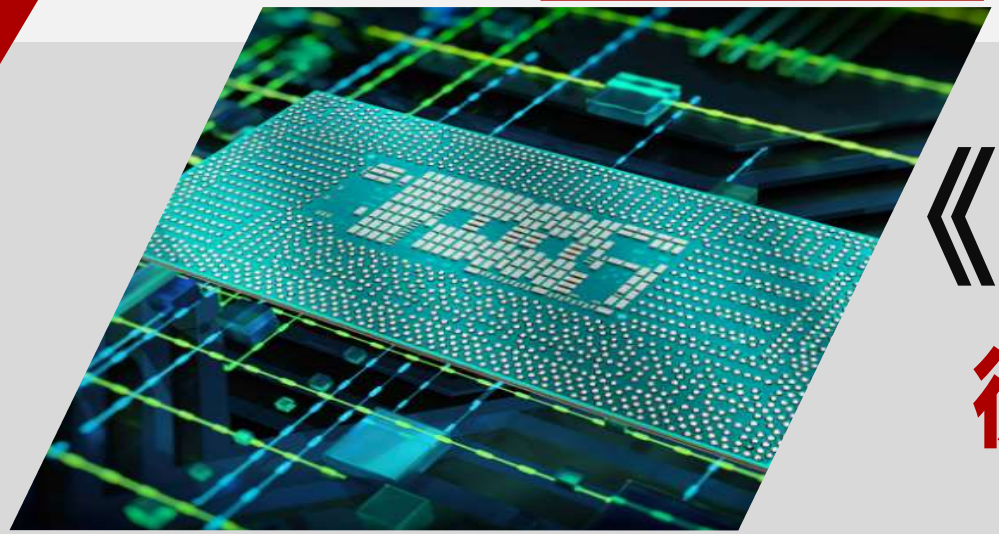
2023年春季学期



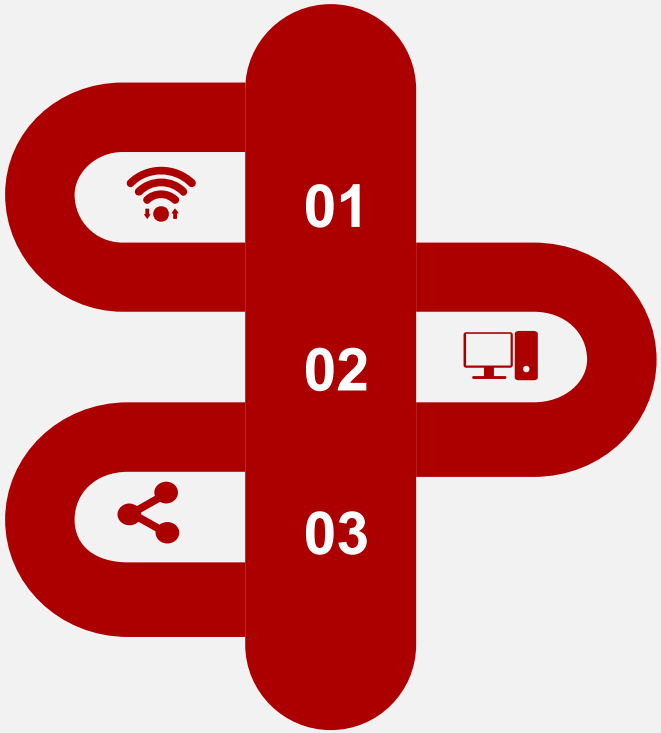
《计算机系统》

微序列控制器

《计算机系统》课程教学组

内容提要

微序列控制器
设计基础



非常简单CPU的
微序列控制器

微程序控制和硬
布线控制的比较

微序列控制器

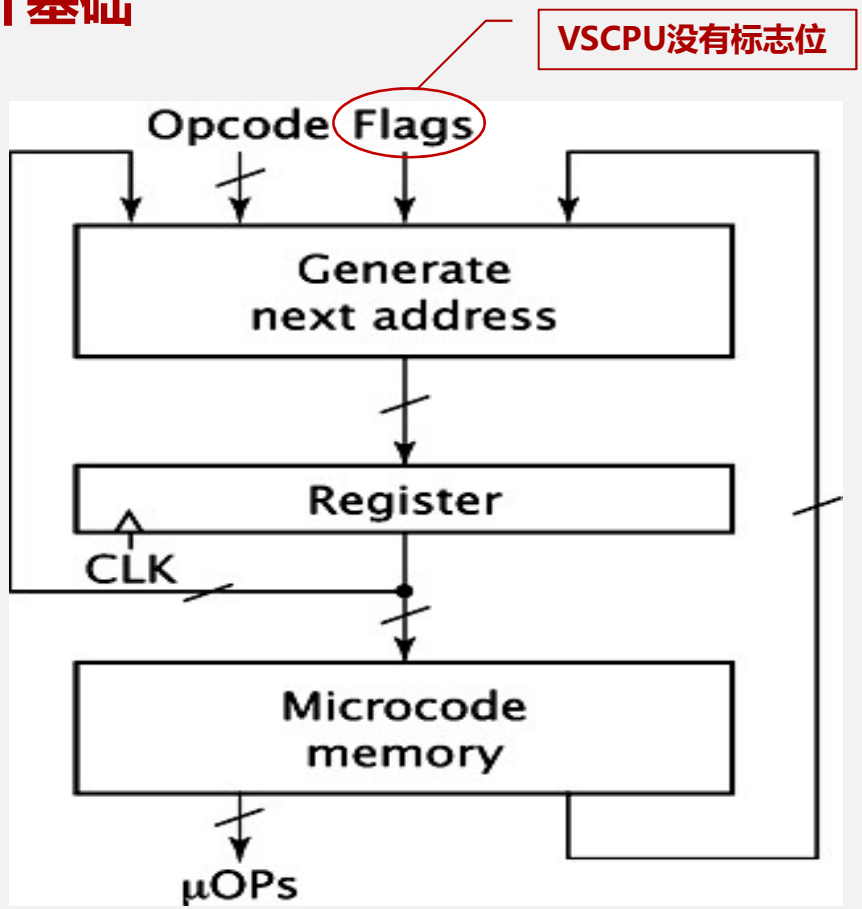
把控制信号存储在一个查找表**ROM**，即**微代码存储器**中。按照正确的次序访问ROM中的内容，查找ROM就可以**按适当的顺序**发出控制信号，从而实现处理器指令集中的指令。

