

计算机组成和体系结构

arm DynamIQ

第五讲

四川大学网络空间安全学院

2020年3月23日

封面来自arm.com

版权声明

课件中所使用的图片、视频等资源版权归原作者所有。

课件原创内容采用 [创作共用署名—非商业使用—相同方式共享4.0国际版许可证\(Creative Commons BY-NC-SA 4.0 International License\)](#) 授权使用。

Copyright@四川大学网络空间安全学院计算机组成与体系结构课程组，2020



上期内容回顾

- 概论
 - 初识计算机系统、计算机发展历史、**计算机层次化结构、冯诺依曼模型**及非冯诺依曼模型
- 数字电子技术基础
 - 计算机系统中**数字的表示**、布尔代数运算符、常见组合逻辑电路和时序电路
- **计算机体系结构**
 - CPU结构（寄存器、算术逻辑单元、总线、时钟）、内存、I/O和中断
- **一个简化的计算机体系结构MARIE**
 - 数据通路的构成、指令集的编码和寻址、寄存器传输表示、MARIE汇编语言、控制信号和加入信号模式的寄存器传输表示
- 前四章内容各部分之间的联系

本期学习目标

- 学习内容：教材第5章“仔细审视指令集架构”
 - 进一步加深理解指令集在整个计算机系统中的重要作用
 - 初步掌握正确解读现有指令集的方法
 - 初步了解指令集设计的方法
- 教材5.2节指令格式
 - 学习掌握指令格式设计中需要考虑的因素和现有的解决方案
- 教材5.3节指令类型
 - 学习掌握常见的指令类型
- 教材5.4节寻址
 - 学习掌握常见的寻址类型



中英文缩写对照表

英文缩写	英文全称	中文全称
ISA	Instruction Set Architecture	指令集架构
GPR	General-purpose Register	通用寄存器
RISC	Reduced Instruction Set Computer	精简指令集计算机
RPN	Reverse Polish Notation	逆波兰式（后缀表达式）



指令集架构

引言

Q1: 指令集架构包含哪些内容

Q2: 为什么要学习指令集架构

引言

Q1: 指令集架构包含哪些内容

指令集架构(Instruction Set Architecture, ISA)包含了一个计算机体系结构所有指令构成的集合，以及这些指令的编码方法。

Q2: 为什么要学习指令集架构

8086 INSTRUCTION SET			
OPCODE	DESCRIPTION	JNAE	slabel Jump if not above or equal
AAA	ASCII adjust addition	JNB	slabel Jump if not below
ASCII adjust division	JNBE	slabel Jump if below or equal	
AAM	ASCII adjust multiply	JNC	slabel Jump if no carry
AAS	ASCII adjust subtraction	JNE	slabel Jump if not equal
ADC	dt,sc Add with carry	JNG	slabel Jump if not greater
ADD	dt,sc Add	JNGE	slabel Jump if not greater or equal
AND	dt,sc Logical AND	JNL	slabel Jump if not less
CALL	proc Call a procedure	JNLE	slabel Jump if not less or equal
CBW	Convert byte to word	JNZ	slabel Jump if not zero
CLC	Clear carry flag	JNO	slabel Jump if not overflow
CDL	Clear direction flag	JNP	slabel Jump if not parity
CLI	Clear interrupt flag	JNS	slabel Jump if not sign
CMC	Complement carry flag	JO	slabel Jump if overflow
CMP	dt,sc Compare	JPO	slabel Jump if parity odd
CMPSB	[dt,sc] Compare string	JP	slabel Jump if parity
CMPSW	" " bytes	JPE	slabel Jump if parity even
	words	JS	slabel Jump if sign
CWD	Convert word to double word	JZ	slabel Jump if zero
DAA	Decimal adjust addition	LAHF	Load AH from flags
DAS	Decimal adjust subtraction	LDS	dt,sc Load pointer using DS
DEC	dt Decrement	LEA	dt,sc Load effective address
DIV	sc Unsigned divide	LES	dt,sc Load pointer using ES
ESC	code,sc Escape	LOCK	Lock bus
HLT	Halt	LODS	[sc] Load string
IDIV	sc Integer divide	LODSB	" " bytes
IMUL	sc Integer multiply	LODSW	" " words
IN	ac,port Input from port	LOOP	slabel Loop
INC	dt Increment	LOOPB	slabel Loop if equal
INT	type Interrupt	LOOPZ	slabel Loop if zero
INTO	Interrupt if overflow	LOOPNE	slabel Loop if not equal
IRET	Return from interrupt	LOOPNZ	slabel Loop if not zero
JA	slabel Jump if above	MOV	dt,sc Move
JAE	slabel Jump if above or equal	MOVSB	[dt,sc] Move string
JB	slabel Jump if below	MOVSW	" " bytes
JBE	slabel Jump if below or equal	MUL	sc Unsigned multiply
JC	slabel Jump if carry	NEG	dt Negate
JCXZ	slabel Jump if CX is zero	NOP	No operation
JE	slabel Jump if equal	NOT	dt Logical NOT
JG	slabel Jump if greater	OR	dt,sc Logical OR
JGE	slabel Jump if greater or equal	OUT	port,ac output to port
JL	slabel Jump if less	POP	dt Pop word off stack
JLE	slabel Jump if less or equal	POPF	Pop flags off stack
JMP	label Jump	PUSH	sc Push word onto stack
JNA	slabel Jump if not above		

