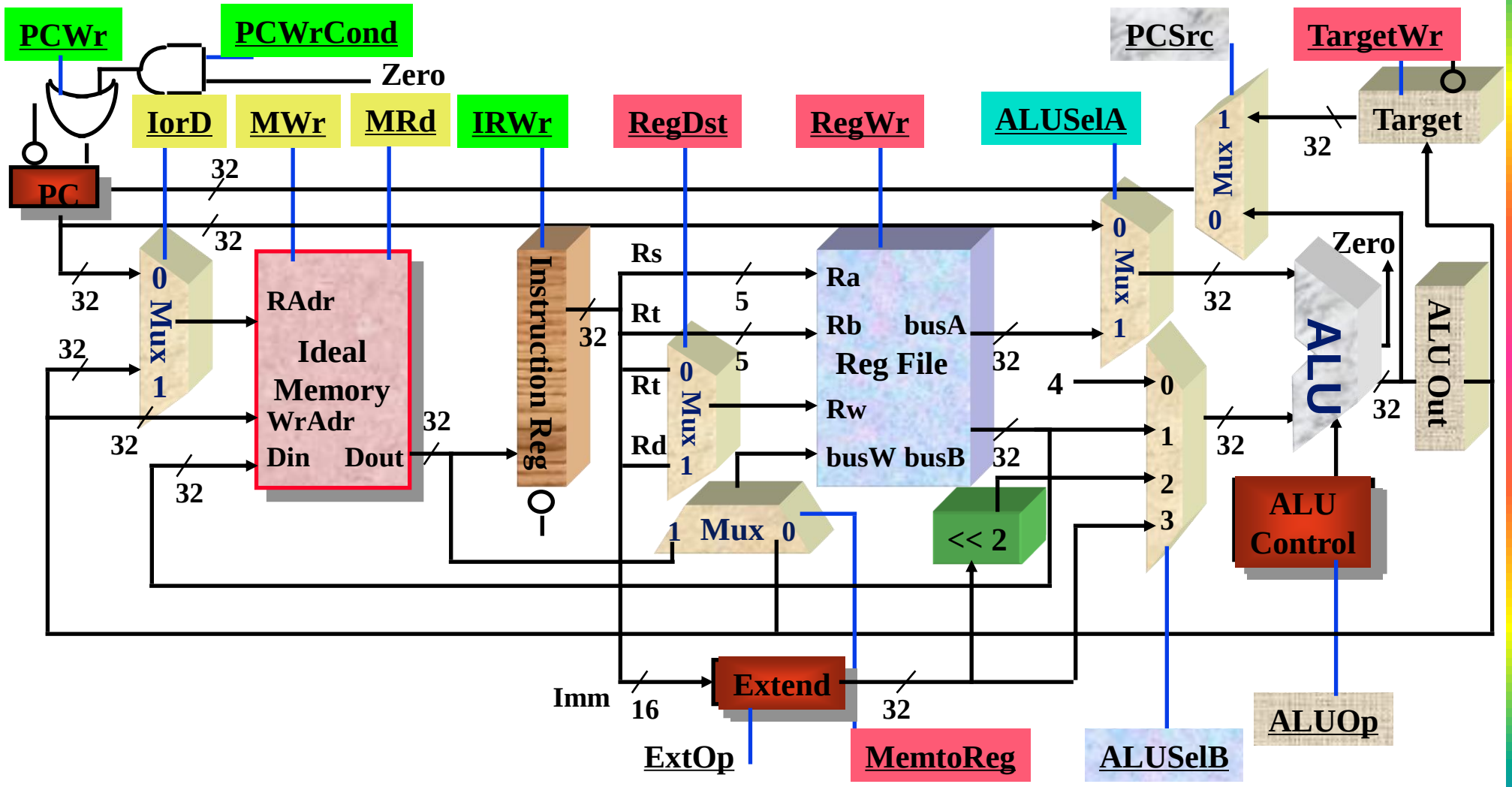
3) 按某种逻辑次序,安放这些场位(例如, 首先编排ALU操作 & ALU操作数, 再排放下一条微指令的地址)

4) 制定微指令格式的符号描述, 说明场位数值的名称,以及它们是如何设定控制信号的

- 使用计算机来设计计算机

5) 为减少宽度, 对那些不可能同时使用的操作进行编码

# 多周期数据通路


PCWr
PCWrCond
Zero
IorD
MWr
MRd
IRWr
RegDst
RegWr
ALUSelA
PCSrc
TargetWr
Target
PC
32
0 Mux 1
RAdr
Ideal Memory
WrAdr
Din Dout
Instruction Reg
Rs
Rt
Rt
Rd
5
Ra
Rb busA
Reg File
Rw
busW busB
4
0 1 2 3
<< 2
Zero
ALU
ALU Out
ALU Control
Imm
16
Extend
ExtOp
MemtoReg
ALUSelB
ALUOp

# 1&2）从控制信号开始，编组成场位--单位控制

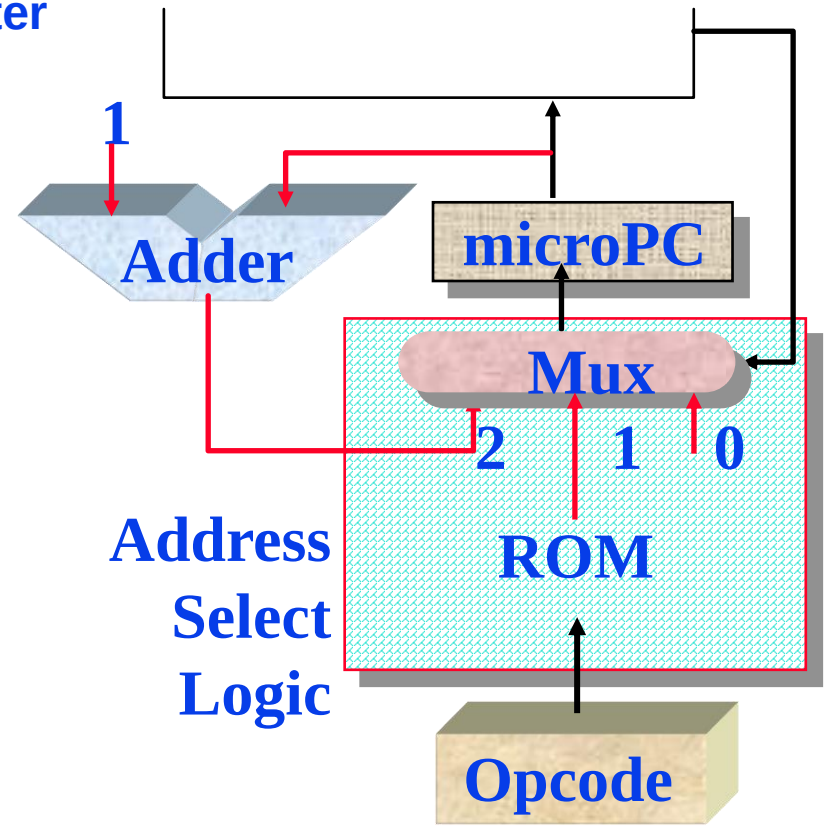
| 信号名称 | 为“0”时的效果 | 为“1”时的效果 |
|---|---|---|
| ALUSelA | 1st ALU operand = PC | 1st ALU operand = Reg[rs] |
| RegWrite | None | Reg. is written |
| MemtoReg | Reg. write data input = ALU | Reg. write data input = memory |
| RegDst | Reg. dest. no. = rt | Reg. dest. no. = rd |
| TargetWrite | None | Target reg. = ALU |
| MemRead | None | Memory at address is read |
| MemWrite | None | Memory at address is written |
| IorD | Memory address = PC | Memory address = ALU |
| IRWrite | None | IR = Memory |
| PCWrite | None | PC = PCSource |
| PCWriteCond | None | IF ALUzero then PC = PCSource |