微序列控制器设计基础

微序列控制器的操作

1．典型微序列控制器的组成

- 寄存器值与CPU状态图中的一个状态相对应，且当作地址输入到微代码存储器中。
- 存储器的输出是一条微指令，即那个地址对应的存储单元内容。所有微指令集中组成CPU的微代码或微程序。
- 下址产生模块产生所有可能的下一地址，然后从中选择一个正确的下址送入寄存器中。



微序列控制器设计基础

微序列控制器的操作

2．微指令的组成

微指令由几个域组成，其bit位可分为两组。

- **微操作（微序列控制器的微序列部分）**

这些信号从微序列控制器输出到CPU的其他部分，它们或者输入到组合逻辑以生成CPU的控制信号，或者直接产生控制信号。

- **下一个地址（微序列控制器的序列控制部分）**

用来产生存储到寄存器中的下一个地址，这些位连同指令的操作码和标志值一起输入到组合逻辑以产生下一条微指令的地址。

微序列控制器的操作

3．下址的产生

四种常用方法：

- **当前地址加1**：微代码存储器中的下一地址，即当前地址加1（顺序存放，可读性高，易于调试）。
- **绝对地址**：由微代码存储器提供。
- **映射逻辑**：已读取指令的操作码输入到一个映射硬件，此硬件将该操作码转换或映射成该指令执行周期的第一条微指令所在的地址；这一地址被装载到寄存器，微序列控制器就能转入到正确的执行过程。该映射硬件只在取指令周期的末端（FETCH3）使用一次 。
- **微子程序**：子程序入口地址是由微代码存储器给出绝对地址。返回地址是当前地址加1，存储在微子程序寄存器或硬件堆栈中，此地址用于从微子程序中返回。

微序列控制器设计基础

微指令的格式

1．典型的微指令格式

SELECT: 选择域，指明下一条微指令地址的来源。

ADDR: 地址域，指明一个绝对地址。

μOP: 微操作域，一个或多个。

微序列控制器设计基础

微指令的格式

2．说明微操作的三种主要方法

a. **水平微代码:** 列出CPU所要执行的每一个微操作,将微指令微操作域中的一位分配给每个微操作。

- **一个CPU需要执行50个微操作，它的每一条微指令的微操作域将占用50位。**
- **微指令太长，大部分微操作位是无效的**

b. **垂直微代码:** 所有微操作组合成域，域中的每个微操作被分配一个唯一的编码值。

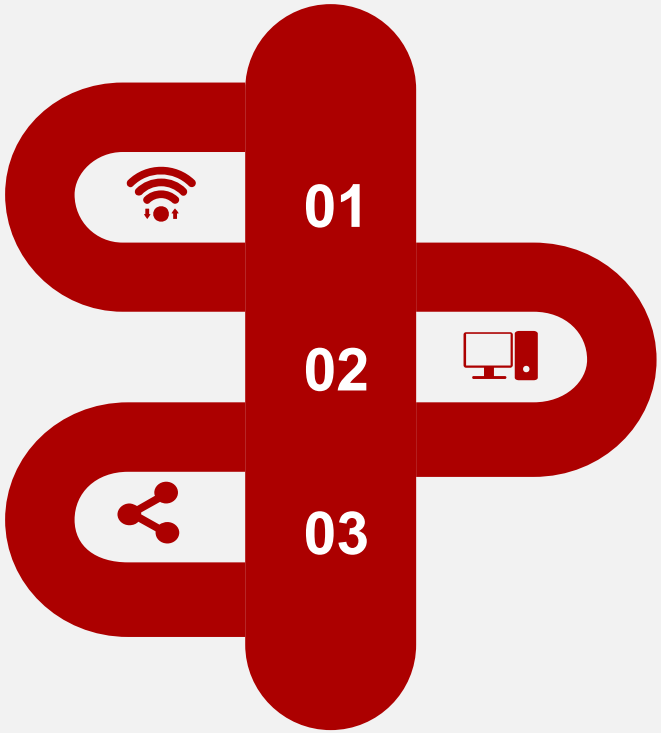
- **例如：16个微操作 —— 四位二进制来编码（0000 ~ 1111 ）**
- **微指令位数少，但需要译码器，所有微操作组合成域**
- **域中的每个微操作被分配一个唯一的编码值**