x86指令集

引言

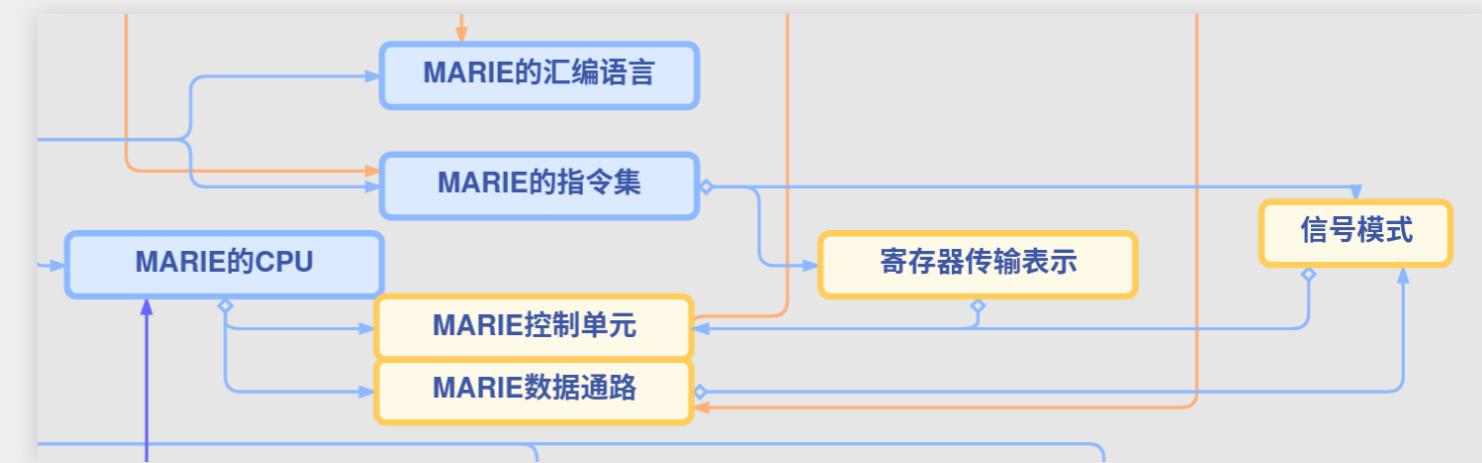
Q1: 指令集架构包含哪些内容

指令集架构(Instruction Set Architecture, ISA)包含了一个计算机体系结构所有指令构成的集合，以及这些指令的编码方法。

Q2: 为什么要学习指令集架构

指令集

- 是计算机系统中软硬件的接口
- 了解指令集架构有助于理解CPU的设计



引言

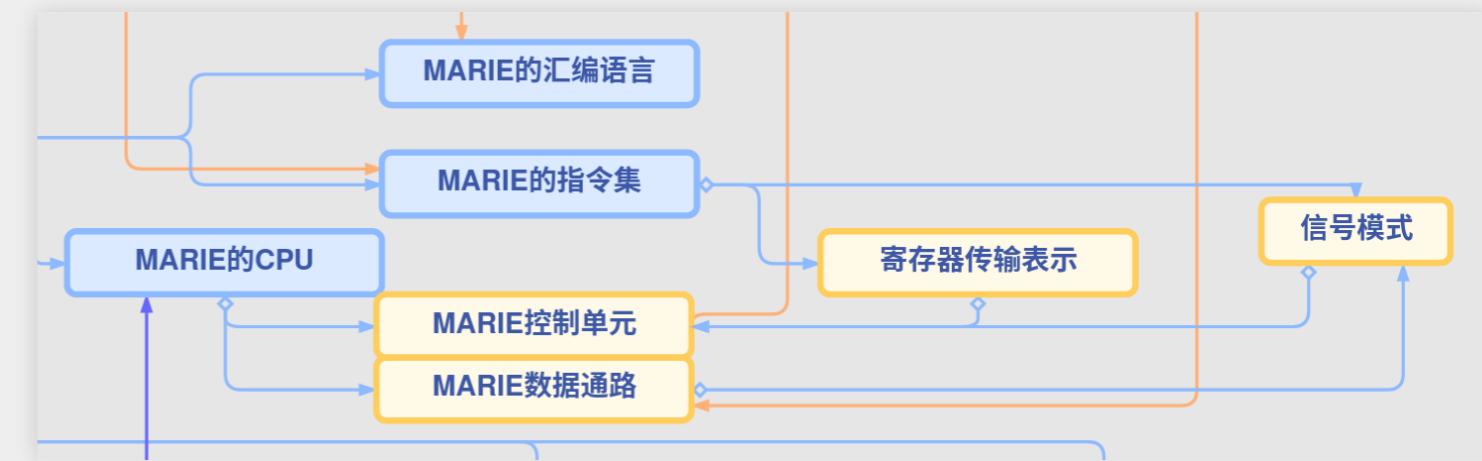
Q1: 指令集架构包含哪些内容

指令集架构(Instruction Set Architecture, ISA)包含了一个计算机体系结构所有指令构成的集合，以及这些指令的编码方法。