# 1&2）从控制信号开始，编组成场位--多位控制

| 信号名称 | 数值 | 效果 |
|---|---|---|
| ALUOp | 00 | ALU adds |
| | 01 | ALU subtracts |
| | 10 | ALU does function code |
| | 11 | ALU does logical OR |
| ALUSelB | 000 | 2nd ALU input = Reg[rt] |
| | 001 | 2nd ALU input = 4 |
| | 010 | 2nd ALU input = sign extended IR[15-0] |
| | 011 | 2nd ALU input = sign extended, shift left 2 IR[15-0] |
| | 100 | 2nd ALU input = zero extended IR[15-0] |
| PCSource | 00 | PC = ALU |
| | 01 | PC = Target |
| | 10 | PC = PC+4[29-26] : IR[25-0] << 2 |

# 从一系列控制信号开始设计

- *使用上一讲介绍的基于的时序部件控制部件, 来完成下一状态函数* (下一微指令的地址)
  - 称为MicroPC或者 ηPC　与　state register




| *信号* | *数值* | *功效* |
|---|---|---|
| Sequencing | 00 | Next Address = 0 |
| | 01 | Next Address = dispatch ROM |
| | 10 | Next Address = Address + 1 |

# 3) 微指令格式：非编码 与 编码 场位

| 场位名称 | 宽度 | | 控制信号集合 |
|---|---|---|---|
| | 宽 | 窄 | |
| ALU Control | 4 | 2 | ALUOp |
| SRC1 | 2 | 1 | ALUSelA |
| SRC2 | 5 | 3 | ALUSelB |
| ALU Destination | 6 | 4 | RegWrite, MemtoReg, RegDst, TargetWr. |
| Memory | 4 | 3 | MemRead, MemWrite, IorD |
| Memory Register | 1 | 1 | IRWrite |
| PCWrite Control | 5 | 4 | PCWrite, PCWriteCond, PCSource |
| Sequencing | 3 | 2 | AddrCtl |
| Total width | 30 | 20 | bits |

# 4) 场位的含义 和 符号名称

| 场位名称 | 场位值 | 具有指定数值的场位的功能 |
|---|---|---|
| ALU | Add | ALU adds |
| | Subt. | ALU subtracts |
| | Func code | ALU does function code |
| | Or | ALU does logical OR |
| SRC1 | PC | 1st ALU input = PC |
| | rs | 1st ALU input = Reg[rs] |
| SRC2 | 4 | 2nd ALU input = 4 |
| | Extend | 2nd ALU input = sign ext. IR[15-0] |
| | Extend0 | 2nd ALU input = zero ext. IR[15-0] |
| | Extshft | 2nd ALU input = sign ex., sl IR[15-0] |
| | rt | 2nd ALU input = Reg[rt] |
| ALU destination | Target | Target = ALUout |
| | rd | Reg[rd] = ALUout |
| Memory | Read PC | Read memory using PC |
| | Read ALU | Read memory using ALU output |
| | Write ALU | Write memory using ALU output |
| Memory register | IR | IR = Mem |
| | Write rt | Reg[rt] = Mem |
| | Read rt | Mem = Reg[rt] |
| PC write | ALU | PC = ALU output |
| | Target-cond. | IF ALU Zero then PC = Target |
| | jump addr. | PC = PCSource |
| Sequencing | Seq | Go to sequential instruction |
| | Fetch | Go to the first microinstruction |
| | Dispatch | Dispatch using ROM. |

# 编写微程序

| *Label* | *ALU* | *SRC1* | *SRC2* | *ALU Dest.* | *Memory* | *Mem. Reg.* | *PC Write* | *Sequencing* |
|---|---|---|---|---|---|---|---|---|
| Fetch | Add | PC | 4 | | Read PC | IR | ALU | Seq |
| | Add | PC | Extshft | Target | | | | Dispatch |
| LW | Add | rs | Extend | | | | | Seq |
| | | | | | Read ALU | | | Seq |
| | | | | | | Write rt | | Fetch |
| SW | Add | rs | Extend | | | | | Seq |
| | | | | | Write ALU | Read rt | | Fetch |
| Rtype | Func | rs | rt | | | | | Seq |
| | | | | rd | | | | Fetch |
| BEQ1 | Subt. | rs | rt | | | | Target-cond. | Fetch |
| JUMP1 | | | | | | | jump address | Fetch |
| ORI | Or | rs | Extend0 | | | | | Seq |
| | | | | rd | | | | Fetch |