c. **控制信号的直接生成:** 微代码将微操作组合在一起，控制信号值直接存储在微指令中。

- **代码的可读性差，调试困难。**

内容提要

微序列控制器
设计基础



非常简单CPU的
微序列控制器

微程序控制和硬
布线控制的比较

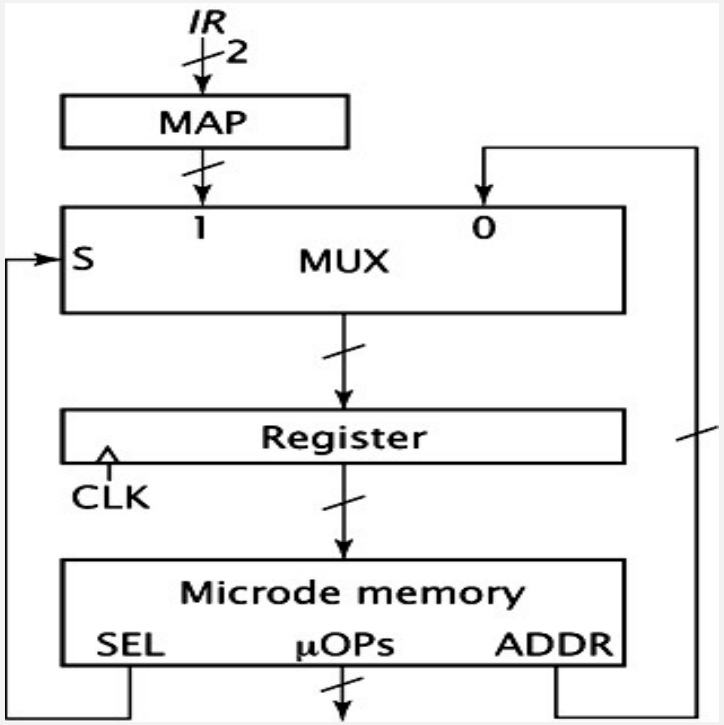
VS-CPU的微序列控制器

非常简单CPU

指令集、有限状态机、数据通路和ALU单元都是相同的，CPU内部的数据流也没有变化，只有控制信号的产生方法有所改变。

VSCPU微序列控制器基本布局

1. 非常简单微序列控制器的基本结构



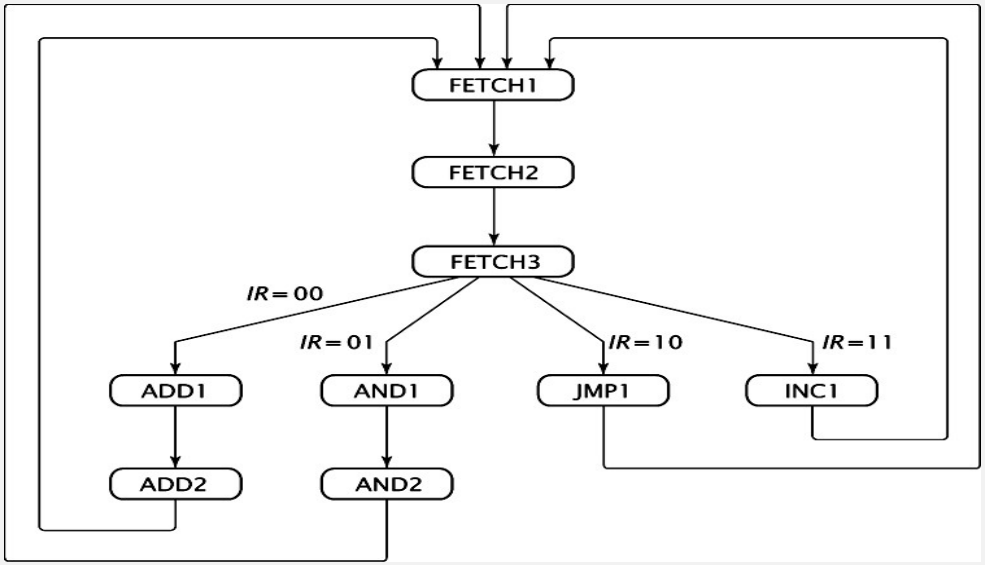
VS-CPU的微序列控制器

非常简单CPU

微序列控制器基本布局

2.只用到两种可能的下址方式

- **操作码映射**
- **绝对跳转**



为什么可以这样做？ (考虑非常简单CPU的状态图)

- 取指令周期的最后一个状态FETCH3可以转到四个执行周期中的一个（IR导致的分支），这必须通过映射输入来实现。
- 剩余的每一状态都必须转到一个具体的次态，可以通过绝对跳转来实现。