Q2: 为什么要学习指令集架构

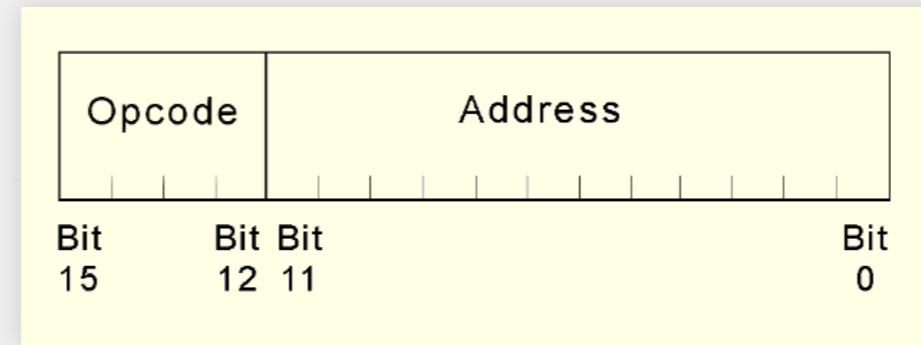
指令集

- 是计算机系统中软硬件的接口
- 了解指令集架构有助于理解CPU的设计和如何设计CPU



指令集设计决策

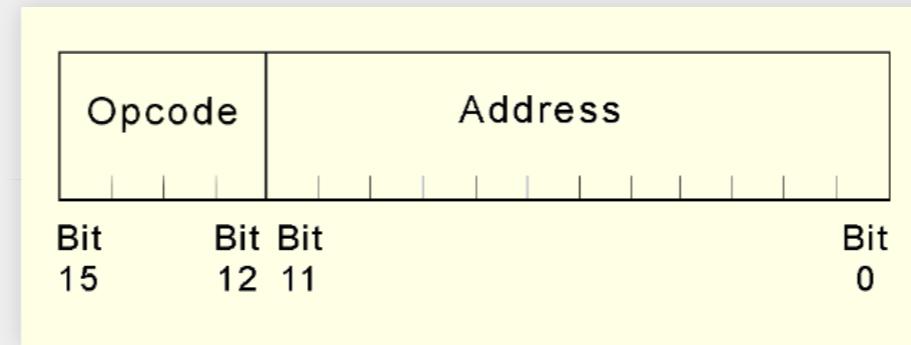
我们用MARIE的指令作为例子，分析指令集设计需要考虑的因素



- **16位固定长度**的指令
- **15条指令**，对应**4位固定长度**操作码
- 最多有一个**12位显式**操作数
- 算术逻辑单元输入来自**AC寄存器**和**内存(MBR寄存器)**
- 支持内存的**直接寻址**和**间接寻址**