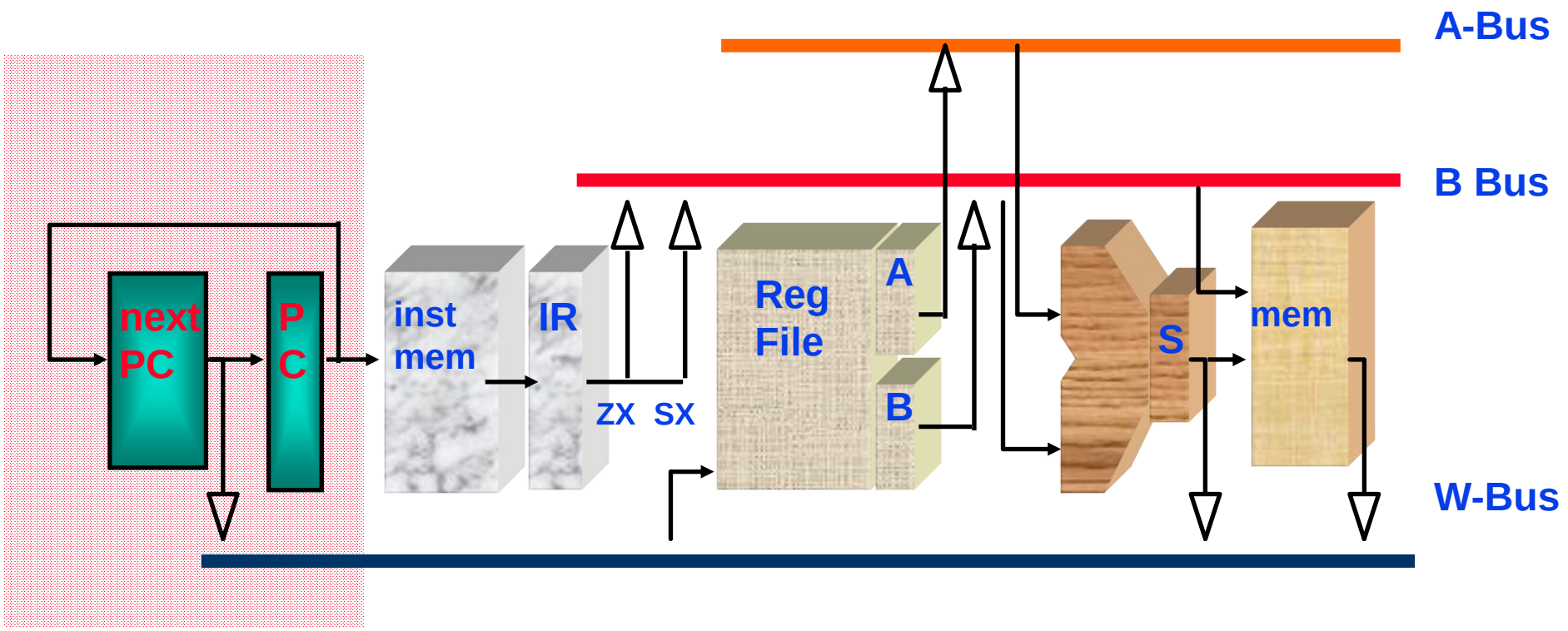
# 软件遗产和微程序设计

- IBM将整个公司投注于IBM 360指令系统体系结构 (ISA):
  单一指令系统支持很多种类的机器 (8位 至 64位)
- Stewart Tucker潜心研究如何支持软件的兼容性
- 如果微程序设计可以很轻易地在很多不同的微体系结构上实现相同的指令系统，那么为什么多种微程序不能也在相同的微系统结构上实现多种不同的指令系统呢？
- 派生出术语“仿真（ Emulation）” 针对其他指令系统的 微码形式的指令系统解释器
- 非常成功: 在 IBM 360 的早期，难以预知是老的指令系统，还是新的指令系统将会使用的更频繁

# 微程序设计的利弊

- 易于设计
- 灵活性
  - 易于适应组织、定时、工艺技术的变化
  - 在设计周期的后期也能够变化
- 可以实现功能非常强大的指令系统 (仅仅需要增加更多的控制存储器)
- 通用性
  - 可以在同一机器上实现多种不同的指令系统.
  - 可以针对具体应用，对指令系统进行剪裁.
- 兼容性
  - 多种组成， 同一指令系统
- 实现成本高
- 速度慢

