计算机系统

-CPU设计之非常简单CPU

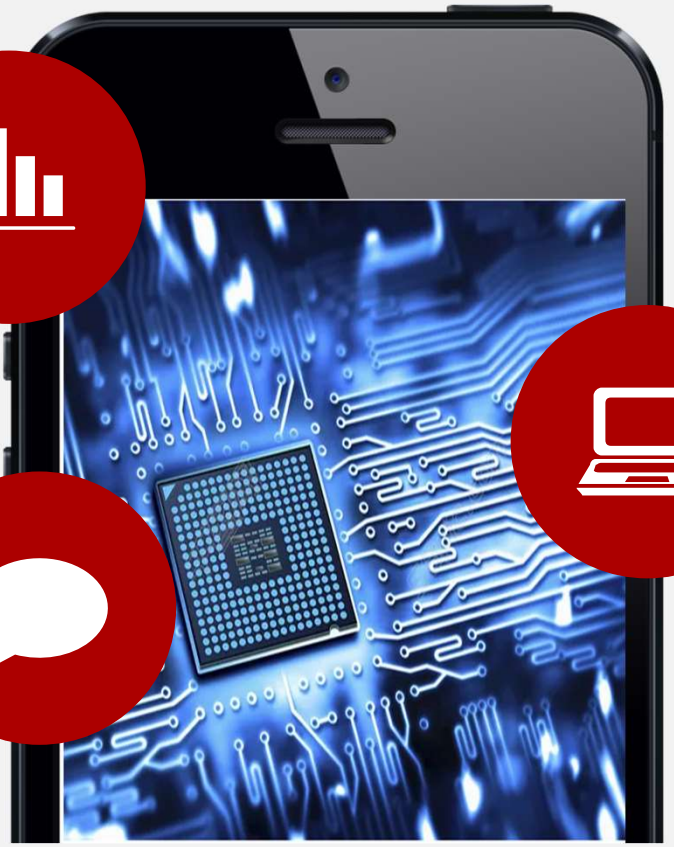
湖南大学

《计算机系统》课程教学组

内容提要

CPU设计规范

简单CPU的不足



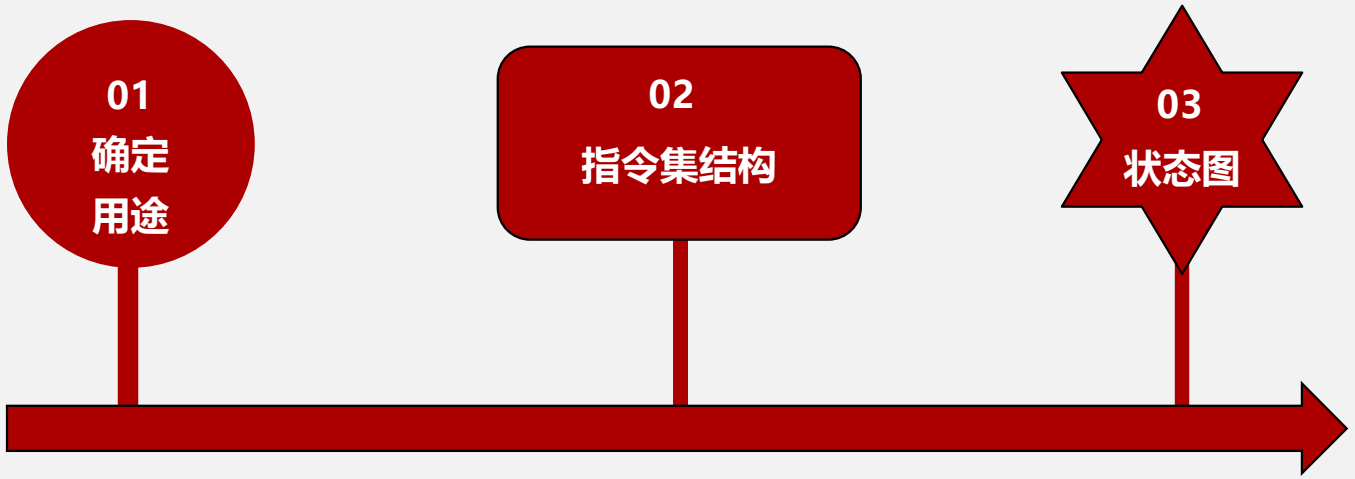
简单CPU的
设计与实现

CPU设计规范

CPU的设计的两种方法：

- **硬布线逻辑控制（hardwired control）**
  - 硬布线控制器是将控制部件做成产生专门固定时序控制信号的逻辑电路，产生各种控制信号，因而又称为组合逻辑控制器。这种逻辑电路以使用最少元件和取得最高操作速度为设计目标，因为该逻辑电路由门电路和触发器构成复杂树型网络，所以称为硬布线控制器。Baidu百科
- **微序列控制器（microsequencer）**
  - 微程序控制信号以微码的形式构成微指令，编成多微指令的微程序，存于存储器中。于是，取出一条微指令就产生一组微操作控制信号，去打开一组控门，控制完成一组微操作。每条机器指令对应一段微程序，这段微程序执毕，该指令规定的功能也就完成了。Baidu百科

CPU设计规范



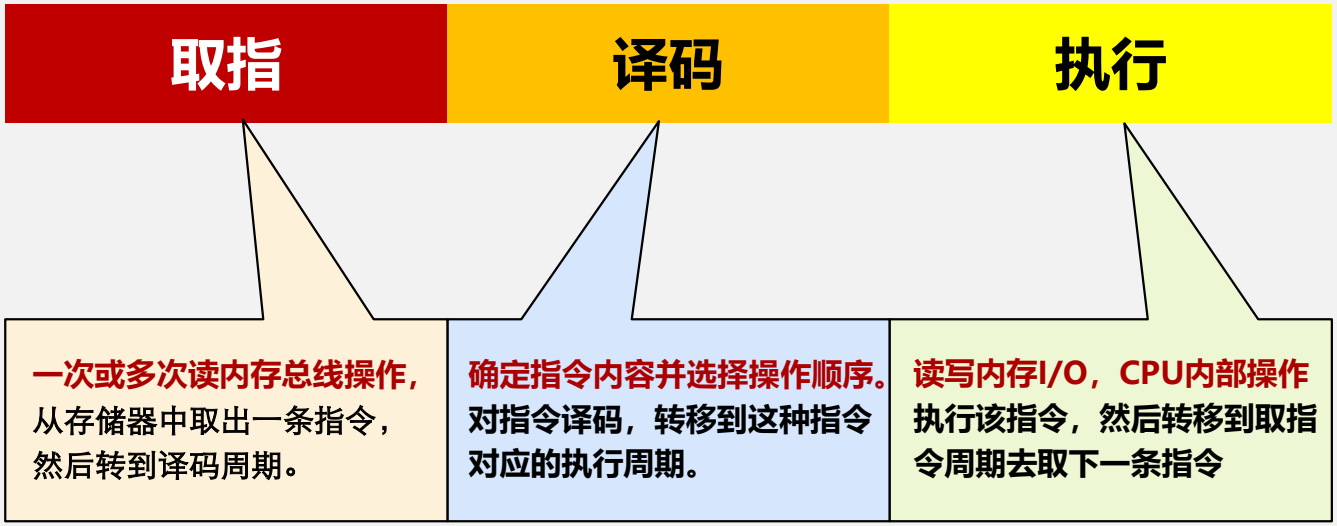
设计CPU的步骤

1. 确定用途： 使CPU的处理能力和它所执行的任务匹配
2. 指令集结构：能够顺利完成所执行的计算任务并且完整而正交的指令集合
3. 状态图：列出在每个状态中执行的微操作以及从一个状态转移到另外一个状态的条件

CPU设计规范

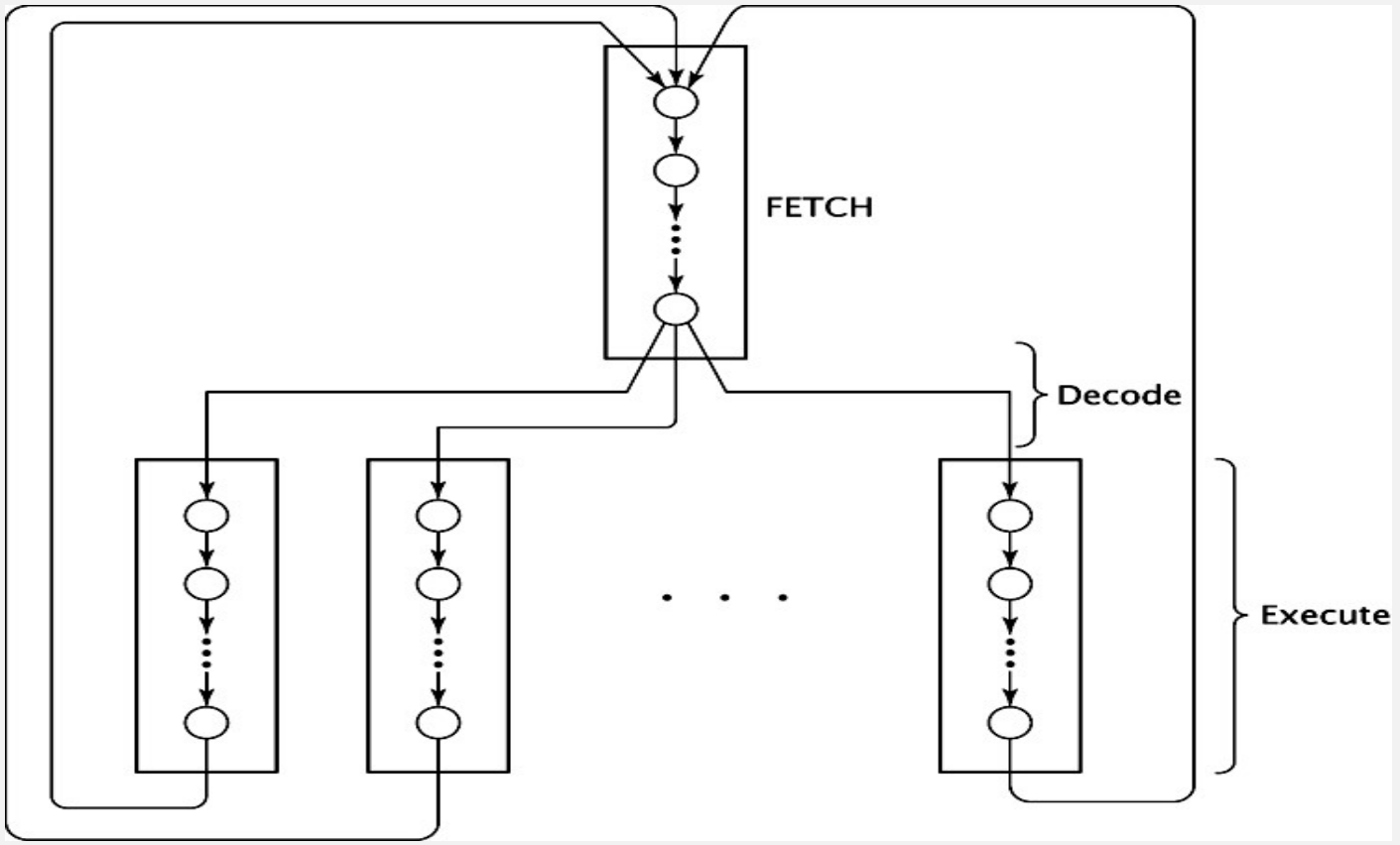
指令周期（Instruction Cycle）是微处理器完成一条指令处理的过程。