指令集设计决策

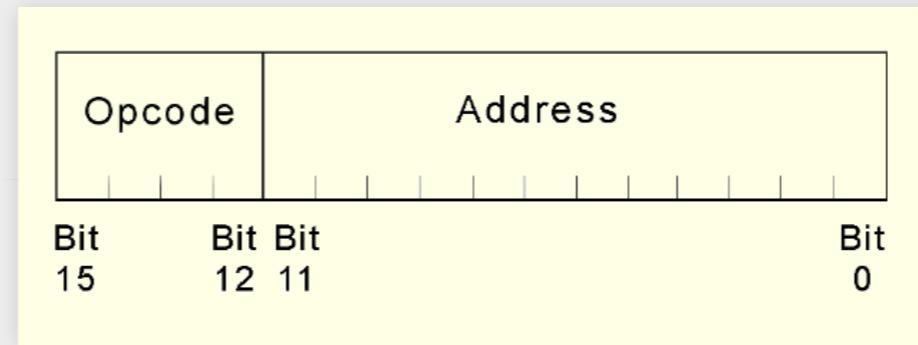
我们用MARIE的指令作为例子，分析指令集设计需要考虑的因素



- **16位固定长度**的指令
- **15条指令**，对应**4位固定长度**操作码
- 最多有一个**12位显式**操作数
- 算术逻辑单元输入来自**AC寄存器**和**内存(MBR寄存器)**
- 支持内存的**直接寻址**和**间接寻址**
- 指令的长度(表示一条指令需要的比特数以及是否是固定值)

指令集设计决策

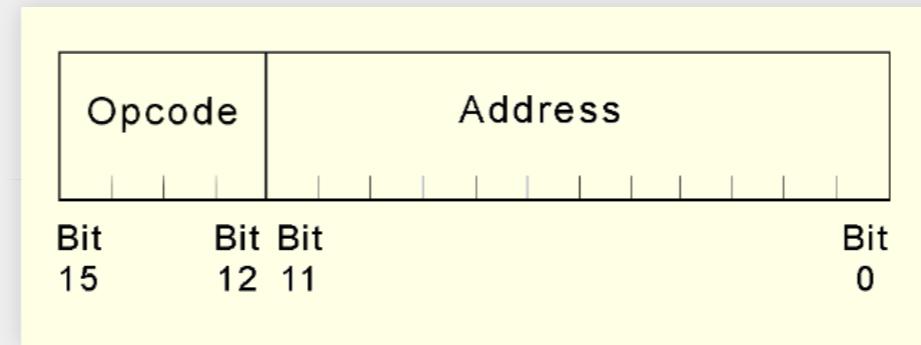
我们用MARIE的指令作为例子，分析指令集设计需要考虑的因素



- **16位固定长度**的指令
- **15条指令**，对应**4位固定长度**操作码
- 最多有一个**12位显式**操作数
- 算术逻辑单元输入来自**AC寄存器**和**内存(MBR寄存器)**
- 支持内存的**直接寻址**和**间接寻址**
- 指令的长度(表示一条指令需要的比特数以及是否是固定值)
- 指令类型的数量以及如何对指令类型进行编码

指令集设计决策

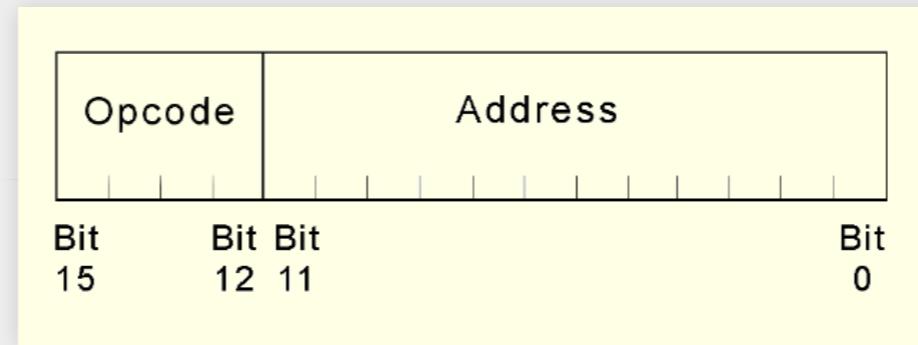
我们用MARIE的指令作为例子，分析指令集设计需要考虑的因素



- **16位固定长度**的指令
- **15条指令**，对应**4位固定长度**操作码
- 最多有一个**12位显式**操作数
- 算术逻辑单元输入来自**AC寄存器**和**内存(MBR寄存器)**
- 支持内存的**直接寻址**和**间接寻址**
- 指令的长度(表示一条指令需要的比特数以及是否是固定值)
- 指令类型的数量以及如何对指令类型进行编码
- 操作数的类型、数量和对应的数据范围