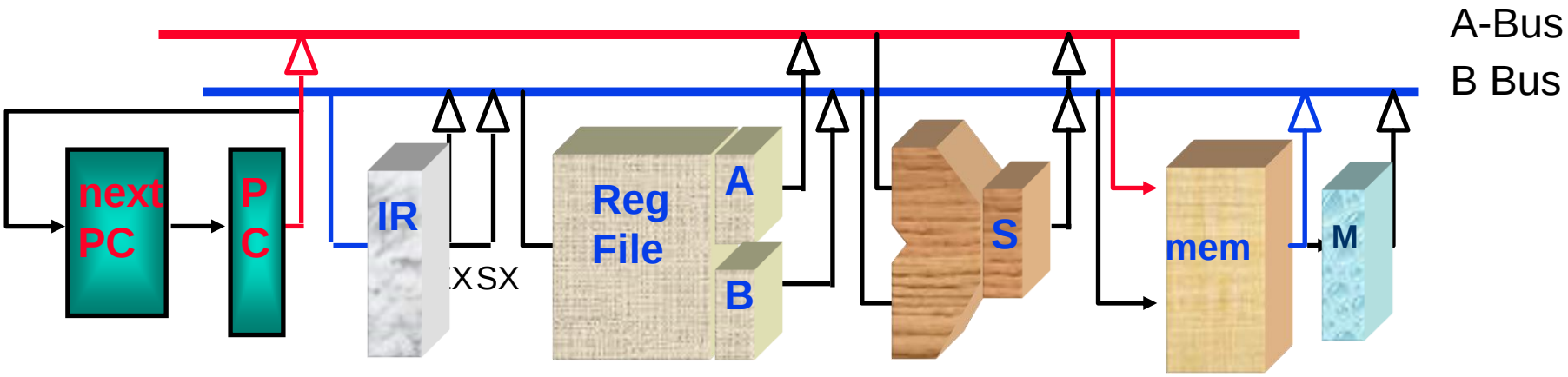
# 另外一种多周期数据通路



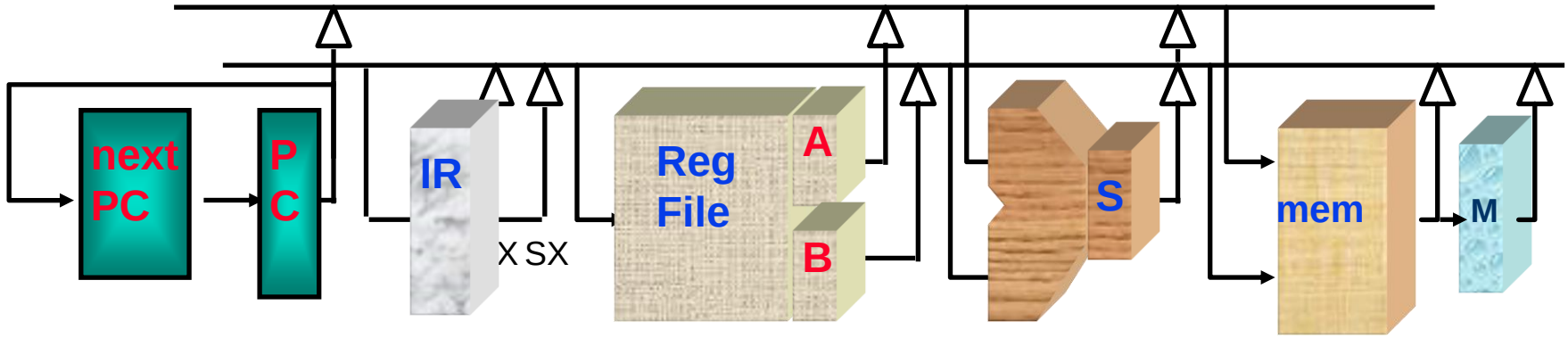

- 在每个时钟周期, 每条总线只能用于从一个源传送信息
- 指令只需要包括 B-Bus 和 W-Dst场位

# 双总线微体系结构(数据通路)

取指

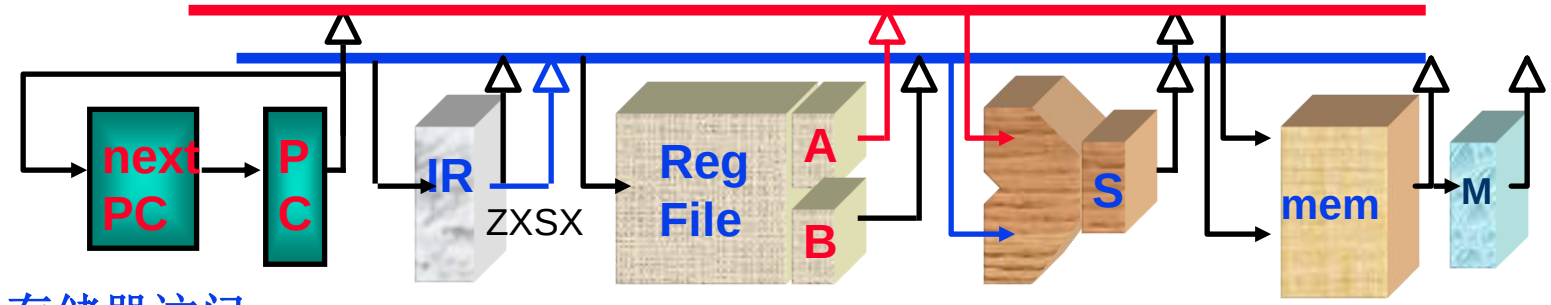



译码 / 取操作数

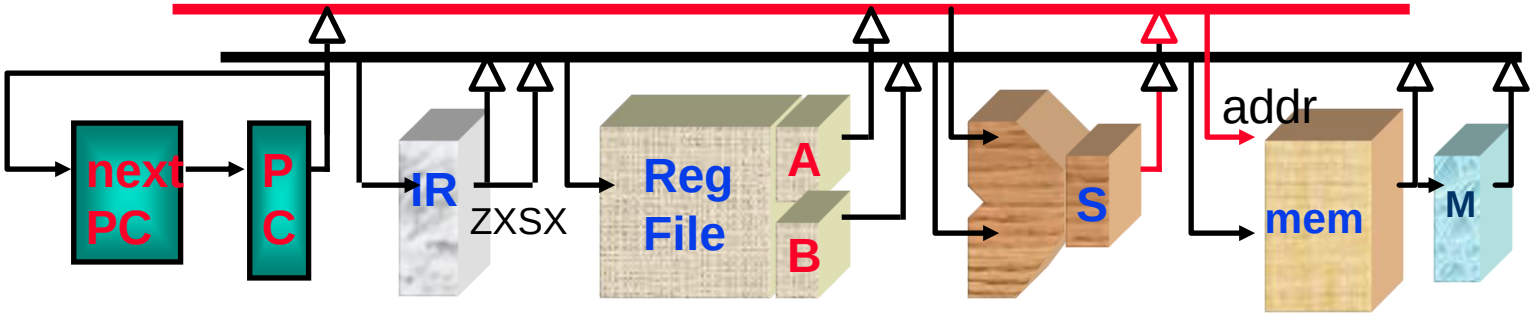


# Load

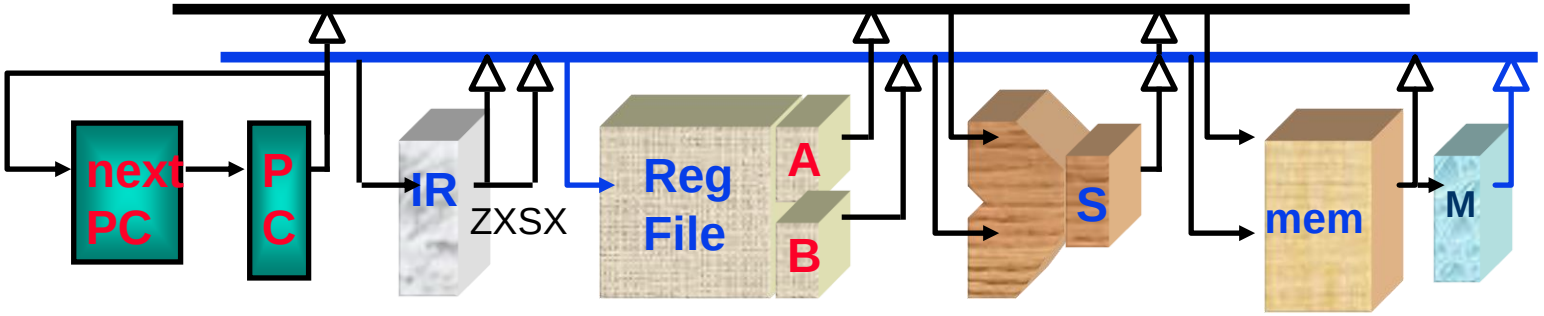
执行




存储器访问




回写



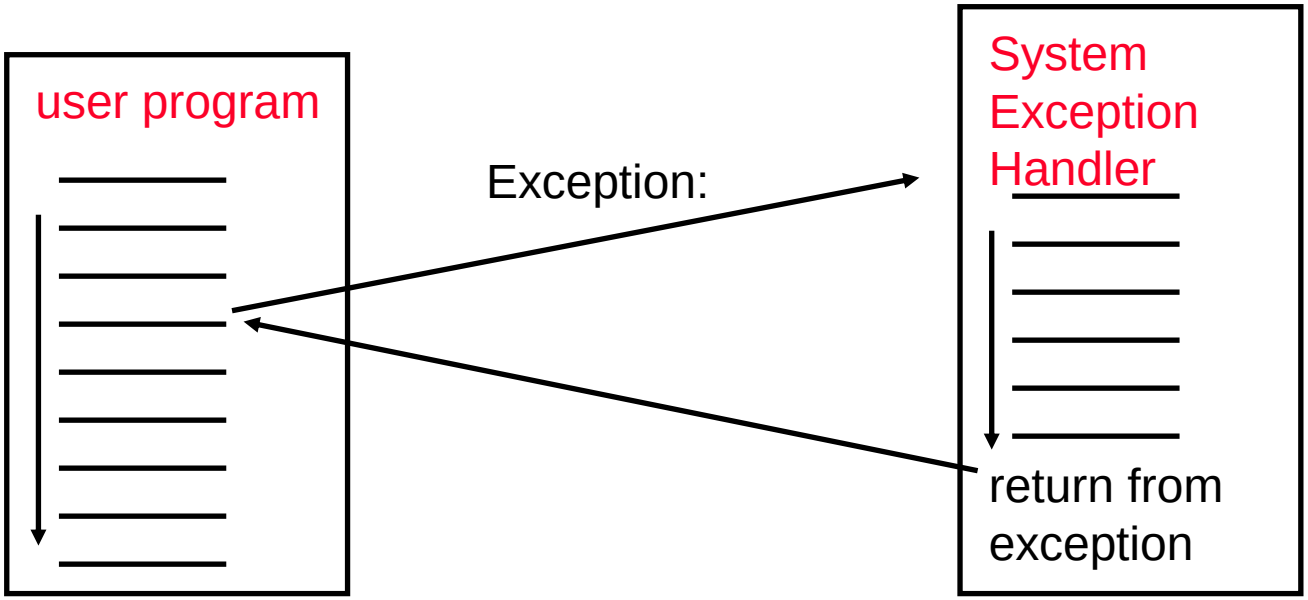

- 单总线+单加法器+单寄存器端口的情况如何?

# 意外和中断(Exceptions and Interrupts)

- 控制是设计中的难点
- 控制部分最难的是意外和中断
  - 除了转移和跳转之外,可以改变指令执行的正常流动的事件
  - 意外是来自处理器内部的不可预知的事件,例如,算术溢出
  - 中断 是来自处理器外部的不可预知的事件,例如, I/O
- MIPS的约定: 意外(exception)意味着任何改变控制流的不可预知的事件,它不区分外部和内部;
  当外部原因导致的事件出现时,才使用术语 “中断(interrupt)”

| *事件类型* | *来自哪里?* | *MIPS的术语* |
|---|---|---|
| I/O device request | External | Interrupt |
| Invoke OS from user program | Internal | Exception |
| Arithmetic overflow | Internal | Exception |
| Using an undefined instruction | Internal | Exception |