包括**取指**（Fetch），**译码**（Decode），**执行**（Execute）三个操作序列。



CPU设计规范

一般CPU状态图

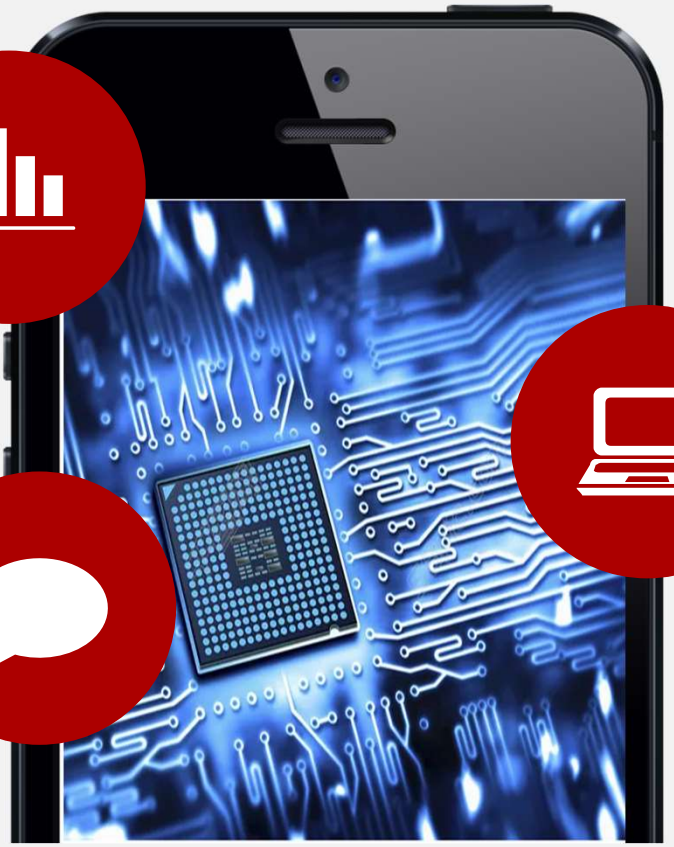


*注意：译码周期不对应任何状态，仅仅是取指结束到各个独立执行阶段的一个多路选择。

内容提要

CPU设计规范

简单CPU的不足



简单CPU的
设计与实现

简单CPU设计实例

存储空间

64字节的存储空间
每个字节是8位。

6位宽的地址：A[5..0]

存储器的8位值：D[7..0]

寄存器

仅有一个程序员可以访问的寄存器：

累加器：AC[7..0]

指令集

ADD：AC←AC+M

AND：AC←AC^M

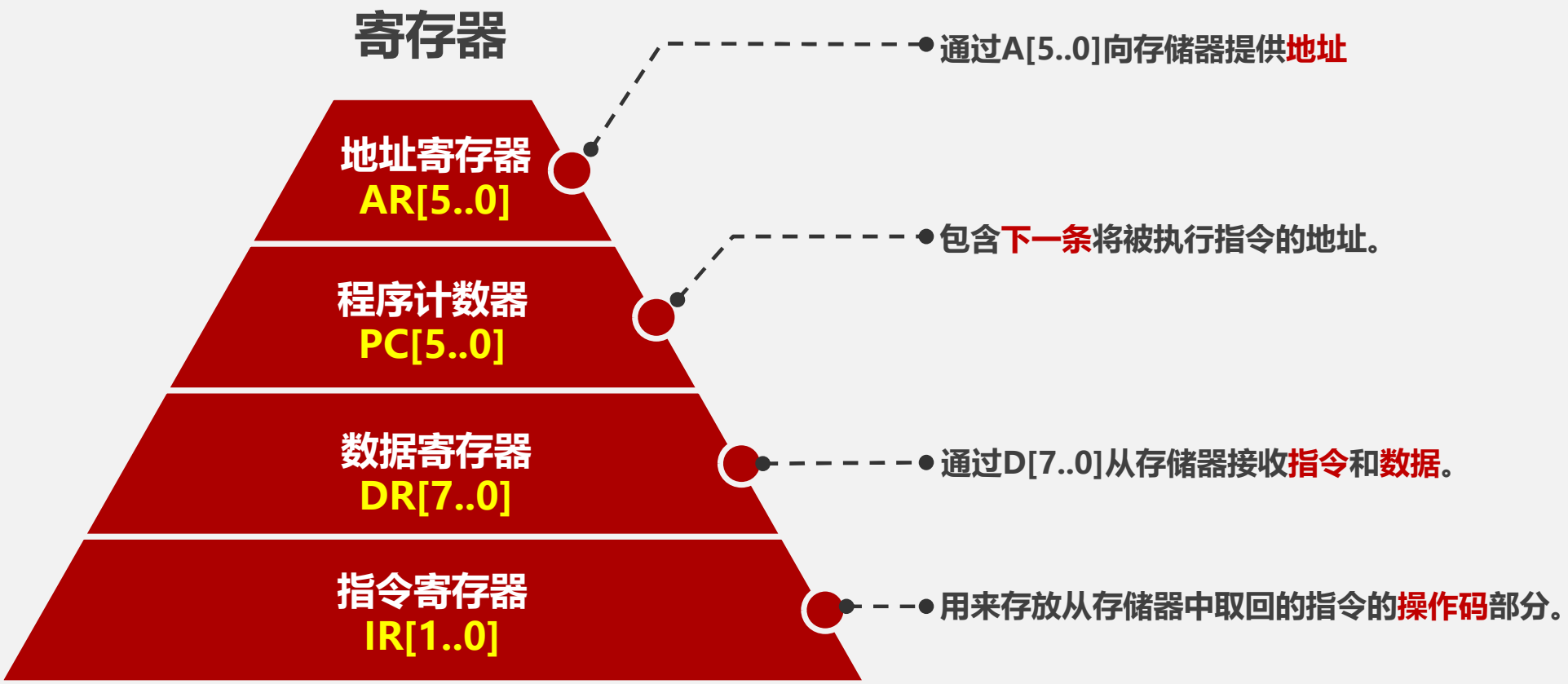
JMP：GOTO M

INC ：AC←AC+1

非常简单CPU设计实例

指令	指令码	操作
ADD	00AAAAAA	AC←AC+M[AAAAAA]
AND	01AAAAAA	AC←AC^M[AAAAAA]
JMP	10AAAAAA	GOTO AAAAAA (PC = AAAAAA)
INC	11XXXXXX	AC←AC+1

简单CPU设计实例

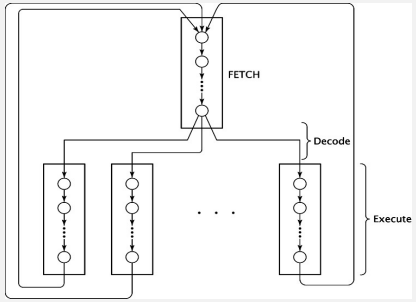

寄存器
地址寄存器
AR[5..0]
程序计数器
PC[5..0]
数据寄存器
DR[7..0]
指令寄存器
IR[1..0]
通过A[5..0]向存储器提供地址
包含下一条将被执行指令的地址。
通过D[7..0]从存储器接收指令和数据。
用来存放从存储器中取回的指令的操作码部分。

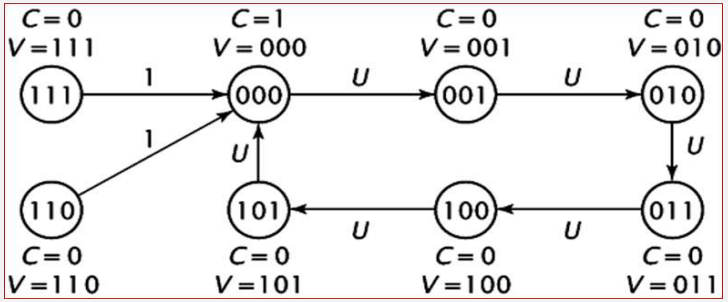
简单CPU设计实例

CPU仅仅就是一个复杂的有限状态机。

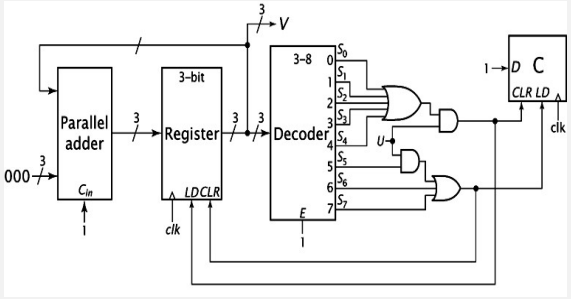
设计CPU的途径

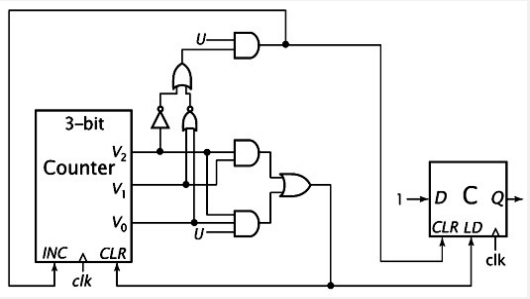
1. 设计CPU的状态图。





2. 设计必要的数据通路和控制逻辑，以便实现这个有限状态机，最终实现这个 CPU。





简单CPU设计实例

取指周期的操作序列

PC 指令计数器

1. 将地址放在地址引脚 **A[5..0]**上，通过地址总线送给存储器。

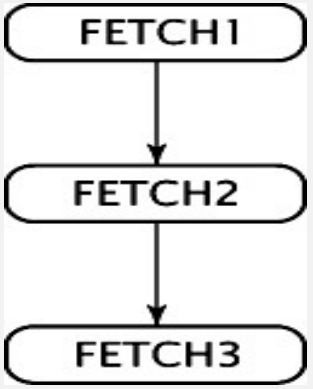
   FETCH1: AR[5..0]←PC[5..0]

2. 存储器内部译码并将需要的指令取出放到数据引脚**D[7..0]**经数据总线送到**DR[7..0]**。

   FETCH2: DR[7..0]←M[7..0]，PC←PC＋1

3. 指令的操作码 **DR[7..6]→IR[1..0]**；指令地址码 **DR[5..0]→AR[5..0]**。

   FETCH3: IR[1..0]←DR[7..6]，AR[5..0]←DR[5..0]