指令集设计决策

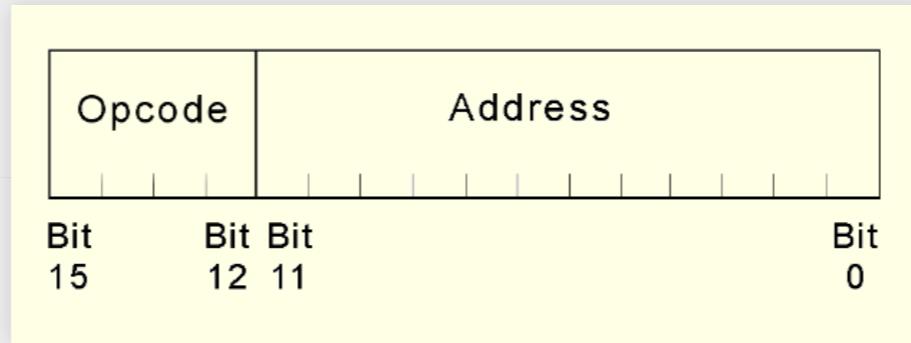
我们用MARIE的指令作为例子，分析指令集设计需要考虑的因素



- **16位固定长度**的指令
- **15条指令**，对应**4位固定长度**操作码
- 最多有一个**12位显式**操作数
- 算术逻辑单元输入来自**AC寄存器**和**内存(MBR寄存器)**
- 支持内存的**直接寻址**和**间接寻址**
- 指令的长度(表示一条指令需要的比特数以及是否是固定值)
- 指令类型的数量以及如何对指令类型进行编码
- 操作数的类型、数量和对应的数据范围
- 指令操作数的存储

指令集设计决策

我们用MARIE的指令作为例子，分析指令集设计需要考虑的因素



- **16位固定长度**的指令
- **15条指令**，对应**4位固定长度**操作码
- 最多有一个**12位显式**操作数
- 算术逻辑单元输入来自**AC寄存器**和**内
存(MBR寄存器)**
- 支持内存的**直接寻址**和**间接寻址**
- 指令的长度(表示一条指令需要的比特数以及是否是固定值)
- 指令类型的数量以及如何对指令类型进行编码
- 操作数的类型、数量和对应的数据范围
- 指令操作数的存储
- 指令集架构支持的内存寻址方式

指令集的评价指标

指令集的评价指标

- 指令执行效率

$$\text{CPU Time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{avg. cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

指令集的评价指标

- 指令执行效率

$$\text{CPU Time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{avg. cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

- 指令和操作数读取效率

指令集的评价指标

- 指令执行效率

$$\text{CPU Time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{avg. cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

- 指令和操作数读取效率
- 指令通过流水线(Pipeline)执行的效率

指令集的评价指标

- 指令执行效率

$$\text{CPU Time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{avg. cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

- 指令和操作数读取效率
- 指令通过流水线(Pipeline)执行的效率
- 实现同样的运算需要的指令数量

指令集的评价指标

- 指令执行效率

$$\text{CPU Time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{avg. cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

- 指令和操作数读取效率
- 指令通过流水线(Pipeline)执行的效率
- 实现同样的运算需要的指令数量
- 指令译码复杂度(控制单元电路实现复杂度)

指令集的评价指标

- 指令执行效率

$$\text{CPU Time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{avg. cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

- 指令和操作数读取效率
- 指令通过流水线(Pipeline)执行的效率
- 实现同样的运算需要的指令数量
- 指令译码复杂度(控制单元电路实现复杂度)
- 可寻址空间的大小

常见设计决策及特点

- 采用短指令提高指令的**读取和译码性能，限制了指令类型的数量**，对操作数的个数和大小有更严格的要求
- 定长指令的译码相对容易但是浪费了存储空间
- 扩展操作码可以**充分利用定长指令的存储空间从而表示更多的指令类型**，同时可能降低指令长度
- 按照字长寻址的内存难以存取单个字节，**大多数现有体系结构采用按字节寻址**