| Hardware malfunctions | Either | Exception or Interrupt |

# 意外事件




正常控制流: 串行执行、跳转、转移、调用、返回

- 意外 = 非程序控制的控制迁移
  - 系统要实施一定动作来处理意外
    - 必须纪录下来产生意外的指令的地址
  - 将控制交还给用户
  - 必须保存 &恢复用户态
- 支持构建一台用户的虚拟机器

# 产生意外的指令的后果

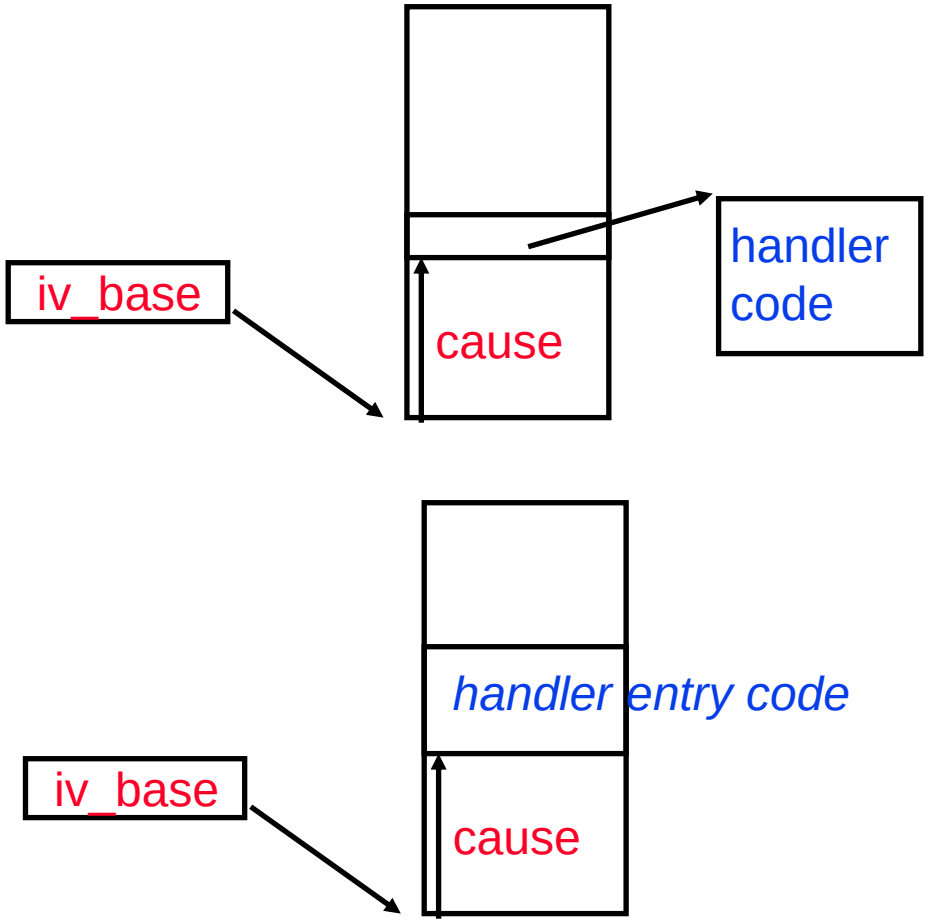
- MIPS体系结构定义如果一条指令产生了意外，这条指令将对机器状态不产生任何影响.
- 在虚拟存储器部分，我们将看到一些特定类型的意外要求我们必须能够阻止这些指令对机器状态产生任何影响.
- 处理意外的这些问题将导致处理器控制设计非常复杂，而且可能制约性能 => 为什么它是难点

# 两种类型的意外事件

- 中断
  - 由外部事件导致
  - 与程序执行异步
  - 可以在指令之间处理
  - 只需简单地挂起和恢复用户程序
- 自陷（Traps）
  - 由内部事件导致
    - 异常情况 (溢出)
    - 出错 (奇偶校验)
    - 失效 (非驻留页面)
  - 与程序执行同步
  - 必须由自陷处理程序来修复状态环境
  - 可以再试或模拟指令，程序可能继续执行，也可能中止

# 意外事件处理程序的寻址

- 传统技术: 中断向量
  - PC ⇐ MEM[ IV_base + cause || 00]
  - 370, 68000, Vax, 80x86, . . .
- RISC处理程序表（ Handler Table）
  - PC ⇐ IT_base + cause || 0000
  - 保存状态并跳转
  - Sparc, PA, M88K, . . .
- MIPS的技术: 固定入口（fixed entry）
  - PC ⇐ EXP_addr
  - 实际上是非常小的表
    - RESET entry
    - TLB
    - other