*** 为什么在 FETCH 2 就将 PC 加1?**

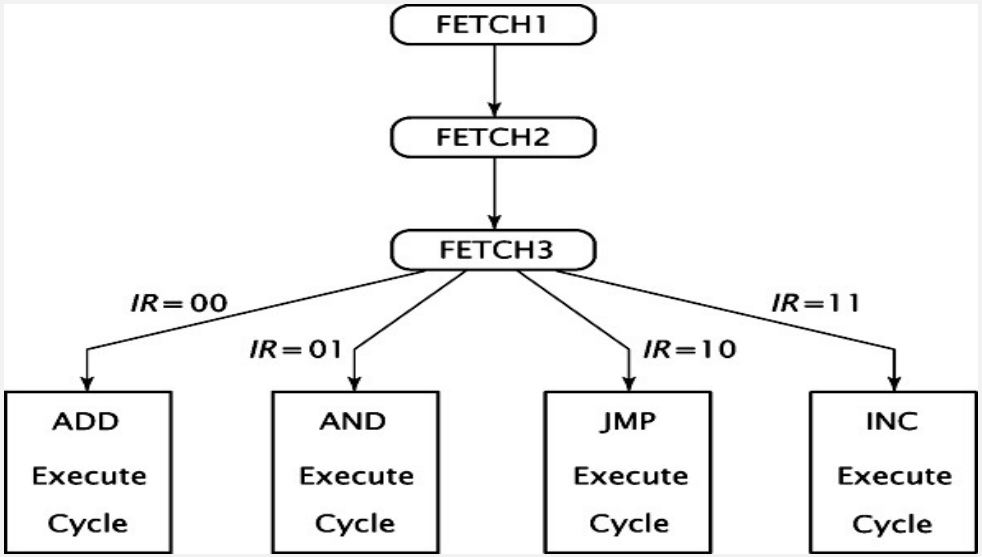
简单CPU设计实例

指令译码的操作序列

1. 判断所取的是何种指令，从而调用对应的执行周期。

2. 在状态图中，此过程表示为一系列的从取指令周期结束到各个执行周期之间的分支。

3. 对于本CPU，有四条指令，因此有四个不同的执行周期。



指令	指令码
ADD	00AAAAAA
AND	01AAAAAA
JMP	10AAAAAA
INC	11XXXXXX

简单CPU设计实例

指令执行的操作序列

AC为寄存器

1. ADD 指令　（1）从存储器中取出一个数；（2）将其与**AC**相加，结果保存在**AC**。

ADD1 : DR←M

ADD2 : AC←AC+DR

2. AND 指令　（1）从存储器取出一个数；（2）将其与**AC**相与，结果保存在**AC**。

AND1 : DR←M

AND2 : AC←AC^DR

3. JMP 指令　（1）从**DR**或**AR**中获取地址码送给**PC**。

PC←DR[5..0] 数据寄存器的5到0位

or

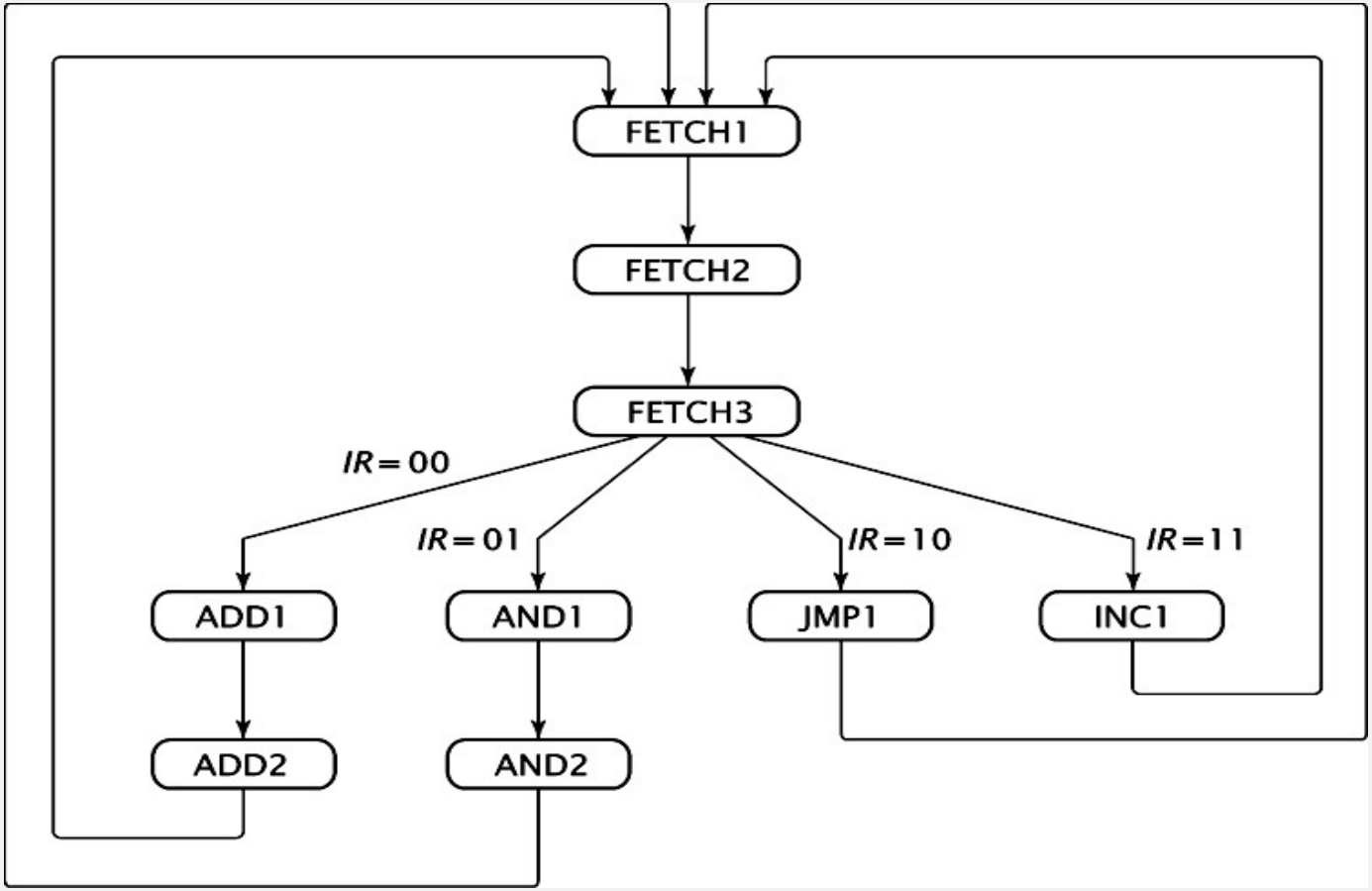
PC←AR[5..0]

4. INC 指令　（1）将**AC**中的值加1，和存入**AC**。

AC←AC+1

简单CPU设计实例

完整的简单CPU状态图



简单CPU设计实例

建立所需的数据通路

一、依据需要传送的数据而设计内部数据通路，确定与CPU状态相关的操作。

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

建立所需的数据通路

二、设计数据通路的两种不同方案

1. 在所有需要传送数据的部件之间创建一条直接通路。使用多路选择器或者缓冲器为那些有多个数据源的寄存器从多个可能的输入中选择一个。随着CPU复杂度的增加，这种方案将变得不现实。

2. 在CPU的内部创建一条总线，在各个部件之间使用总线传递数据（总线的优势）。

建立所需的数据通路

三、将操作重新分组

1. 依据操作所修改的寄存器进行操作分组如下：

AR ： AR←PC；AR←DR[5..0]

PC ： PC←PC＋1；PC←DR[5..0]

DR ： DR←M

IR ： IR←DR[7..6]

AC ： AC←AC＋DR；AC←AC∧DR；AC←AC＋1

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

简单CPU设计实例

建立所需的数据通路

四、对每一个操作进行分析从而决定每个部件应该完成的功能

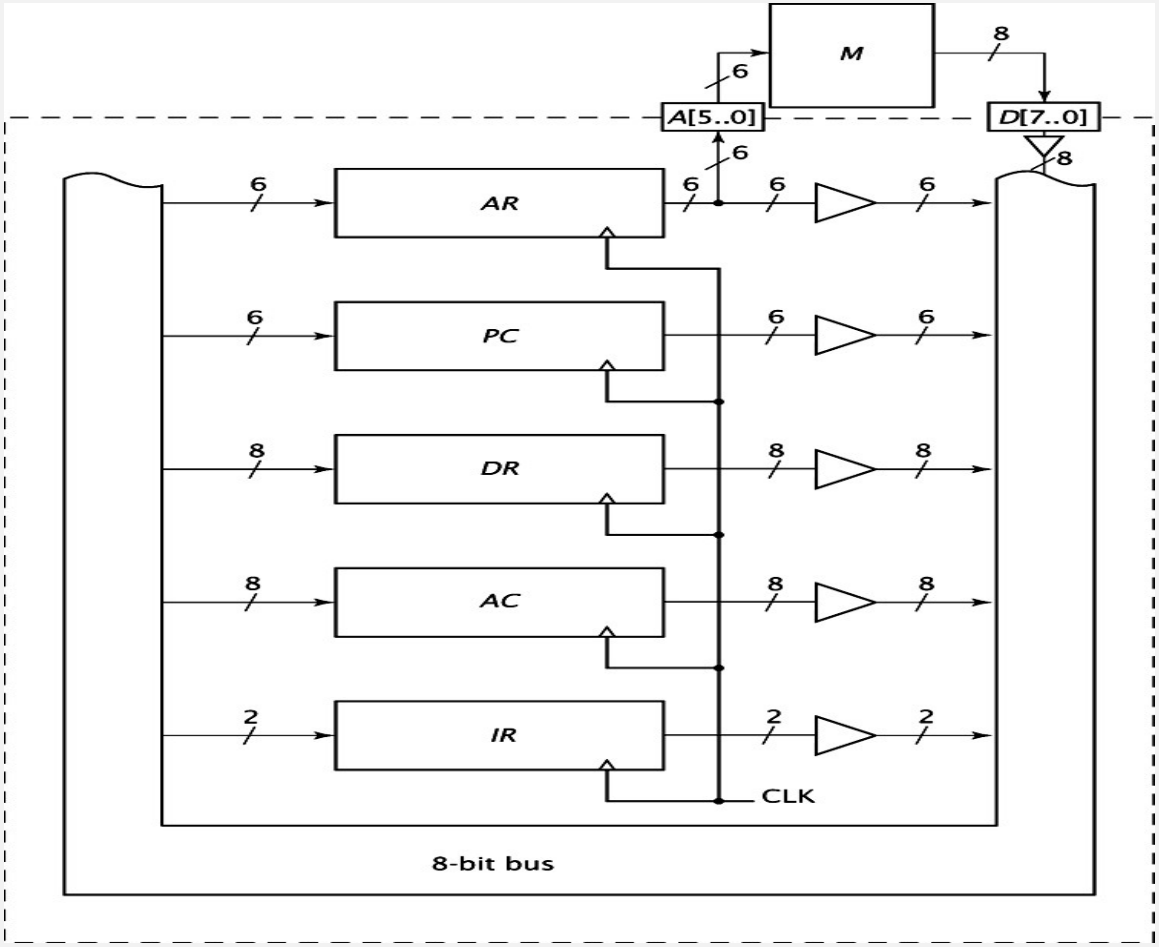
1. AR，DR以及IR总是从其它一些部件中装入数据且是并行的装入操作；

2. PC和AC能够从其它部件装入数据，但它们还要实现自增1，需要一个单独的自增硬件，并且使结果能够重新装入寄存器；可以将它们都设计成计数器并且能够并行装载。

简单CPU设计实例

建立所需的数据通路

五、将所有部件连接到系统总线上



AR ： AR←PC；AR←DR[5..0]

PC ： PC←PC＋1；PC←DR[5..0]

DR ： DR←M

IR ： IR←DR[7..6]