多字节存储方式

Q: 如何用字节寻址的内存存储字? A: 大端存储和小端存储

端(Endianness)代表了计算机中存储多字节数据的顺序

- 小端(Little Endian): 低位有效字节存储在低位地址
- 大端(Big Endian): 低位有效字节存储在高位地址

假设字长为32,一个地址为 x 的字 $b_3 b_2 b_1 b_0$ 存储单元排列如下:

地址	x	$x + 1$	$x + 2$	$x + 3$
小端存储	b_0	b_1	b_2	b_3
大端存储	b_3	b_2	b_1	b_0

大端存储和小端存储实例

假设字长为32, 初始地址为0x200, 连续三个无符号整数
0xABCD1234、0x00FF4321和0x10的内存存储情况如下:

Address	Big Endian	Little Endian
0x200	AB	34
0x201	CD	12
0x202	12	CD
0x203	34	AB
0x204	00	21
0x205	FE	43
0x206	43	FE
0x207	21	00
0x208	00	10
0x209	00	00
0x20A	00	00
0x20B	10	00

大端存储和小端存储总结

- 大端存储和小端存储代表的**字节顺序**而不是位顺序
- 大端存储的优缺点：
 - 更加自然：字节顺序和地址顺序一致
 - 可以快速确定数字的符号(符号位在最低地址)
 - 字符串的存储顺序和整数一致："ABC" = 0x41 0x42 0x43 => 0x414243
- 小端存储的优缺点
 - 高位数字转为低位数字更方便：32位数字 0x12345678 和低16位 0x5678 的地址是一样的

CPU内部的数据存储

CPU内部的数据存储

- 堆栈型架构：操作数**隐式**放置在一个堆栈中

CPU内部的数据存储

- 堆栈型架构：操作数**隐式**放置在一个堆栈中
- 累加器型架构：当指令对应的运算包含两个操作数时，其中一个操作数**隐式**放置在累加器寄存器中

CPU内部的数据存储

- 堆栈型架构：操作数**隐式**放置在一个堆栈中
- 累加器型架构：当指令对应的运算包含两个操作数时，其中一个操作数**隐式**放置在累加器寄存器中
- 通用寄存器型架构：操作数**可以**保存在寄存器中

CPU内部的数据存储

- 堆栈型架构：操作数**隐式**放置在一个堆栈中
- 累加器型架构：当指令对应的运算包含两个操作数时，其中一个操作数**隐式**放置在累加器寄存器中
- 通用寄存器型架构：操作数**可以**保存在寄存器中
 - 存储器-存储器型：可以有多个操作数存放在内存，允许实现任何操作数都不在寄存器的指令

CPU内部的数据存储

- 堆栈型架构：操作数**隐式**放置在一个堆栈中
- 累加器型架构：当指令对应的运算包含两个操作数时，其中一个操作数**隐式**放置在累加器寄存器中
- 通用寄存器型架构：操作数**可以**保存在寄存器中
 - 存储器-存储器型：可以有多个操作数存放在内存，允许实现任何操作数都不在寄存器的指令
 - 寄存器-存储器型：至少一个操作数在寄存器中，另一个操作数在内存中

CPU内部的数据存储

- 堆栈型架构：操作数**隐式**放置在一个堆栈中
- 累加器型架构：当指令对应的运算包含两个操作数时，其中一个操作数**隐式**放置在累加器寄存器中
- 通用寄存器型架构：操作数**可以**保存在寄存器中
 - 存储器-存储器型：可以有多个操作数存放在内存，允许实现任何操作数都不在寄存器的指令
 - 寄存器-存储器型：至少一个操作数在寄存器中，另一个操作数在内存中
 - 取-存型：所有操作数都在寄存器中

不同存储方式指令集架构示例

例子：假设要实现地址为 x 的存储单元和地址为 y 的存储单元中的数据相加之后写入地址为 z 的存储单元

堆栈型架构：

```
PUSH x  
PUSH y  
ADD  
POP z
```

累加器型架构：

```
LOAD x  
ADD y  
STORE z
```

通用寄存器型架构(存储器-存储器型)：

```
ADD ax, x, y  
STORE ax, z
```

通用寄存器型架构(寄存器-存储器型)：

```
LOAD ax, x  
IADD ax, y  
STORE ax, z
```

通用寄存器型架构(取-存型)：

```
LOAD ax, x  
LOAD bx, y  
ADD cx, ax, bx  
STORE cx, z
```

指令操作数的数量和长度

按照操作数的数量分类：

- 零地址指令：仅有操作码
- 单地址指令：操作码 + 一个操作数(通常是一个内存地址)
- 双地址指令：操作码 + 两个操作数(通常是两个寄存器，或者是一个寄存器一个地址)
- 三地址指令：操作码 + 三个操作数(三个寄存器地址，或者是寄存器地址和内存地址的组合)