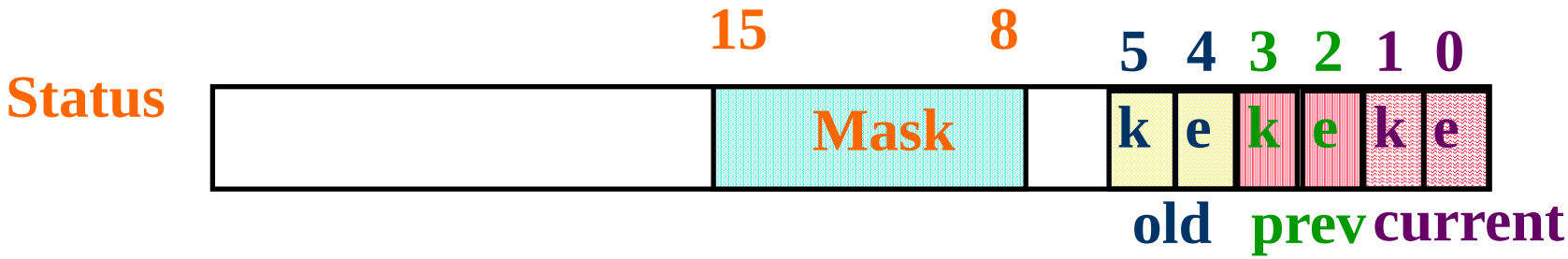
# 保存状态

- 将它压入堆栈中
  - **Vax, M68k, Intel 80x86**
- 将它保存在特殊寄存器中
  - **MIPS EPC, BadVaddr, Status, Cause**
- 影子寄存器
  - **Motorola 88000**
  - 将状态存储在内部流水线寄存器的影子中

# 为支持中断，需要对MIPS ISA增加什么？

- EPC - 32位寄存器, 用于保存受影响的指令的地址(协处理器0的寄存器 14).
- Cause - 用于纪录产生意外事件原因的寄存器. 在MIPS体系结构中这个寄存器 32位, 但是其中的一些位目前还没有用途. 假设这个寄存器的5到2位编码了上面介绍的两类可能的意外事件来源: 未定义指令=0 和算术溢出=1 (协处理器0的寄存器13).
- BadVAddr - 包括发生存储器访问的存储器地址的寄存器 (协处理器0的寄存器 8)
- Status - 中断屏蔽和使能位(协处理器0的寄存器12)
- 控制写入EPC 、Cause、BadVAddr和Status的控制信号
- 为了能够将意外事件地址写入PC, 增加多路选择器，以便增加输入 $01000000\ 00000000\ 00000000\ 01000000_{two}$ ($8000\ 0080_{hex}$)
- 需要能够撤销 PC = PC + 4, 这是因为我们希望EPC指向产生意外事件的指令（而不是它的后继指令）； PC = PC - 4

# 状态寄存器探秘

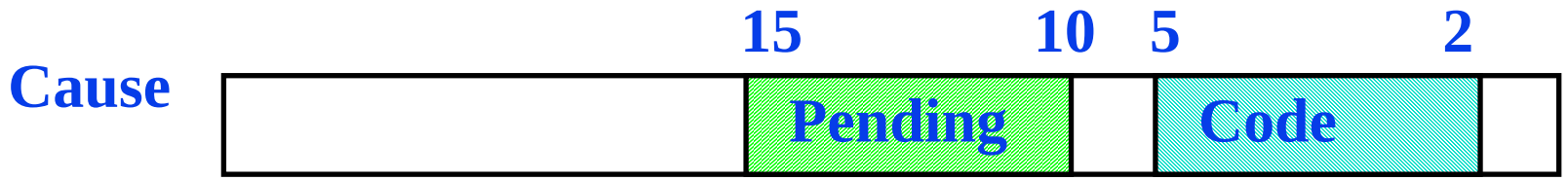



- Mask = 5级硬件中断和3级软件中断，每级中断一位
  - 1 => 使能中断
  - 0 => 废止中断
- k = 核心（kernel）/用户（user）
  - 0 => 当出现中断时，处于核心模式
  - 1 => 当出现中断时，处于运行用户模式
- e = 中断使能
  - 0 => 中断被废止
  - 1 => 中断被使能
- 当出现中断时，最低的（LSB）6位向左移2位, 将最低2位置为0
  - 在核心模式运行，中断废止

# 用户/系统模式

- 通过提供两种执行模式 (用户/系统)，计算机就可能自己管理自己
  - 操作系统是在特权模式运行的一种特殊程序，它可以访问计算机的所有资源
  - 向用户提供比实际物理资源更加便捷的虚拟资源
    - 文件 与 磁盘扇区
    - 虚拟存储器 与 物理存储器
  - 保护每个用户程序不受其他程序影响
- 当用户程序正在执行时，意外事件允许系统针对出现的事件采取行动
  - 在处理程序中，操作系统功能开始工作

# 原因寄存器（Cause register）探秘




- 未决中断（Pending interrupt） 5 个硬件级别: 如果出现了中断，但是还没有得到服务，就设置这些位
  - 当同时出现多个中断时，逐个处理；或在中断废止时，记录中断请求
- 意外事件代码（Exception Code） 对中断原因进行编码
  - 0 (INT) => 外部中断
  - 4 (ADDRL) => 地址错误意外事件 (装入数据 或 取指)
  - 5 (ADDRS) =>地址错误意外事件(存储数据)
  - 6 (IBUS) => 在取指时，总线错误
  - 7 (DBUS) =>在取数据时，总线错误
  - 8 (Syscall) => 系统调用意外事件
  - 9 (BKPT) => 断点意外事件（Breakpoint exception）
  - 10 (RI) => 保留指令意外事件
  - 12 (OVF) => 算术溢出意外事件