AC ： AC←AC＋DR；AC←AC∧DR；AC←AC＋1

简单CPU设计实例

建立所需的数据通路

六、修改设计

仔细分析连接图中各个部件的数据走向：

1. **AR**仅仅向存储器提供数据，除此之外不跟任何部件传送数据。因此，没有必要将它的输出连接到内部总线上。

2. **IR**不通过内部总线向任何其他部件提供数据，所以IR 的输出到内部总线的连接可以删除。

简单CPU设计实例

建立所需的数据通路

六、修改设计

仔细分析连接图中各个部件的数据走向：

3. **AC**不向其他任何单元提供数据；因此与内部总线的连接也可以删除。

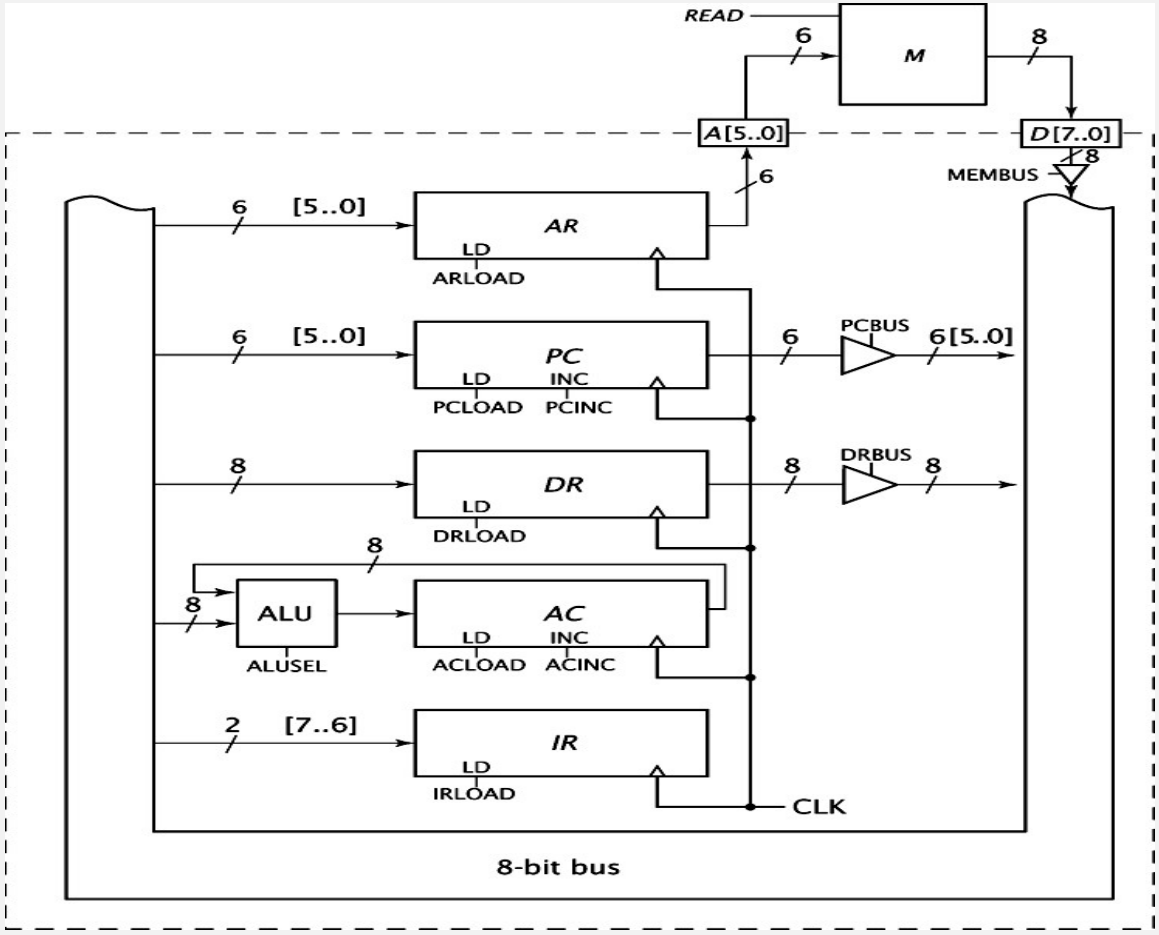
4. 总线是**8位**宽，但是并非所有被传送的数据都是8位宽；有一些是**6位**宽，有一个是**2位**宽。必须确定哪些寄存器从总线的哪些位上接收和发送数据。

5. **AC**必须能够装载**AC**和**DR**的**和**以及它们的**逻辑与**的结果。CPU必须包含一个能够产生这些结果 的**ALU**。

简单CPU设计实例

建立所需的数据通路

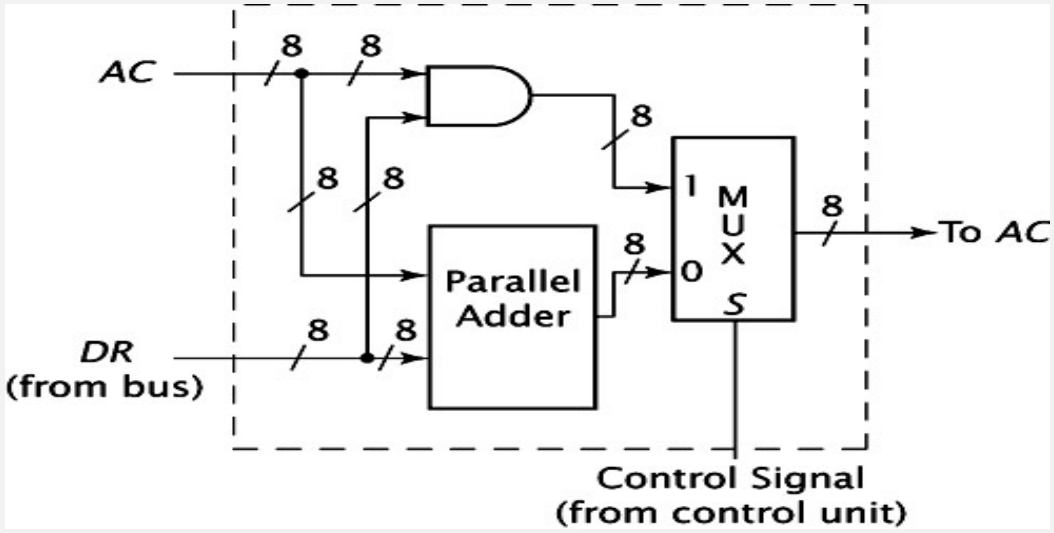
七、修改后的CPU内部组织结构图（其中的控制信号将有控制单元产生）



简单CPU设计实例

简单ALU的设计（Arithmetic Logic Unit）

1. ALU完成两个功能: a. 将两个输入相加; b. 将两个输入相与
2. 设计方法
   a. 创建两个单独的硬件来实现每个功能，
   b. 使用一个多路选择器从两个结果中选择一个输出。



简单CPU设计实例

用硬布线逻辑设计控制单元

控制单元：产生控制信号，从而使所有的操作能以正确的顺序执行。

设计控制单元有两种主要的方法：

- **硬布线控制：**使用时序逻辑和组合逻辑产生控制信号。
- **微程序控制/微序列控制：**使用存储器查表方式来输出控制信号。

本节重点：硬布线控制方法

简单CPU设计实例

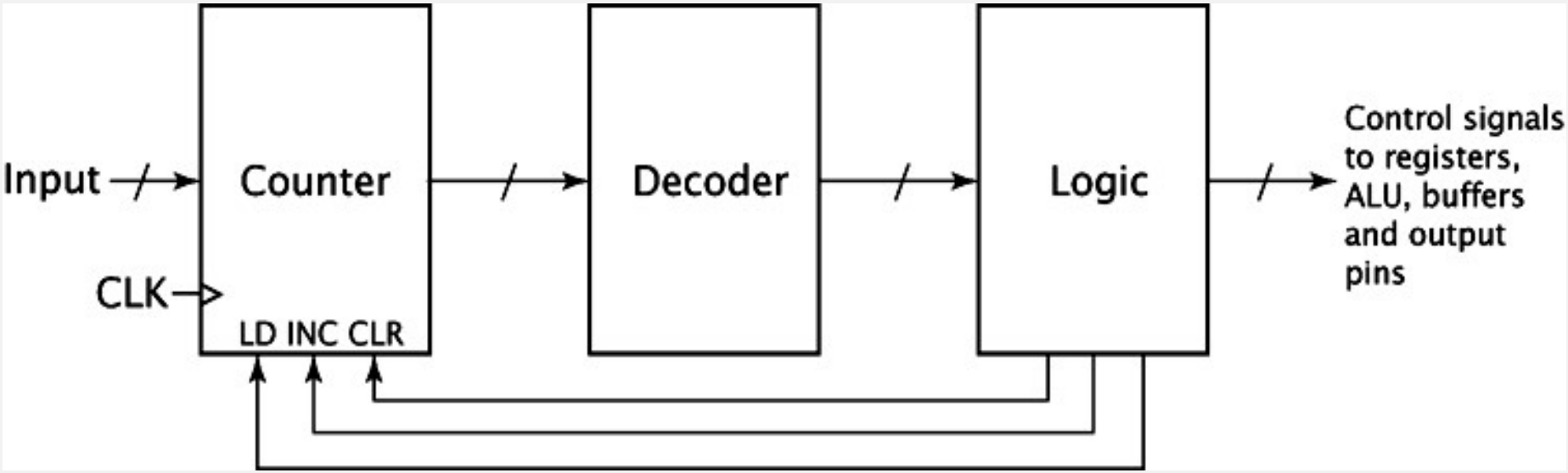
用硬布线逻辑设计控制单元

简单控制单元的组成

- **计数器**：保存当前状态
- **译码器**：接受当前状态并为每个状态生成单独的信号
- **组合逻辑**：
  接收单独的状态信号为每一部件生成控制信号，以及计数器的控制信号。

简单CPU设计实例

一般硬布线控制单元



对于本CPU，总共有**9个**状态（**FETCH1~3, ADD1~2, AND1~2, JMP, INC**）

- 需要一个4位计数器
- 一个4→16位译码器（译码器的输出位中有7个用不到）

简单CPU设计实例

实现控制单元功能

一、将状态分配到译码器各个输出的原则

1. 将**FETCH1**规定为计数器的**0**值，并使用计数器的**CLR**输入来达到这个状态。

2. 将顺序的状态指派为计数器的连续值，并且使用计数器的**INC**输入来遍历所有的这些状态。

   FETCH2：计数器值1；**FETCH3**：计数器值2

   同样，将**ADD1**和**ADD2**指派为连续的计数值；对**AND1**和**AND2**也是一样。

3. 根据指令操作码和执行周期的最大状态数量来指派每个执行周期的第一个状态，由操作码产生计数器输入值，由LD信号使之达到正确的执行周期。

简单CPU设计实例

实现控制单元功能

二、 为了装入正确执行周期的地址，控制单元必须完成两件事情。

1. 必须能够将正确的执行周期的第一个状态的地址放到计数器的输入上。

IR中的操作码是2位，表示状态的计数器值Q是4位。IR和Q之间应该建立一个函数关系，实现Q对各个状态的正确分配。所有指令和他们的初态及对应IR值如**左表**。

假设我们采用 **Q=10IR[1..0]**的方式（回想《指令集结构》中4位指令前面的0），得到**右表**中的结果。

*右表中的结果显然是不理想的（ADD1与AND1是连续值，如何使ADD1与ADD2连续加载？AND同理。）

指令	初态	IR
ADD	ADD1	00
AND	AND1	01
JMP	JMP1	10
INC	INC1	11

初态	IR	计数值
ADD1	00	1000 (8)
AND1	01	1001 (9)
JMP1	10	1010 (10)
INC1	11	1011 (11)

实现控制单元功能

二、 为了装入正确执行周期的地址，控制单元必须完成两件事情。

1. 必须能够将正确的执行周期的第一个状态的地址放到计数器的输入上。

我们只需要让这4个指令初态的状态值拉开一个距离，就能让需要连续加载的两个状态获得连续的状态值。

因此我们采用 **Q=1IR[1..0]0** 的方式来实现这个目标。

指令	初态	IR
ADD	ADD1	00
AND	AND1	01
JMP	JMP1	10
INC	INC1	11

初态	IR	计数值
ADD1	00	1000 (8)
AND1	01	1010 (10)
JMP1	10	1100 (12)
INC1	11	1110 (14)

ADD2←1001 (9)