VS-CPU的微序列控制器

微序列控制器的设计

一、 确定从微代码存储器输出的绝对地址宽度

- VSCPU的状态图中共有九种状态，每种状态代表一条微指令，从中选择的话最少需要4位绝对地址。既然映射硬件也生成同样宽度的地址，那么输出的地址就是4位宽度。

- 多路选择器输出到寄存器的输入、以及寄存器输出到微代码存储器的输入，也都是4位宽度。

VS-CPU的微序列控制器

必须确定的几件事情

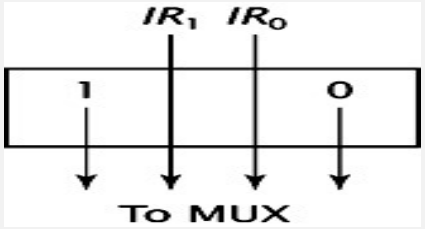
二、 生成正确序列、设计映射逻辑并设计控制有限状态机状态转换的微代码

1. 给有限状态机的每种状态分配一个微代码地址：

每一执行周期的第一个状态的地址分配（它决定了实现映射功能的逻辑）

FETCH3 映射函数：**1IR[1..0]0** 状态**ADD1**、**AND1**、**JMP1**和**INC1** → 1000、1010、1100和1110

映射逻辑



状态	地址
FETCH1	0000 (0)
FETCH2	0001 (1)
FETCH3	0010 (2)
ADD1	1000 (8)
ADD2	1001 (9)
AND1	1010 (10)
AND2	1011 (11)
JMP1	1100 (12)
INC1	1110 (14)

VS-CPU的微序列控制器

二、 生成正确序列、设计映射逻辑并设计控制有限状态机状态转换的微代码

2. 生成正确的微代码序列：

- 为了无条件转移到一种具体的状态，微序列控制器通过地址域和选择域来提供状态的地址

例如： FETCH1 → FETCH2

0号单元：SEL=0 ADDR=0001

  - 0号单元对应于状态FETCH1
  - SEL=0使微序列控制器从地址域得到它的下一地址
  - 把地址域置为0001，使它转到状态FETCH2所对应的单元。

- FETCH3必须映射到正确的执行周期，所以要求 SEL=1，表示采用映射地址。

状态	地址	SEL	下址
FETCH1	0000 (0)	0	0001
FETCH2	0001 (1)	0	0010
FETCH3	0010 (2)	1	XXXX
ADD1	1000 (8)	0	1001
ADD2	1001 (9)	0	0000
AND1	1010 (10)	0	1011
AND2	1011 (11)	0	0000
JMP1	1100 (12)	0	0000
INC1	1110 (14)	0	0000

VS-CPU的微序列控制器

二、 生成正确序列、设计映射逻辑并设计控制有限状态机状态转换的微代码

3. 用水平微代码生成微操作

- 9个状态下共有9个微操作（巧合），需要9个bit来表示各自是否发生（0-不发生；1-发生）
- VS-CPU的微操作及它们的助记符见**右表**

FETCH1:	AR←PC
FETCH2:	DR←M, PC←PC+1
FETCH3:	IR←DR[7..6], AR←DR[5..0]
ADD1:	DR←M
ADD2:	AC←AC+DR
AND1:	DR←M
AND2:	AC←AC^DR
JMP1:	PC←DR[5..0]
INC1:	AC←AC+1

助记符	微操作
ARPC	AR←PC
ARDR	AR←DR[5..0]
PCIN	PC←PC+1
PCDR	PC←DR[5..0]
DRM	DR←M
IRDR	IR←DR[7..6]
PLUS	AC←AC+DR
AND	AC←AC^DR
ACIN	AC←AC+1

VS-CPU的微序列控制器

二、 生成正确序列、设计映射逻辑并设计控制有限状态机状态转换的微代码

3. 用水平微代码生成微操作

VS-CPU的水平微代码

状态	地址	SEL	ARPC	ARDR	PCIN	PCDR	DRM	IRDR	PLUS	AND	ACIN	下址
FETCH1	0000 (0)	0	1	0	0	0	0	0	0	0	0	0001
FETCH2	0001 (1)	0	0	0	1	0	1	0	0	0	0	0010
FETCH3	0010 (2)	1	0	1	0	0	0	1	0	0	0	XXXX
ADD1	1000 (8)	0	0	0	0	0	1	0	0	0	0	1001
ADD2	1001 (9)	0	0	0	0	0	0	0	1	0	0	0000
AND1	1010 (10)	0	0	0	0	0	1	0	0	0	0	1011
AND2	1011 (11)	0	0	0	0	0	0	0	0	1	0	0000
JMP1	1100 (12)	0	0	0	0	1	0	0	0	0	0	0000
INC1	1110 (14)	0	0	0	0	0	0	0	0	0	1	0000

- 在所有的状态中，ARDR和IRDR的值是相同的。
- 用一个输出AIDR来驱动这两个微操作。AIDR组合了两个微操作 AR←DR[5..0] 和 IR←DR[7..6]。