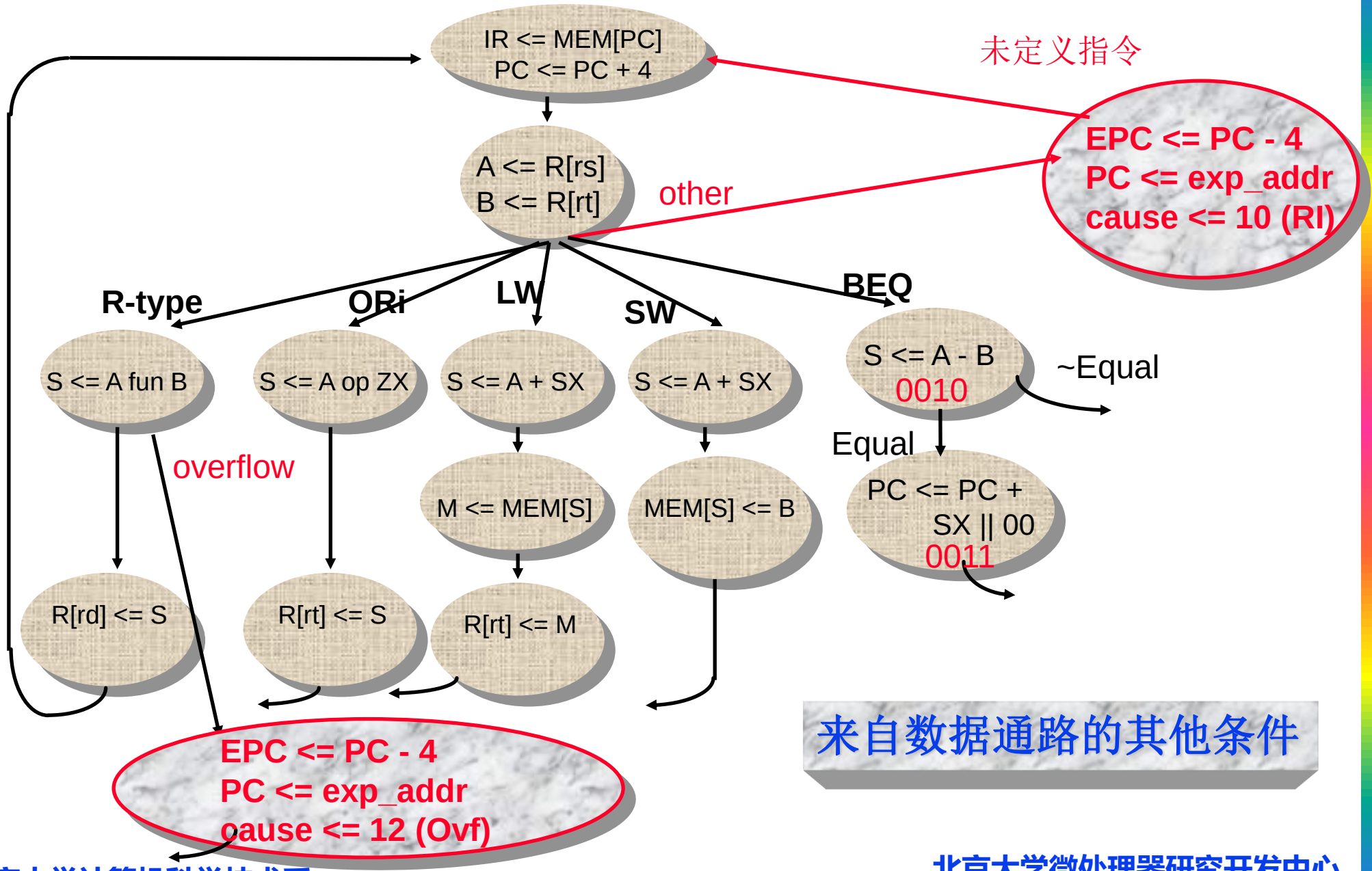
# 精确中断

- 精确 => 精确地保留程序执行到产生意外事件的指令时的机器的状态
  - 同一系统代码将能够在系统结构的不同实现上正常工作
  - IBM明确了精确中断的地位
  - 在采用流水技术、乱序执行，…等技术后，难以实现
  - MIPS实现了精确中断
- 非精确中断 => 系统程序不得不判断在哪里出现了中断，并一起卷回
- 为了追求性能，设计人员有时不得不放弃精确中断
  - 系统软件开发人员、用户、市场销售等等通常希望他们不这样做

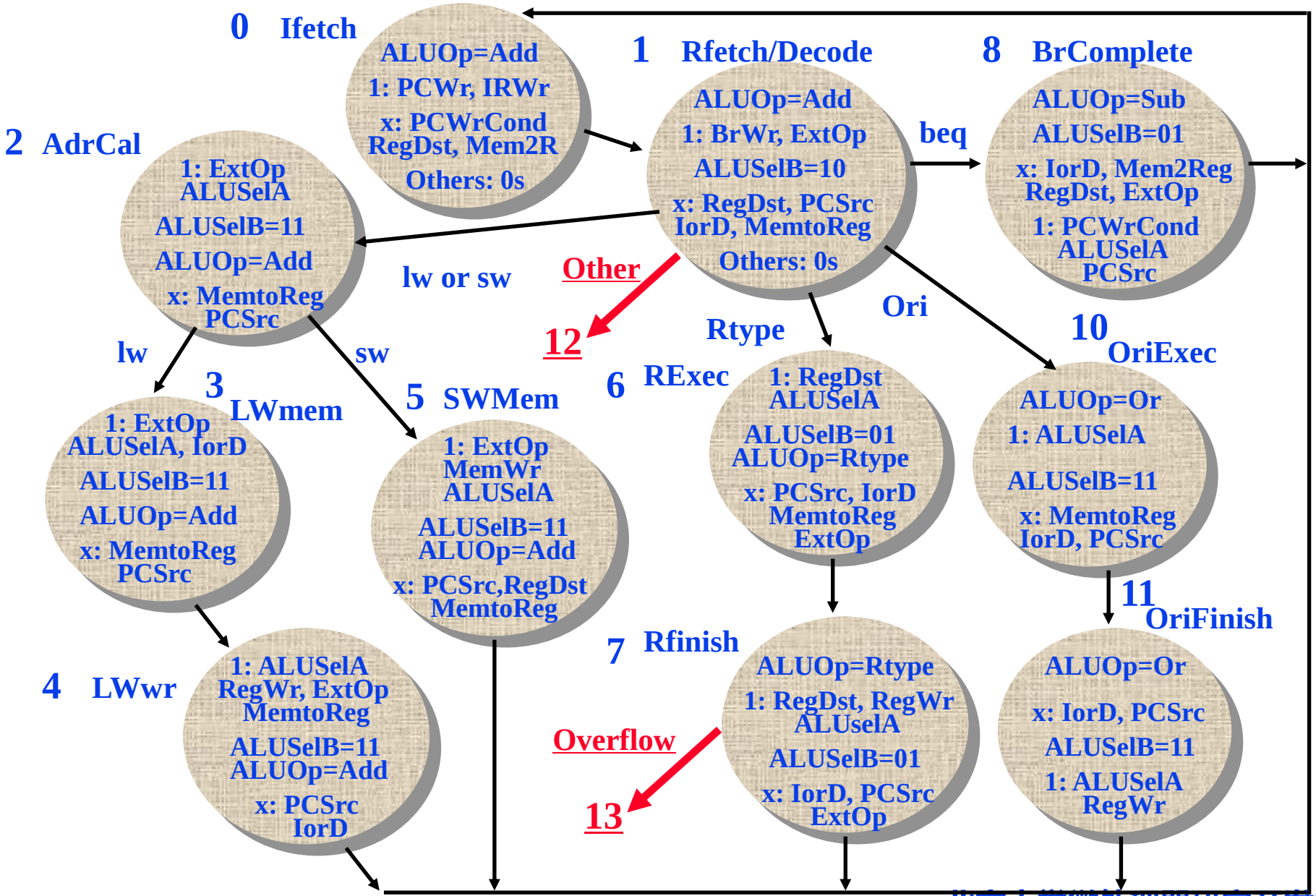
# 在我们的FSD中控制如何检测意外事件？

- 对于特定的操作码数值，如果状态1没有后续状态，那么就检测到了“非定义指令”.
  - 为处理这一中断，我们定义一个新的后续状态来接受所有非定义的操作码
  - 用“Other”来指明这些非定义的操作码
- 第四章中介绍ALU中检测溢出的逻辑, 并且ALU能够提供一个称为“Overflow”的信号。在修改后的有穷状态机中将使用这一信号来指定另外一个可能的后续状态。
- 注释: 在设计实际机器时面临的一大挑战是处理 指令与其他由于意外事件而产生的事件（例如，控制逻辑要保持小而快）之间的不同相互作用
  - 这种相互作用非常复杂，从而使得控制部件是硬件设计的一大难点