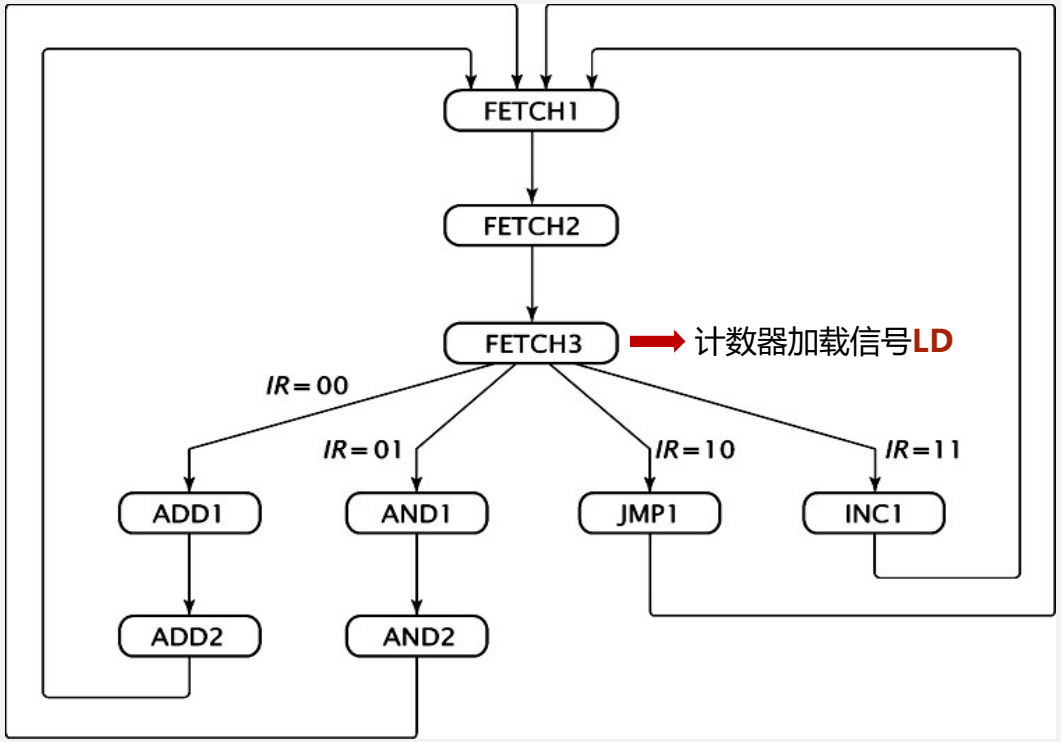
AND2←1011 (11)

实现控制单元功能

二、 为了装入正确执行周期的地址，控制单元必须完成两件事情。

2. 必须发出计数器的LD信号。

控制单元在取指周期末尾**FETCH3**状态发出**LD**信号，才能使得当前**执行周期**对应的状态被加载到计数器中。



简单CPU设计实例

实现控制单元功能

三、为计数器以及CPU的其它部分产生控制信号。

对于计数器，我们必须产生INC，CLR和LD信号。

当控制单元遍历顺序状态（FETCH1，FETCH2，ADD1以及AND1）时，INC信号有效。

01 INC 信号

02 CLR 信号

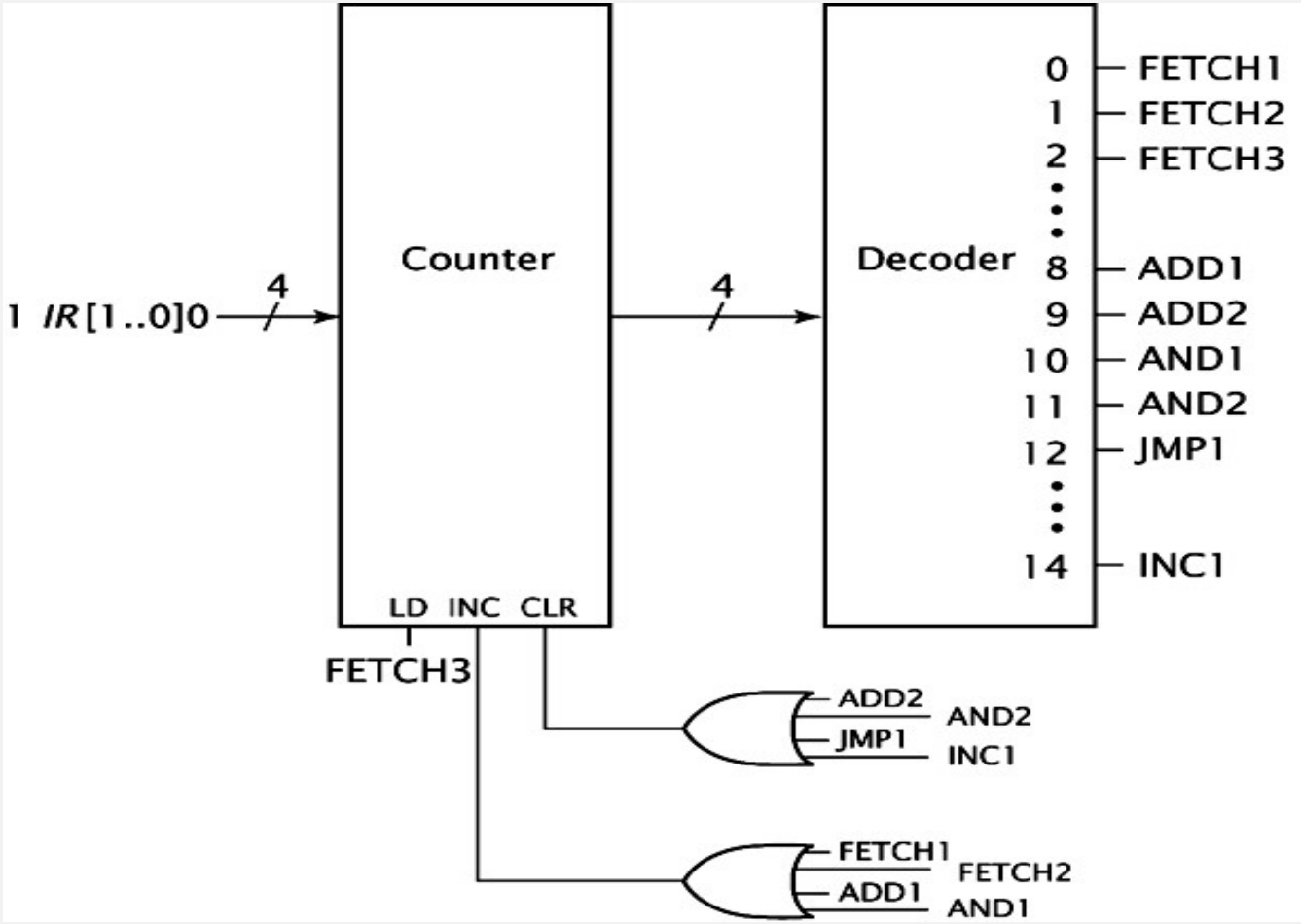
CLR则用来从每一个执行周期的末尾返回到取指令周期，这可能发生在ADD2，AND2，JMP1和INC1状态。

LD信号在每个取指令周期的末尾FETCH3状态中发出

03 LD 信号

简单CPU设计实例

四、非常简单CPU的硬布线控制单元



五、组合形成控制AR，PC，DR，IR，M，ALU以及缓冲器的控制信号

1. AR寄存器的控制信号

AR在状态 **FETCH1 (AR←PC)** 和 **FETCH3 (AR←DR[5..0])** 期间进行装载。

通过将这两个状态信号进行逻辑**OR**操作，CPU就为AR产生了**LD**信号(ARLOAD=FETCH1 v FETCH3)。

2. PC寄存器的控制信号

在发生跳转时，**PC** 需要加载跳转目的地的地址 (PCLOAD=JMP1)。

在取指周期第二个状态，**PC** 需要自增**1** (PCINC=FETCH2)。

五、组合形成控制AR，PC，DR，IR，M，ALU以及缓冲器的控制信号

3. DR寄存器的控制信号

DR在FETCH2、ADD1以及AND1状态时需要加载数据 (DRLOAD=FETCH2 v ADD1 v AND1)。

4. IR寄存器的控制信号

IR是在FETCH3状态下进行指令操作码的加载 (IRLOAD=FETCH3)；

五、组合形成控制AR，PC，DR，IR，M，ALU以及缓冲器的控制信号

5. AC累加器的控制信号

AC在ADD2或AND2时需要保存计算结果 (ACLOAD=ADD2 v AND2)。

AC在INC1时进行自增1 (ACINC=INC1)。

6. ALU的输出选择信号

ALU中是一个并行加法器和一个与门，需要一个选择信号决定哪一个作为输出

(ALUSEL=0, 输出算术和；ALUSEL=1, 输出逻辑与)；

把ALUSEL设置为AND2，就能保证当CPU执行ADD或AND指令时，有正确的结果从ALU流向AC。

五、组合形成控制AR，PC，DR，IR，M，ALU以及缓冲器的控制信号

7. DR缓冲器的有效信号

在下列状态下，DR的内容必须放到总线上，将这些状态值进行逻辑或，就能够得到DRBUS信号。

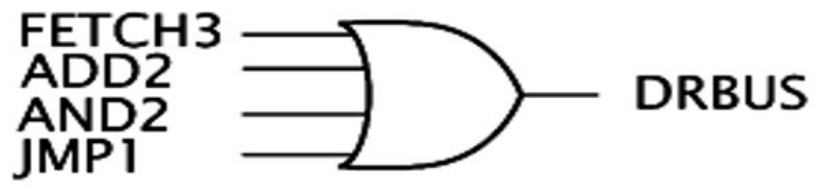
FETCH3 (IR←DR[7..6]，AR←DR[5..0])

ADD2 (AC←AC+DR)

AND2 (AC←AC∧DR)

JMP1 (PC←DR[5..0])

DRBUS=FETCH3 v ADD2 v AND2 v JMP1



简单CPU设计实例

五、组合形成控制AR，PC，DR，IR，M，ALU以及缓冲器的控制信号

7. 内存M缓冲器的有效信号

在FETCH2、ADD1以及AND1状态下，内存提供指令内容或运算数据到数据总线上。

MEMBUS=FETCH2 v ADD1 v AND2

8. PC缓冲器的有效信号

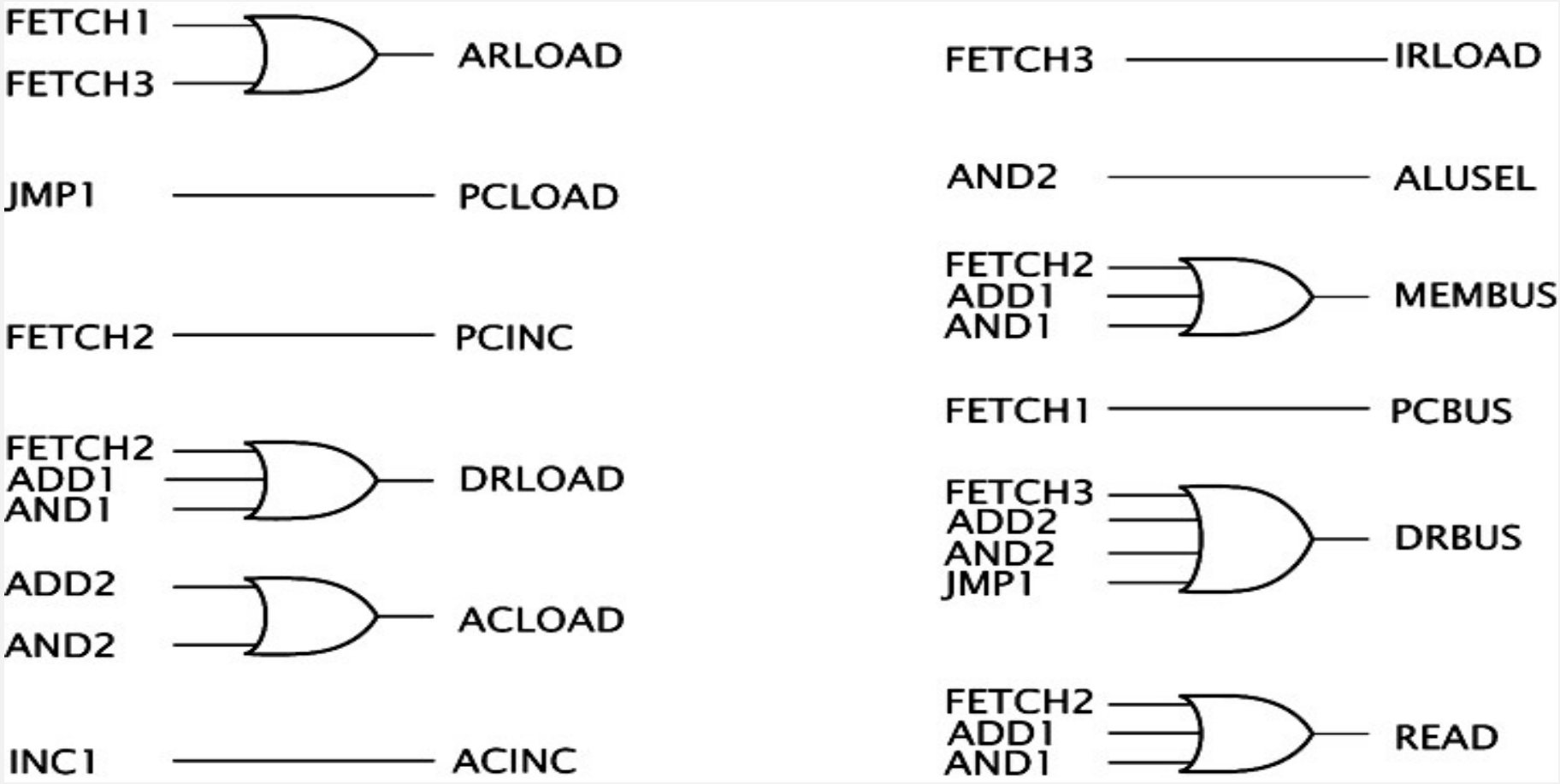
在FETCH1状态下，PC需要将下一条指令地址放到地址总线上 (PCBUS=FETCH1)。