VS-CPU的微序列控制器

二、 生成正确序列、设计映射逻辑并设计控制有限状态机状态转换的微代码

3. 用水平微代码生成微操作

优化后的VS-CPU的水平微代码

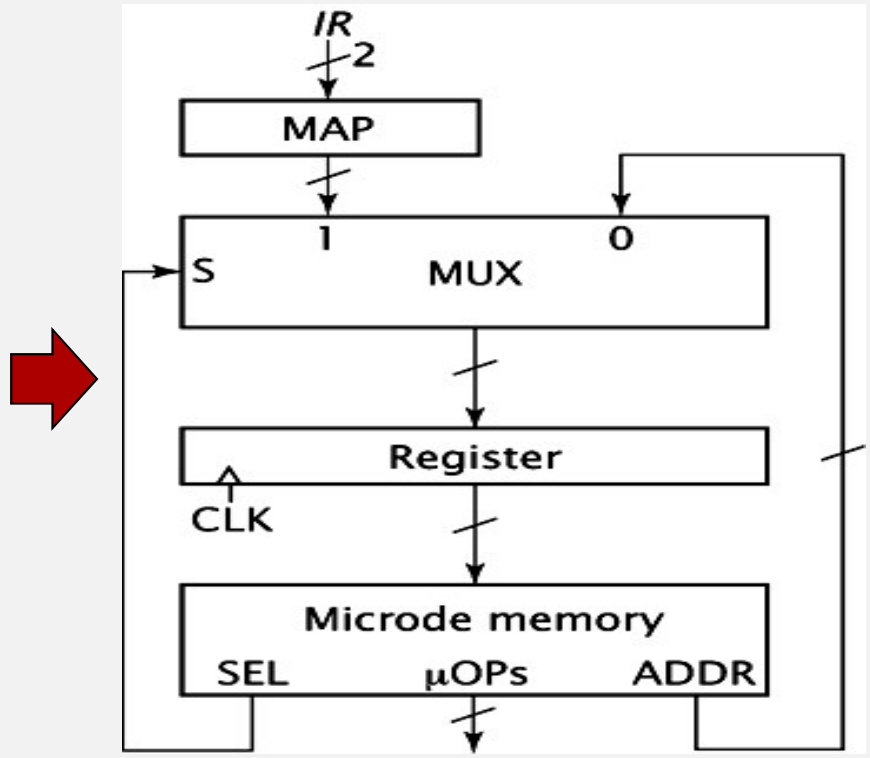
状态	地址	SEL	ARPC	AIDR	PCIN	PCDR	DRM	PLUS	AND	ACIN	下址
FETCH1	0000 (0)	0	1	0	0	0	0	0	0	0	0001
FETCH2	0001 (1)	0	0	0	1	0	1	0	0	0	0010
FETCH3	0010 (2)	1	0	1	0	0	0	0	0	0	XXXX
ADD1	1000 (8)	0	0	0	0	0	1	0	0	0	1001
ADD2	1001 (9)	0	0	0	0	0	0	1	0	0	0000
AND1	1010 (10)	0	0	0	0	0	1	0	0	0	1011
AND2	1011 (11)	0	0	0	0	0	0	0	1	0	0000
JMP1	1100 (12)	0	0	0	0	1	0	0	0	0	0000
INC1	1110 (14)	0	0	0	0	0	0	0	0	1	0000

VS-CPU的微序列控制器

二、 生成正确序列、设计映射逻辑并设计控制有限状态机状态转换的微代码

3. 用水平微代码生成微操作

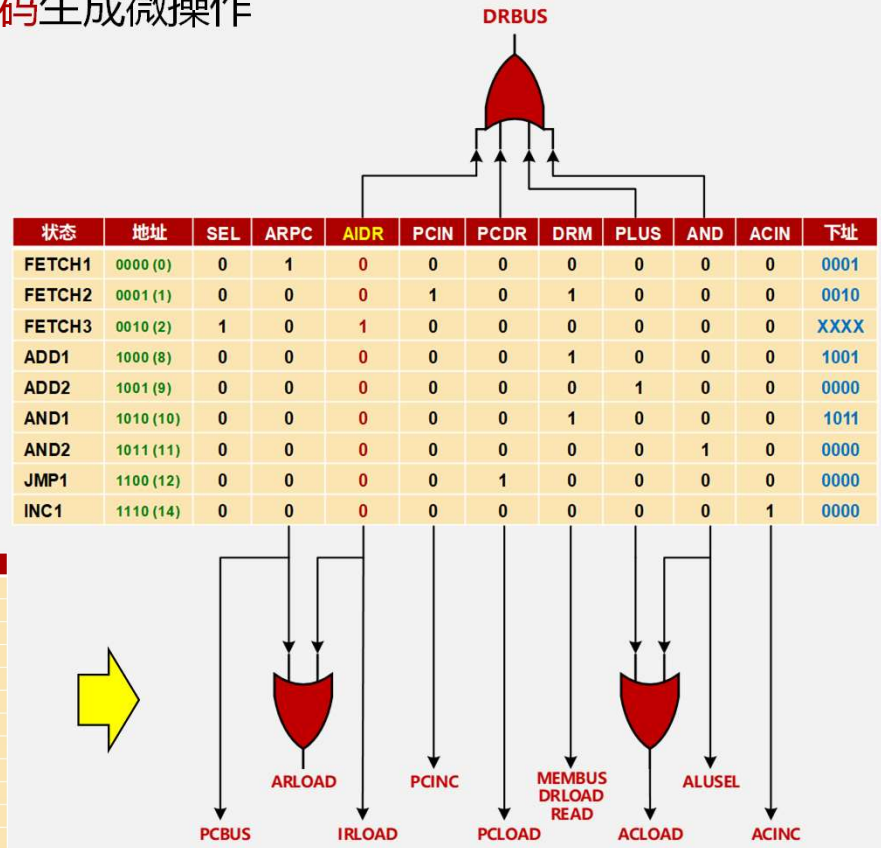
控制信号	微操作
ARLOAD	ARPC v AIDR
PCLOAD	PCDR
PCINC	PCIN
DRLOAD	DRM
ACLOAD	PLUS v AND
ACINC	ACIN
IRLOAD	AIDR
ALUSEL	AND
MEMBUS	DRM
PCBUS	ARPC
DRBUS	AIDR v PCDR v PLUS v AND
READ	DRM