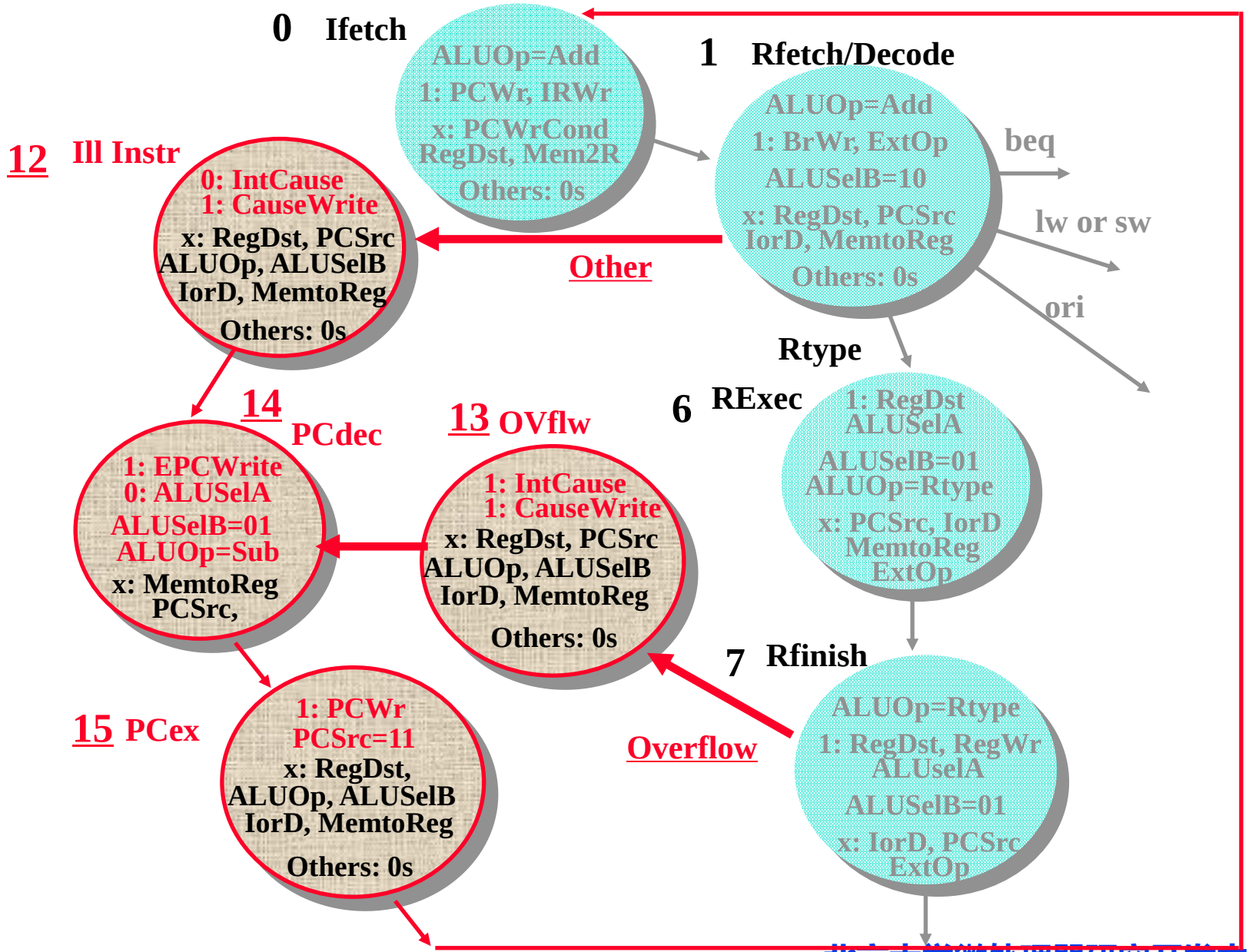
# 对控制描述进行修改


IR <= MEM[PC]
PC <= PC + 4
未定义指令
EPC <= PC - 4
PC <= exp_addr
cause <= 10 (RI)
A <= R[rs]
B <= R[rt]
other
R-type
ORi
LW
SW
BEQ
S <= A fun B
S <= A op ZX
S <= A + SX
S <= A + SX
S <= A - B
0010
~Equal
Equal
overflow
M <= MEM[S]
MEM[S] <= B
PC <= PC +
SX || 00
0011
R[rd] <= S
R[rt] <= S
R[rt] <= M
EPC <= PC - 4
PC <= exp_addr
cause <= 12 (Ovf)
来自数据通路的其他条件

# 为检测意外事件，对有穷状态图进行修改



# 处理意外事件的附加状态



# 总结

- 微序列器很容易处理专门处理后的状态图
  - 简单递增 & 转移场位
  - 数据通路控制场位
- 控制设计简化成微程序设计
- 意外处理是控制部分的难点
- 需要找到可以保存PC、并激活操作系统功能的方便地方来放置检测意外事件、转移到状态或微指令
- 当以后我们学习支持存储访问出现页面失效的流水化CPU时，由于指令不能完成，并且 恰好要在产生意外事件的指令处重启程序，这使得控制的设计难上加难！

# 微程序设计对RISC产生的灵感

- 如果简单指令能够以很高的时钟频率执行
- 如果人们有能力编写产生微指令的编译器
- 如果绝大多数程序仅仅使用简单指令和简单的寻址方式
- 如果微码保存在RAM中，而非ROM中，那么就易于排错
- 如果用于控制存储器的同一存储器还可被用为宏指令的 cache
- 那么，为什么不用微程序和可直接产生机器最低级语言的编译器来跨越指令解释器呢？

# 总结：多周期控制

- 微程序设计和硬联线控制具有很多相似之处，或许它们之间最大的不同是初始表示和实现变更的难易， 使用 ROM通常比使用PLA更容易

初始表示
Initial Representation

时序控制
Sequencing Control

逻辑表示
Logic Representation

实现技术
Implementation Technique