9. 内存读取信号READ

在FETCH2、ADD1以及AND1状态下，CPU发出READ信号 (READ=FETCH2 v ADD1 v AND1)。

简单CPU设计实例

五、组合形成控制AR，PC，DR，IR，M，ALU以及缓冲器的控制信号



设计验证

对每条指令的取指令、译码以及执行周期进行跟踪。

1.考虑如下这段代码，它每条指令仅仅包含了一次。

0: ADD 4	AC←AC+M[4]
1: AND 5	AC←AC^M[5]
2: INC	AC←AC+1
3: JMP 0	
4: 27H	
5: 39H	

简单CPU设计实例

设计验证

对每条指令的取指令、译码以及执行周期进行跟踪。

2. CPU遵循状态图并以合适的状态顺序取出、译码和执行每条指令：

ADD4 : FETCH1→FETCH2→FETCH3→ADD1→ADD2

AND5 : FETCH1→FETCH2→FETCH3→AND1→AND2

INC : FETCH1→FETCH2→FETCH3→INC1

JMP 0 : FETCH1→FETCH2→FETCH3→JMP1

简单CPU设计实例

设计验证

对每条指令的取指令、译码以及执行周期进行跟踪。

3. 对这段程序的一次循环的跟踪情况（所有寄存器的初始值都是0）

指令	状态	有效信号	执行操作	下一状态
ADD4	FETCH1	PCBUS, ARLOAD	AR←0	FETCH2
	FETCH2	READ, MEMBUS DRLOAD, PCINC	DR←04H, PC←1	FETCH3
	FETCH3	DRBUS, ARLOAD IRLOAD	IR←00, AR←04H	ADD1
	ADD1	READ, MEMBUS DRLOAD	DR←27H	ADD2
	ADD2	DRBUS, ACLOAD	AC←0+27H	FETCH1

0: ADD 4　AC←AC+M[4]

1: AND 5　AC←AC^M[5]

2: INC　AC←AC+1

3: JMP 0

4: 27H

5: 39H

简单CPU设计实例

指令	状态	有效信号	执行操作	下一状态
AND5	FETCH1	PCBUS, ARLOAD	AR←1	FETCH2
	FETCH2	READ, MEMBUS, DRLOAD, PCINC	DR←45H, PC←2	FETCH3
	FETCH3	DRBUS, ARLOAD IRLOAD	IR←01, AR←05H	AND1
	AND1	READ, MEMBUS DRLOAD	DR←39H	AND2
	AND2	DRBUS, ALUSEL, ACLOAD	AC←27H^39H=21H	FETCH1

0: ADD 4	AC←AC+M[4]
1: AND 5	AC←AC^M[5]
2: INC	AC←AC+1
3: JMP 0	
4: 27H	
5: 39H	

简单CPU设计实例

指令	状态	有效信号	执行操作	下一状态
INC	FETCH1	PCBUS, ARLOAD	AR←2	FETCH2
	FETCH2	READ, MEMBUS, DRLOAD, PCINC	DR←C0H,PC←3	FETCH3
	FETCH3	DRBUS, ARLOAD IRLOAD	IR←11, AR←00H	INC1
	INC1	ACINC	AC←21H+1=22H	FETCH1

0: ADD 4　　AC←AC+M[4]

1: AND 5　　AC←AC^M[5]

2: INC　　AC←AC+1

3: JMP 0

4: 27H

5: 39H

简单CPU设计实例

指令	状态	有效信号	执行操作	下一状态
JMP0	FETCH1	PCBUS, ARLOAD	AR←3	FETCH2
	FETCH2	READ, MEMBUS, DRLOAD, PCINC	DR←80H,PC←4	FETCH3
	FETCH3	DRBUS, ARLOAD IRLOAD	IR←10, AR←00H	JMP1
	JMP1	DRBUS, PCLOAD	PC←0	FETCH1

0: ADD 4　　AC←AC+M[4]

1: AND 5　　AC←AC^M[5]

2: INC　　AC←AC+1

3: JMP 0

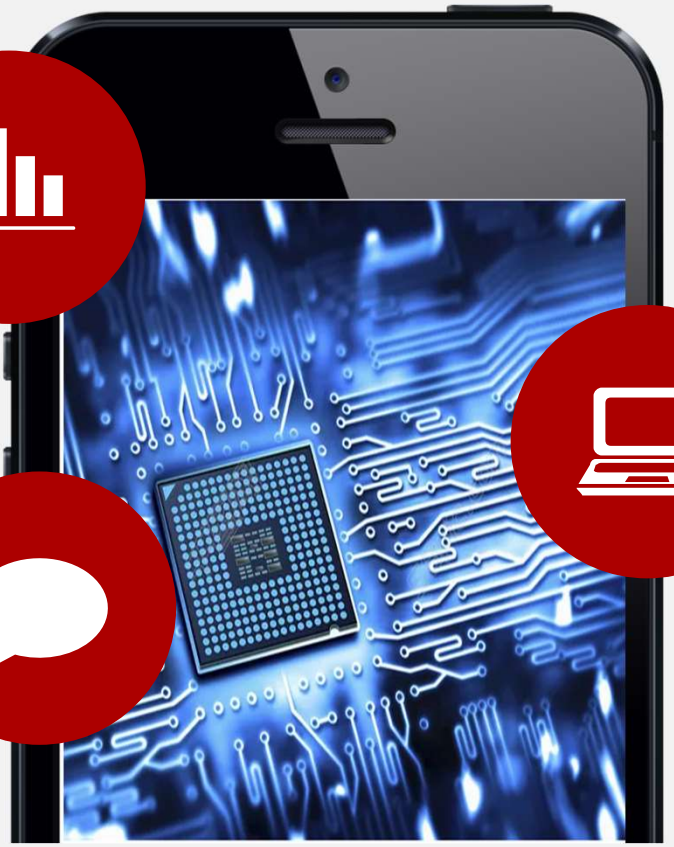
4: 27H

5: 39H

内容提要

CPU设计规范

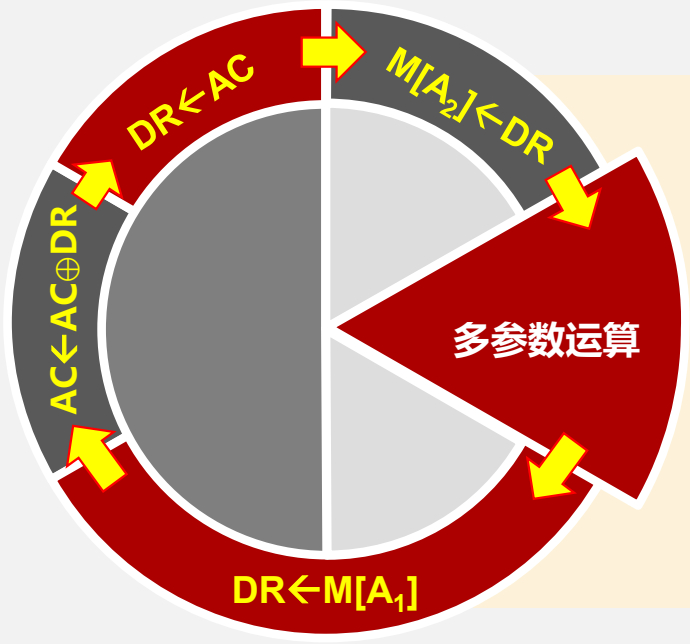
简单CPU的不足



简单CPU的
设计与实现

非常简单CPU的缺点

1. 只有一个可访问的寄存器AC



- 有限的寄存器给复杂的运算带来巨大的困难，极大地增加了内存访问的开销和对总线的占用，计算时间也大大增加。
- 提高微处理器性能的最好方法之一：在CPU的内部提供更多的存储空间。使一些外部的存储访问用更快速的内部访问代替。改善那些具有子程序的程序的性能。

- Intel的第一个微处理器4004没有任何通用寄存器。
- 8008，8080和8085，在芯片的内部包含有6个通用寄存器，以及1个累加器。8086微处理器有8个通用寄存器，80286，80386，80486也都一样。
- Pentium微处理器同样有8个内部通用寄存器，但是它们是32位宽，而不象它的祖先是16位宽。
- Itaium微处理器包含了超过4MB的三级高速缓冲存储器。拥有128个通用寄存器和额外的128个通用浮点寄存器。

非常简单CPU的不足

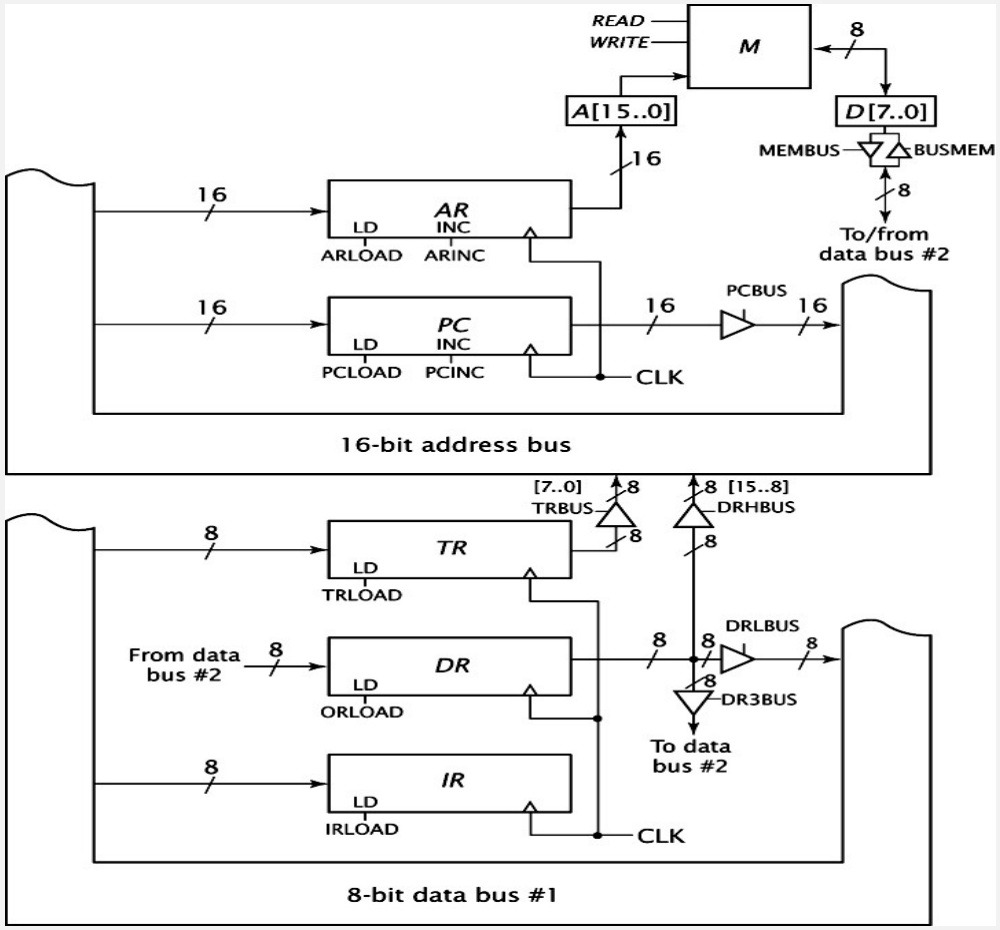
2. 只有一条总线 😔

01

地址、数据都依赖这一条总线传输。当出现大量数据参与的运算，总线将不堪重负，由此大幅降低机器运行的效率。

02

多总线允许多个数据同时传送，减少取指、译码以及执行指令的时间，减少各部件之间的直接连接，提高系统性能。



非常简单CPU的不足

3. 没有流水线 😔

- 简单CPU没有流水线，指令的取指、译码、执行总是在下一条指令被处理之前发生。
- 流水线的引入使得第一条指令周期的中间状态允许下一条指令进入到新的周期。重叠的执行可以让程序执行得更快（尽管每条指令的执行时间没有变化）。