VS-CPU的微序列控制器

二、 生成正确序列、设计映射逻辑并设计控制有限状态机状态转换的微代码

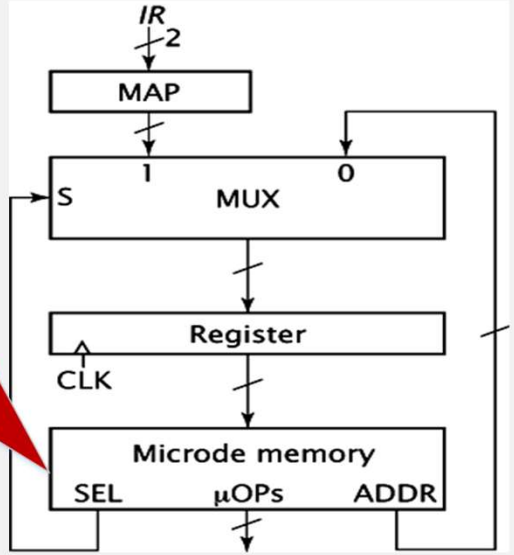
3. 用水平微代码生成微操作



状态	地址	SEL	ARPC	AIDR	PCIN	PCDR	DRM	PLUS	AND	ACIN	下址
FETCH1	0000 (0)	0	1	0	0	0	0	0	0	0	0001
FETCH2	0001 (1)	0	0	0	1	0	1	0	0	0	0010
FETCH3	0010 (2)	1	0	1	0	0	0	0	0	0	XXXX
ADD1	1000 (8)	0	0	0	0	0	1	0	0	0	1001
ADD2	1001 (9)	0	0	0	0	0	0	1	0	0	0000
AND1	1010 (10)	0	0	0	0	0	1	0	0	0	1011
AND2	1011 (11)	0	0	0	0	0	0	0	1	0	0000
JMP1	1100 (12)	0	0	0	0	1	0	0	0	0	0000
INC1	1110 (14)	0	0	0	0	0	0	0	0	1	0000

控制信号	微操作
ARLOAD	ARPC v AIDR
PCLOAD	PCDR
PCINC	PCIN
DRLOAD	DRM
ACLOAD	PLUS v AND
ACINC	ACIN
IRLOAD	AIDR
ALUSEL	AND
MEMBUS	DRM
PCBUS	ARPC
DRBUS	AIDR v PCDR v PLUS v AND
READ	DRM

微指令→微操作→控制信号



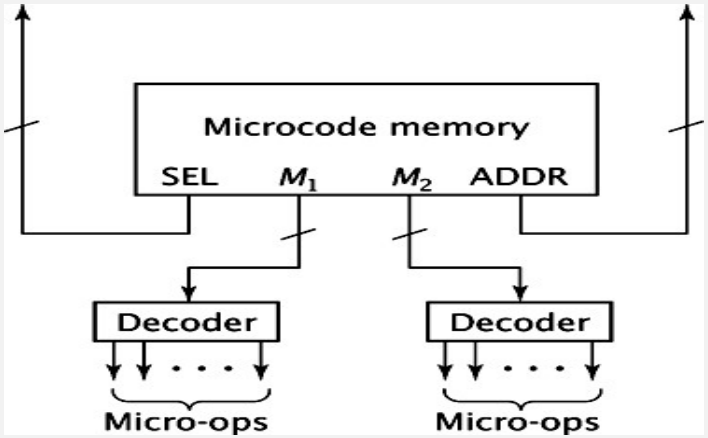
VS-CPU的微序列控制器

二、 生成正确序列、设计映射逻辑并设计控制有限状态机状态转换的微代码

4. 用垂直微代码生成微操作

在垂直微代码中，所有的微操作被分组到不同的域中，使得任何状态一个域中最多只有一种微操作是有效的，然后域中的每个微操作被赋予一个唯一的域值。

- **8种不同的微操作 —— 3位二进制（从000到111之间的任何一个值）**
- **微操作域位从微代码存储器输出到一个译码器，译码器的输出就是在水平微代码中直接产生的微操作。**



VS-CPU的微序列控制器

二、 生成正确序列、设计映射逻辑并设计控制有限状态机状态转换的微代码

4. 用垂直微代码生成微操作

给各种微操作分配不同域的原则：

- **对于同一状态下发生的两个不同的微操作，将它们分配到不同的域中。每个域在一个周期中只能输出唯一一个微操作的值，如果两个微操作要同时出现，则它们不可能在同一个域中。**
- **必要的话在每个域中包括一个NOP操作。**
- **分配剩下的微操作以便充分地利用微操作的域位。**
- **把修改相同的寄存器的微操作组合在同一个域中。**

VS-CPU的微序列控制器

二、 生成正确序列、设计映射逻辑并设计控制有限状态机状态转换的微代码

4. 用垂直微代码生成微操作

a. 检查同时进行的微操作：

- 例如DRM和PCIN都在FETCH2状态下发生，那么这两种微操作必须被分配到不同的域中。则该CPU的微操作至少需要两个域，我们分别把它标记为M1和M2，包括NOP操作在内每个域的微操作如**右表**：