不同存储方式支持的操作数数量：

存储方式	堆栈型	累加器型	通用寄存器型
零地址指令	是	是	是
单地址指令	是	是	是
双地址指令	否	否	是
三地址指令	否	否	是

堆栈型架构和逆波兰式

堆栈型架构可以实现

- 所有的运算都是零地址指令
- 单地址指令只包括PUSH和POP

考虑一个三地址指令OP dt, sc1, sc2,
其中

- OP代表操作码
- dt代表目的地址
- sc1代表第一个操作数的地址
- sc2代表第二个操作数的地址

则该指令可以改写为

```
PUSH sc1  
PUSH sc2  
OP  
POP dt
```

堆栈型架构和逆波兰式

堆栈型架构可以实现

- 所有的运算都是零地址指令
- 单地址指令只包括PUSH和POP

堆栈型架构的应用：求解由逆波兰式(后缀表达式)表达的长算术表达式

- 后缀表达式：运算符放在表达式的最后
- 例：4 + 3的后缀表达式是4 3 +

考虑一个三地址指令OP dt, sc1, sc2,
其中

- OP代表操作码
- dt代表目的地址
- sc1代表第一个操作数的地址
- sc2代表第二个操作数的地址

则该指令可以改写为

```
PUSH sc1  
PUSH sc2  
OP  
POP dt
```

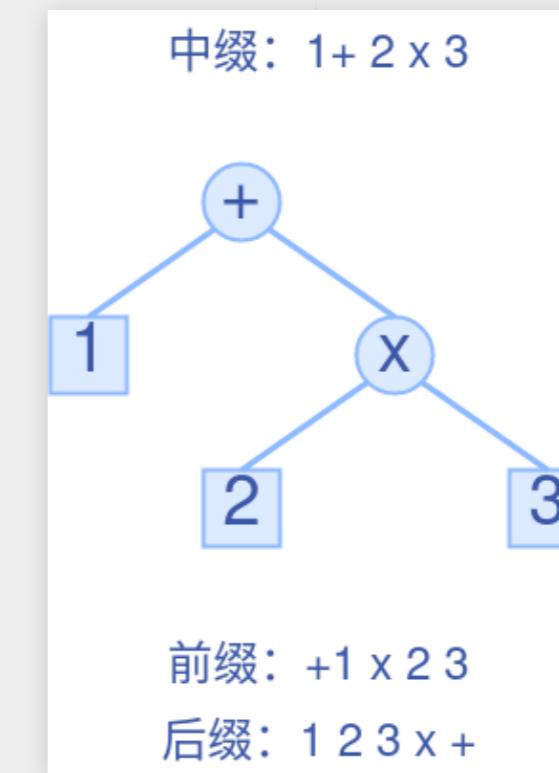
后缀表达式的转换

表达式树遍历法

- 给定一个前缀/中缀/后缀描述的表达式，写出其对应的前缀/中缀/后缀表达式
- 写出原前缀/中缀/后缀表达式对应的表达式树，对表达式树进行对应的前缀/中缀/后缀遍历，适当添加括号保证执行顺序
- 中缀表达式每一步都加括号可以保证执行顺序，前缀表达式和后缀表达式不需要括号