非常简单CPU的不足

4. 有限的指令集 😔

- 指令数量更多的指令集可以是程序使用更少的指令而完成同样的功能。

- 考虑只能进行逻辑与和取反运算的CPU，为了对A和B进行逻辑或运算，它不得不执行下面的指令：

运算	操作
取A的反X	X←NOT A
取B的反Y	Y←NOT B
X和Y做逻辑与	Z←AND X, Y
将结果取反	NOT Z

* 如果指令集中本身就有OR指令，则一条指令就可以完成。

* NEG A = NOT A +1 也是一个例子。

非常简单CPU的不足

5. 没有子程序与中断处理方式 😔

- 几乎左右的CPU都提供了处理子程序的**堆栈指针**以及**调用子程序**和**从子程序返回**的指令。

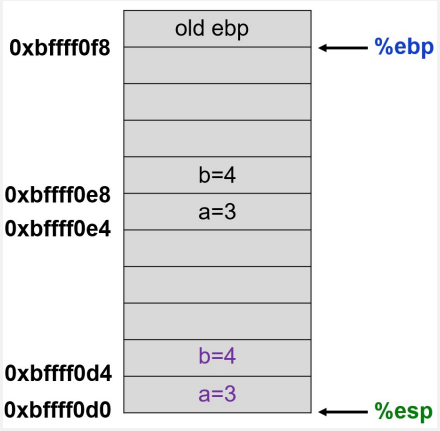
%ebp

%esp

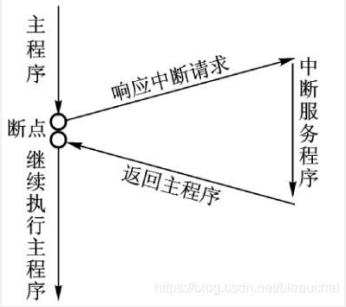
call

leave

ret



- 大多数CPU都具备**中断输入**，允许外部硬件中断CPU当前正在执行的指令。



https://blog.csdn.net/bleauchat

内容提要

进阶内容
相对简单CPU设计

- 指令集结构
- 寄存器内容
- 取指操作序列
- 执行操作序列

相对简单CPU设计

存储空间

64KB的存储空间
每个字节是8位。

16位宽的地址：A[15..0]

存储器的8位值：D[7..0]

2^{16}

寄存器

可访问的寄存器：
累加器：AC[7..0]
通用　：R[7..0]
零标志：Z

指令集

16条指令
指令码：8 bit
数据传送
程序控制
数据运算

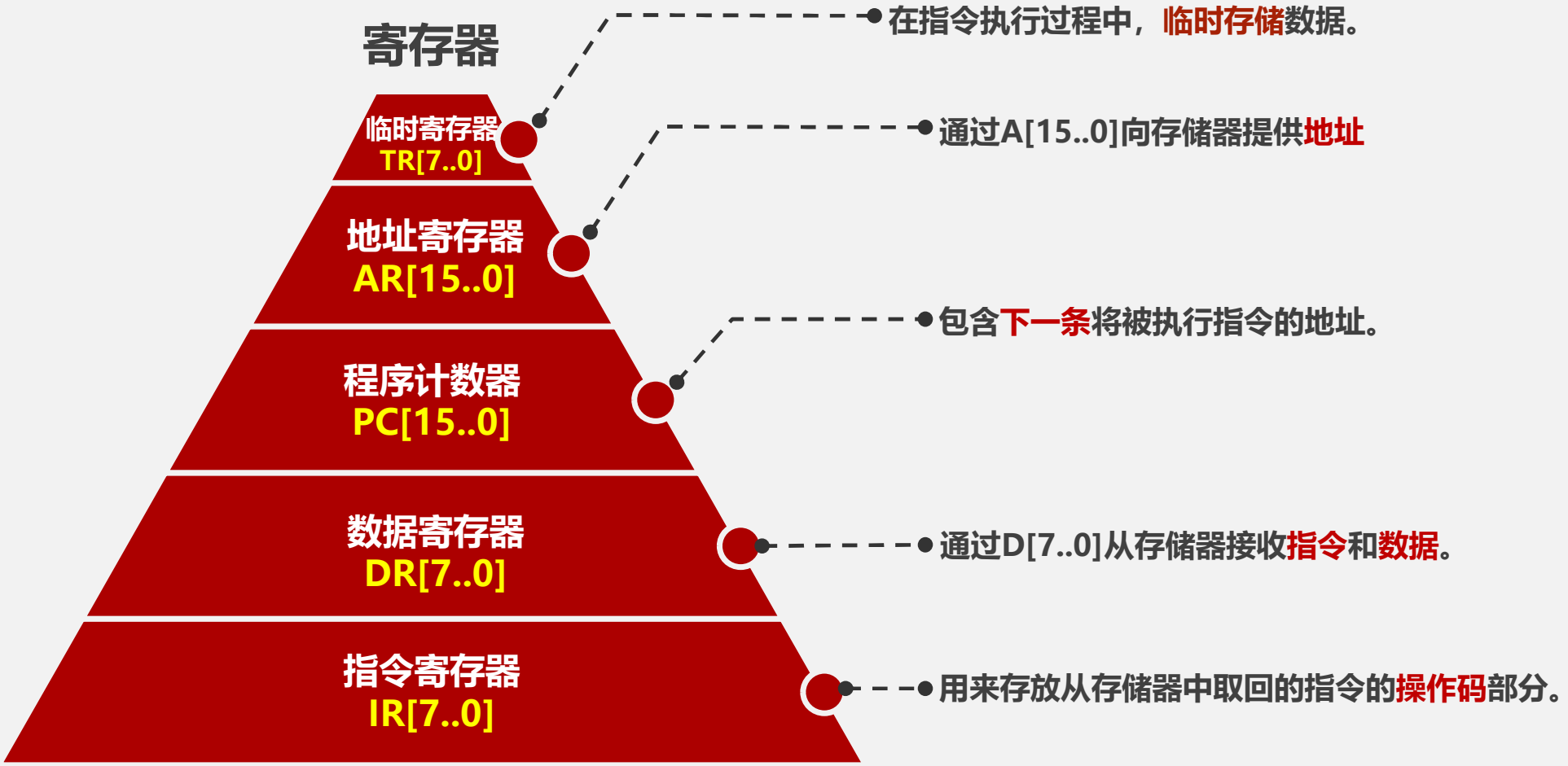
相对简单CPU设计

Table 3.1

Instruction set for a Relatively Simple CPU （Γ为内存地址）

Instruction	Instruction Code	Operation
NOP	0000 0000	No operation
LDAC	0000 0001 Γ	$AC = M[\Gamma]$
STAC	0000 0010 Γ	$M[\Gamma] = AC$
MVAC	0000 0011	$R = AC$
MOVR	0000 0100	$AC = R$
JUMP	0000 0101 Γ	GOTO Γ
JMPZ	0000 0110 Γ	IF (Z=1) THEN GOTO Γ
JPNZ	0000 0111 Γ	IF (Z=0) THEN GOTO Γ
ADD	0000 1000	$AC = AC + R$, If $(AC + R = 0)$ Then $Z = 1$ Else $Z = 0$
SUB	0000 1001	$AC = AC - R$, If $(AC - R = 0)$ Then $Z = 1$ Else $Z = 0$
INAC	0000 1010	$AC = AC + 1$, If $(AC + 1 = 0)$ Then $Z = 1$ Else $Z = 0$
CLAC	0000 1011	$AC = 0$, $Z = 1$
AND	0000 1100	$AC = AC \wedge R$, If $(AC \wedge R = 0)$ Then $Z = 1$ Else $Z = 0$
OR	0000 1101	$AC = AC \vee R$, If $(AC \vee R = 0)$ Then $Z = 1$ Else $Z = 0$
XOR	0000 1110	$AC = AC \oplus R$, If $(AC \oplus R = 0)$ Then $Z = 1$ Else $Z = 0$
NOT	0000 1111	$AC = AC'$, If $(AC' = 0)$ Then $Z = 1$ Else $Z = 0$

相对简单CPU设计


寄存器
临时寄存器
TR[7..0]
地址寄存器
AR[15..0]
程序计数器
PC[15..0]
数据寄存器
DR[7..0]
指令寄存器
IR[7..0]
在指令执行过程中，临时存储数据。
通过A[15..0]向存储器提供地址
包含下一条将被执行指令的地址。
通过D[7..0]从存储器接收指令和数据。
用来存放从存储器中取回的指令的操作码部分。

取指周期的操作序列

1. 将地址放在地址引脚 **A[15..0]**上，通过地址总线送给存储器。

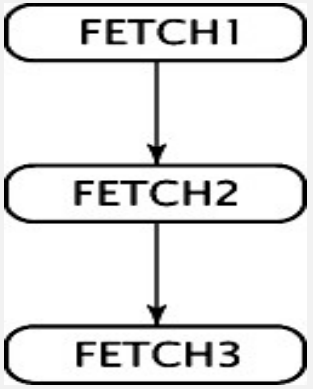
   FETCH1： AR[15..0]←PC[15..0]