M1	M2
NOP	NOP
DRM	PCIN

b. 以此类推地分配剩余的微操作到M1和M2

- PCIN和PCDR都能修改PC值，所以我们现在把PCDR也加到M2中。把剩下的微操作任意分配给这两个域，但要注意把那些改变同一个寄存器值的微操作放到同一个域中。

M1	M2
NOP	NOP
DRM	PCIN
ACIN	PCDR
PLUS	ARPC
AND	AIDR

每个域有五种微操作，因此每个域需要3 bits，一共6 bits

VS-CPU的微序列控制器

二、 生成正确序列、设计映射逻辑并设计控制有限状态机状态转换的微代码

4. 用垂直微代码生成微操作

a. 对这种分配进行一些调整，减少总的位数：

- 将AIDR从M2移到M1。M2的微操作数从5变到4，而M1的从5变到6。这时，M1仍需要3位，但是M2现在只需要2位，微代码的宽度就减少了1位。

- 把 ARPC 和 PCDR 都从M2 移到M1，这样一来M1就有8种微操作，需要3位；而M2只有2种微操作，需要1位。总共4位，可能是最少的了。

M1	
值	微操作
000	NOP
001	DRM
010	ARPC
011	AIDR
100	PCDR
101	PLUS
110	AND
111	ACIN

M2	
值	微操作
0	NOP
1	PCIN

VS-CPU的微序列控制器

二、 生成正确序列、设计映射逻辑并设计控制有限状态机状态转换的微代码

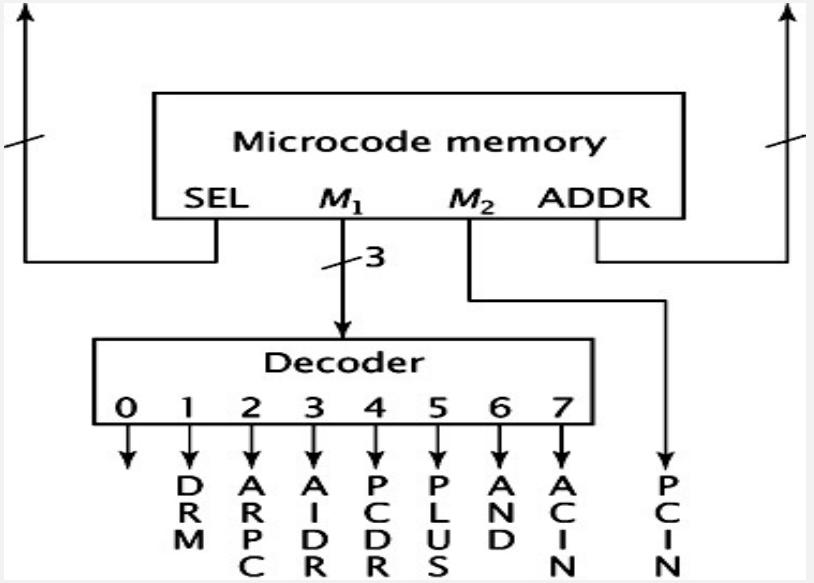
4. 用垂直微代码生成微操作

M1：输入到一个3-8译码器中，例如：输出2 → ARPC

M2：只有1位，不需要用译码器，直接驱动PCIN

状态	地址	SEL	M1	M2	下址
FETCH1	0000 (0)	0	010	0	0001
FETCH2	0001 (1)	0	001	1	0010
FETCH3	0010 (2)	1	011	0	XXXX
ADD1	1000 (8)	0	001	0	1001
ADD2	1001 (9)	0	101	0	0000
AND1	1010 (10)	0	001	0	1011
AND2	1011 (11)	0	110	0	0000
JMP1	1100 (12)	0	100	0	0000
INC1	1110 (14)	0	111	0	0000

控制信号的产生参考水平微代码控制器



VS-CPU的微序列控制器

二、 生成正确序列、设计映射逻辑并设计控制有限状态机状态转换的微代码

5. 从微代码直接产生控制信号

a. 直接输出控制信号：

- 微序列控制器为每个控制信号保留一位，以代替水平微代码设计中的微操作和垂直微代码设计中的 M1 和 M2。
- 对微代码存储器的每个字来说，如果信号有效则相应的控制位置1，否则置0。

示例：**FETCH2：DR←M，PC←PC+1**

1. **DR←M**
   - READ、MEMBUS、DRLOAD信号有效
2. **PC←PC+1**
   - PCINC 信号有效
3. **微指令其它信号置0**

直接产生控制信号与前两种方法相比：

优点：不需要额外的硬件将微代码存储器的输出转换成控制信号

缺点：可读性差，难于调试。

VS-CPU的微序列控制器

二、 生成正确序列、设计映射逻辑并设计控制有限状态机状态转换的微代码

5. 从微代码直接产生控制信号

状态	地址	SEL	ARLOAD	PCLOAD	PCINC	DRLOAD	ACLOAD	ACINC	IRLOAD	ALUSEL	MEMBUS	PCBUS	DRBUS	READ	下址
FETCH1	0000 (0)	0	1	0	0	0	0	0	0	0	0	1	0	0	0001
FETCH2	0001 (1)	0	0	0	1	1	0	0	0	0	1	0	0	1	0010
FETCH3	0010 (2)	1	1	0	0	0	0	0	1	0	0	0	1	0	XXXX
ADD1	1000 (8)	0	0	0	0	1	0	0	0	0	1	0	0	1	1001
ADD2	1001 (9)	0	0	0	0	0	1	0	1	0	0	0	1	0	0000
AND1	1010 (10)	0	0	0	0	1	0	0	0	0	1	0	0	1	1011
AND2	1011 (11)	0	0	0	0	0	1	0	0	1	0	0	1	0	0000
JMP1	1100 (12)	0	0	1	0	0	0	0	0	0	0	0	1	0	0000
INC1	1110 (14)	0	0	0	0	0	0	1	0	0	0	0	0	0	0000

VS-CPU的微序列控制器

二、 生成正确序列、设计映射逻辑并设计控制有限状态机状态转换的微代码

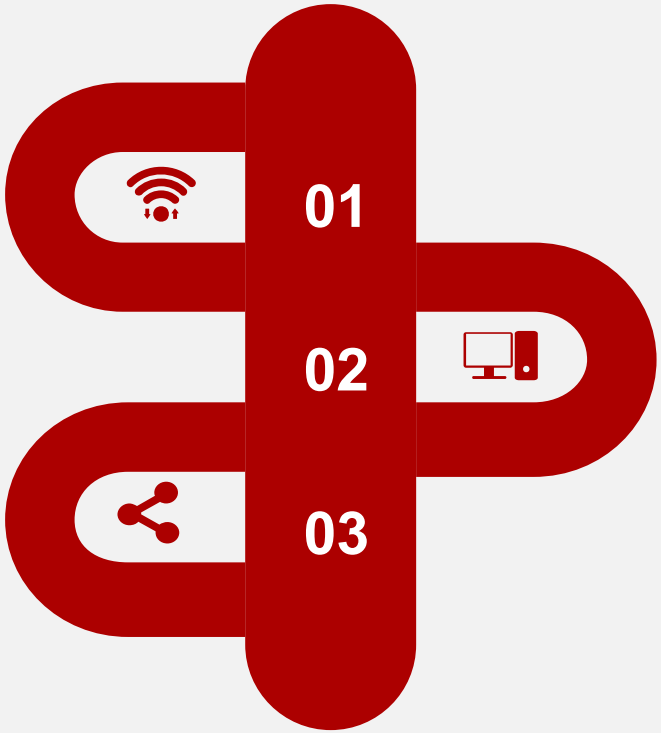
5. 从微代码直接产生控制信号

◆ DRLOAD、MEMBUS和READ信号总有相同的值，把这些信号组合起来。用1位（DMR）驱动这3个信号。

状态	地址	SEL	ARLOAD	PCLOAD	PCINC	DMR	ACLOAD	ACINC	IRLOAD	ALUSEL	PCBUS	DRBUS	下址
FETCH1	0000 (0)	0	1	0	0	0	0	0	0	0	1	0	0001
FETCH2	0001 (1)	0	0	0	1	1	0	0	0	0	0	0	0010
FETCH3	0010 (2)	1	1	0	0	0	0	0	1	0	0	1	XXXX
ADD1	1000 (8)	0	0	0	0	1	0	0	0	0	0	0	1001
ADD2	1001 (9)	0	0	0	0	0	1	0	1	0	0	1	0000
AND1	1010 (10)	0	0	0	0	1	0	0	0	0	0	0	1011
AND2	1011 (11)	0	0	0	0	0	1	0	0	1	0	1	0000
JMP1	1100 (12)	0	0	1	0	0	0	0	0	0	0	1	0000
INC1	1110 (14)	0	0	0	0	0	0	1	0	0	0	0	0000

内容提要

微序列控制器
设计基础



非常简单CPU的
微序列控制器

微程序控制和硬
布线控制的比较

微程序控制和硬布线控制的比较

一、 指令集的复杂度

指令条数的增加导致控制硬件的复杂性增加

- **硬布线控制**：更多状态发出微操作，导致生成信号的组合逻辑增加；
- **水平微代码**：同一个微操作更加频繁地发出，不会改变产生实际控制信号的组合逻辑的复杂性，会增加逻辑的复杂性，因为会有更多的微代码位需要检查；
- **垂直微代码**：控制逻辑的复杂性会增加，组合起来以产生装载信号的微代码位数也会增加。微操作的个数越多，需要的译码器的个数或位数就越多；
- **直接生成**：增加的微操作并不需要增加硬件。不管多少微操作要生成该信号，每个控制信号只对应微代码中的一位。

微程序控制和硬布线控制的比较

二、 修改的容易度

修改的容易度：指CPU设计的可扩展性。

- **硬布线控制**：修改硬件来扩充CPU；
- **微代码**：常常只需改变微程序；

修改微代码比重新设计硬件要容易的多。

三、 时钟速度

- 组合逻辑电路运行速度高于ROM的查表速度；
- 硬布线CPU运行的速度通常比微代码控制CPU的要快。

中国仿真学会会员发展

http://hygl.bitcast.org.cn/cast/reg.jsp?sid=E54

2023年春季学期

下一节：体系结构概述

《计算机系统》课程教学组