2. 存储器内部译码并将需要的指令取出放到数据引脚**D[7..0]**经数据总线送到**DR[7..0]**。

   FETCH2： DR[7..0]←M[7..0]，PC←PC＋1

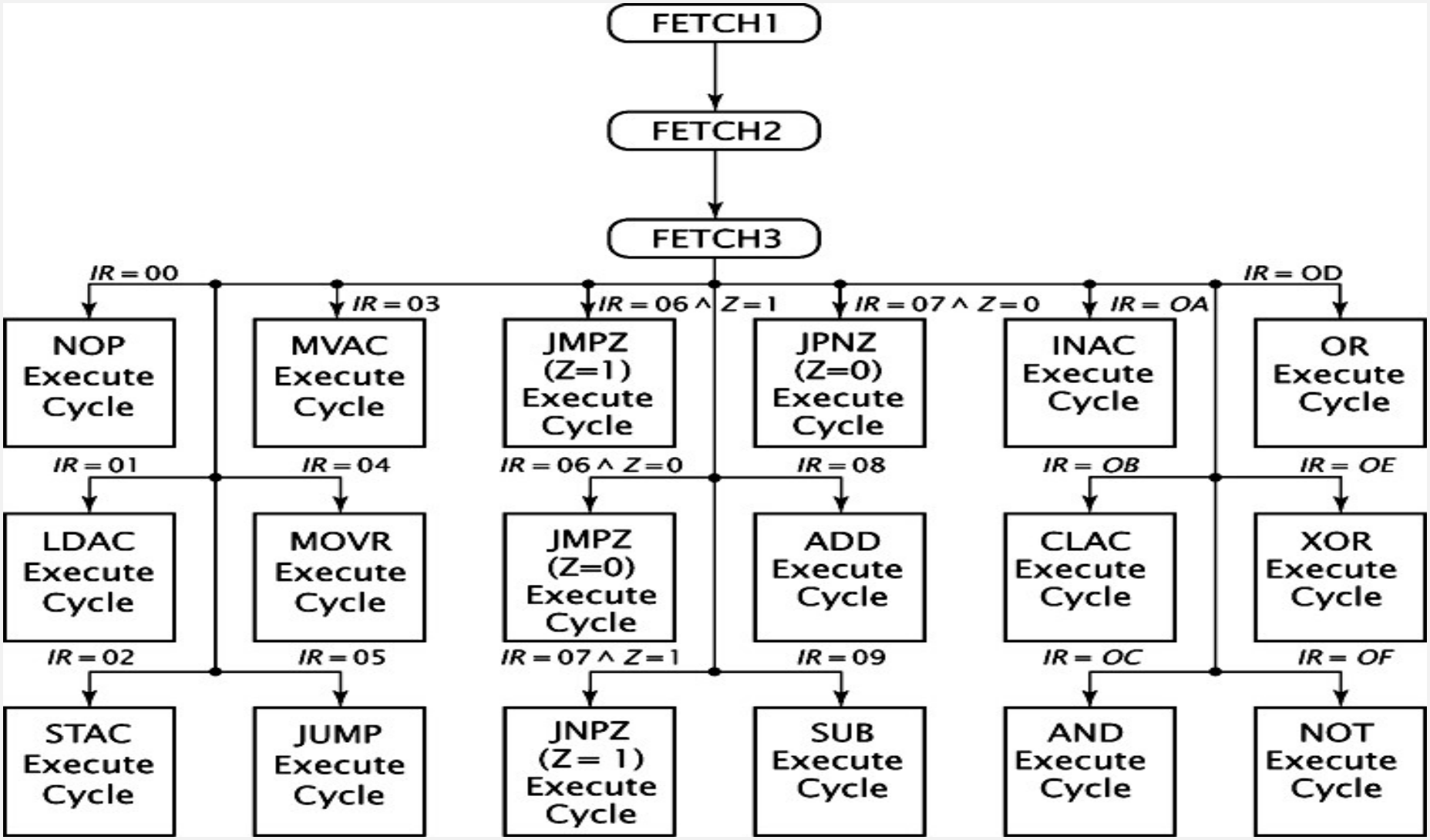
3. 指令的操作码 **DR[7..0]→IR[7..0]**。

   FETCH3： IR[7..0]←DR[7..0]，AR[15..0]←DR[15..0]



***注意：两条指令，JMPZ和JPNZ，具有两个不同的执行周期。**

相对简单CPU设计


FETCH1
FETCH2
FETCH3
IR = 00
IR = 03
IR = 06 ^ Z = 1
IR = 07 ^ Z = 0
IR = OA
IR = OD
NOP Execute Cycle
MVAC Execute Cycle
JMPZ (Z=1) Execute Cycle
JPNZ (Z=0) Execute Cycle
INAC Execute Cycle
OR Execute Cycle
IR = 01
IR = 04
IR = 06 ^ Z = 0
IR = 08
IR = OB
IR = OE
LDAC Execute Cycle
MOVR Execute Cycle
JMPZ (Z=0) Execute Cycle
ADD Execute Cycle
CLAC Execute Cycle
XOR Execute Cycle
IR = 02
IR = 05
IR = 07 ^ Z = 1
IR = 09
IR = OC
IR = OF
STAC Execute Cycle
JUMP Execute Cycle
JNPZ (Z = 1) Execute Cycle
SUB Execute Cycle
AND Execute Cycle
NOT Execute Cycle

指令执行的操作序列

1. NOP 指令 无操作

2. LDAC 指令 （1）多字节指令：三字节，**操作码[7..0] 地址低半部分[7..0] 地址高半部分[7..0]**

（2）功能：从存储器中获得地址，然后从存储器中获得数据并加载到累加器中。

LDAC1 : DR←M, PC←PC+1, AR←AR+1

LDAC2 : TR←DR, DR←M, PC←PC+1

LDAC3 : AR←DR : TR

LDAC4 : DR←M

LDAC5 : AC←DR

指令执行的操作序列

3. STAC 指令　（1）多字节指令：三字节，**操作码[7..0]** **地址低半部分[7..0]** **地址高半部分[7..0]**

（2）功能：从存储器中获得地址，然后从累加器中获得数据并加载到存储器中。

LDAC1 : DR←M, PC←PC+1, AR←AR+1

LDAC2 : TR←DR, DR←M, PC←PC+1

LDAC3 : AR←DR : TR

LDAC4 : DR←AC

LDAC5 : M←DR

4. MVAC 和 MOVR 指令

MVAC1 : R←AC

MOVR1 : AC←R

指令执行的操作序列

5. JUMP 指令

JUMP1 : DR←M, AR←AR+1

JUMP2 : TR←DR, DR←M

JUMP3 : PC←DR : TR

6. JMPZ 指令

Z=1 → JMPZY1 : DR←M, AR←AR+1

JMPZY2 : AC←R

JMPZY3 : PC←DR : TR

Z=0 → JMPZN1 : PC←PC+1

JMPZN2 : PC←PC+1

指令执行的操作序列

7. JPNZ 指令

Z=1 → JPNZY1 : DR←M, AR←AR+1

JPNZY2 : TR←DR, DR←M

JPNZY3 : PC←DR : TR

Z=0 → JPNZN1 : PC←PC+1

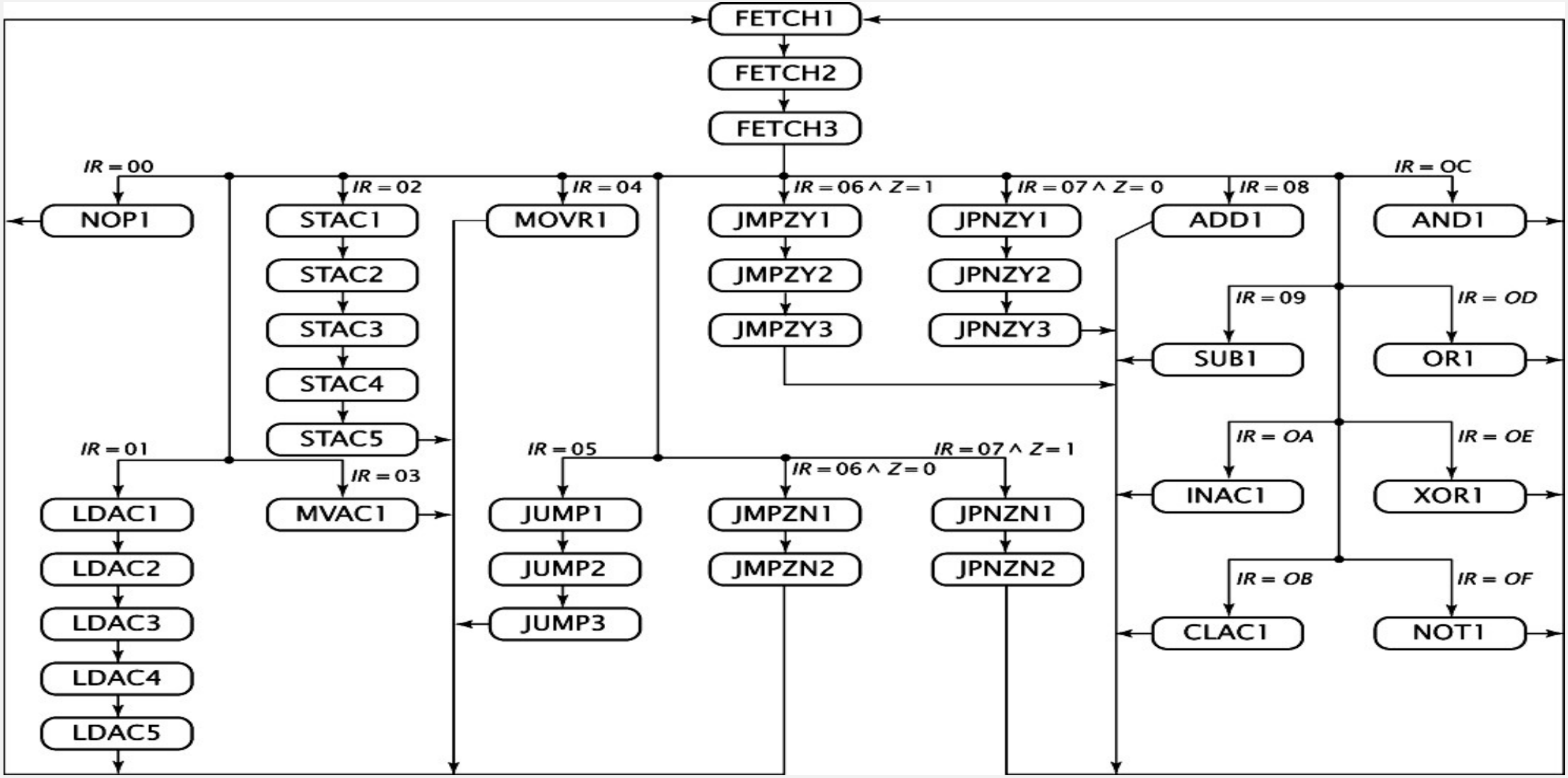
JPNZN2 : PC←PC+1

指令执行的操作序列

8. 其余指令：均在一个状态内完成

ADD1 :	AC←AC + R，IF (AC + R = 0) THEN Z←1 ELSE Z←0
SUB1 :	AC←AC - R，IF (AC - R = 0) THEN Z←1 ELSE Z←0
INAC1 :	AC←AC + 1，IF (AC + 1 = 0) THEN Z←1 ELSE Z←0
CLAC1 :	AC←0，Z←1
AND1 :	AC←AC∧R，IF (AC∧R = 0) THEN Z←1 ELSE Z←0
OR1 :	AC←AC∨R，IF (AC∨R = 0) THEN Z←1 ELSE Z←0
XOR1 :	AC←AC⊕R，IF (AC⊕R = 0) THEN Z←1 ELSE Z←0
NOT1 :	AC←AC'，IF (AC' = 0) THEN Z←1 ELSE Z←0

相对简单CPU设计


FETCH1
FETCH2
FETCH3
IR = 00
NOP1
IR = 02
STAC1
STAC2
STAC3
STAC4
STAC5
IR = 04
MOVR1
IR = 06 ∧ Z = 1
JMPZY1
JMPZY2
JMPZY3
IR = 07 ∧ Z = 0
JPNZY1
JPNZY2
JPNZY3
IR = 08
ADD1
IR = 09
SUB1
IR = OC
AND1
IR = OD
OR1
IR = OA
INAC1
IR = OE
XOR1
IR = OB
CLAC1
IR = OF
NOT1
IR = 01
LDAC1
LDAC2
LDAC3
LDAC4
LDAC5
IR = 03
MVAC1
IR = 05
JUMP1
JUMP2
JUMP3
IR = 06 ∧ Z = 0
JMPZN1
JMPZN2
IR = 07 ∧ Z = 1
JPNZN1
JPNZN2

下一节：微序列控制器

湖南大学

《计算机系统》课程教学组