按优先级替换法

按照运算符优先级依次将该运算符对应的中缀表达式变换为后缀表达式



$$1 + 2 \times 3 \Rightarrow 1 + 2 3 \times \Rightarrow 1 2 3 \times +$$

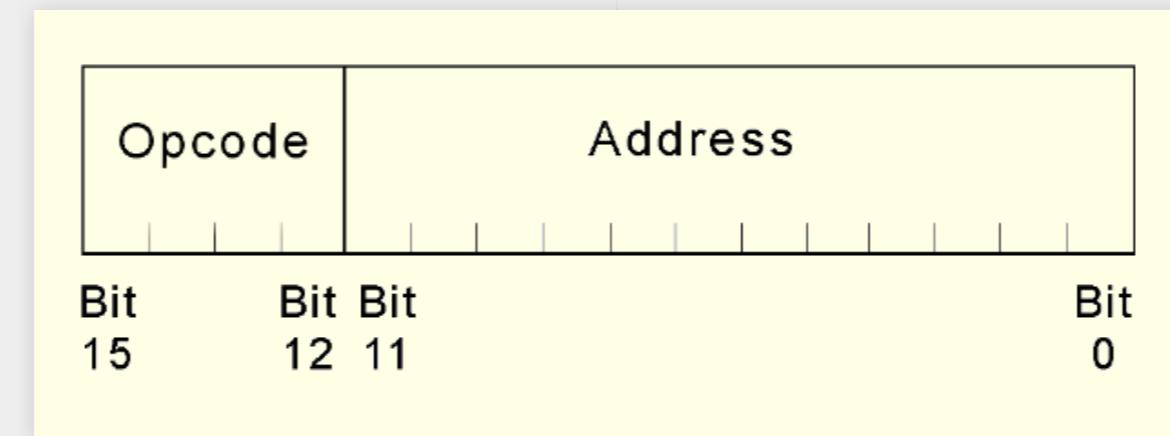
堆栈执行后缀表达式

- 如果当前token是操作数，入栈
- 如果当前token是运算符，从栈顶读取操作数(出栈)进行计算，将结果入栈

例：对于后缀表达式1 2 3 × +

token	栈	操作
1	(空)	PUSH 1
2	1	PUSH 2
3	1 2	PUSH 3
×	1 2 3	POP 3; POP 2; MUL 2, 3; PUSH 6
+	1 6	POP 6; POP 1; ADD 1, 6; PUSH 7
\$	7	

扩展操作码



MARIE的指令集中**采用固定长度的操作码表示不同的指令类型**

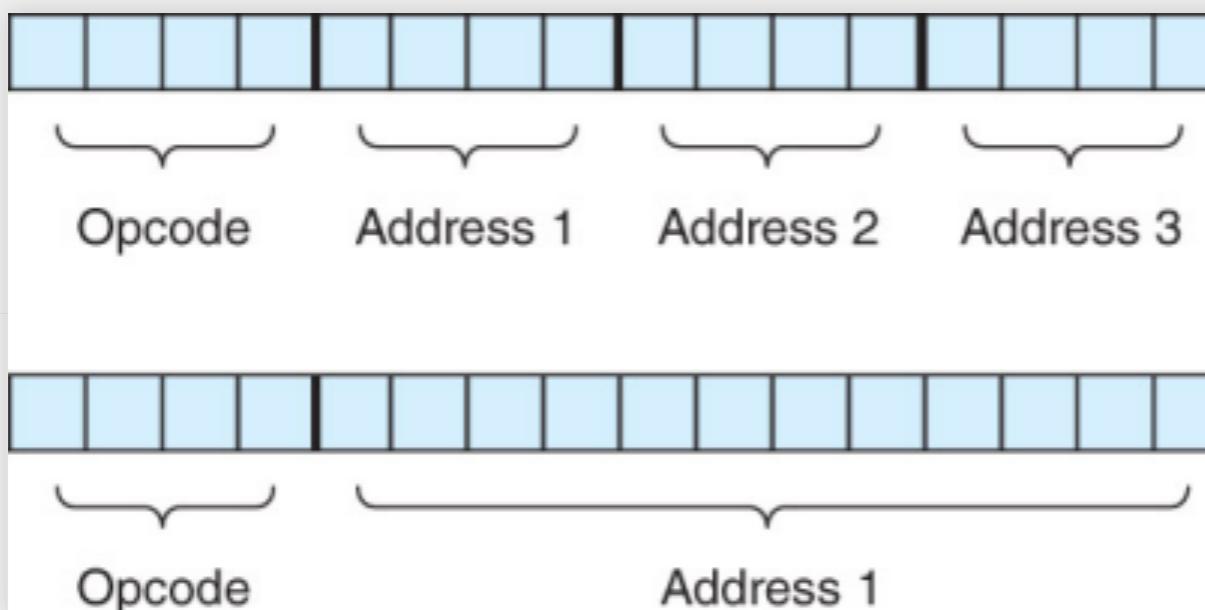
- 定长指令的译码相对容易但是浪费了存储空间
- 变长指令译码复杂
- 不同类型的指令需要的操作数数量不同

扩展操作码(Expanding Opcodes): 在定长的指令中，通过变长的操作码，表示更多的指令类型

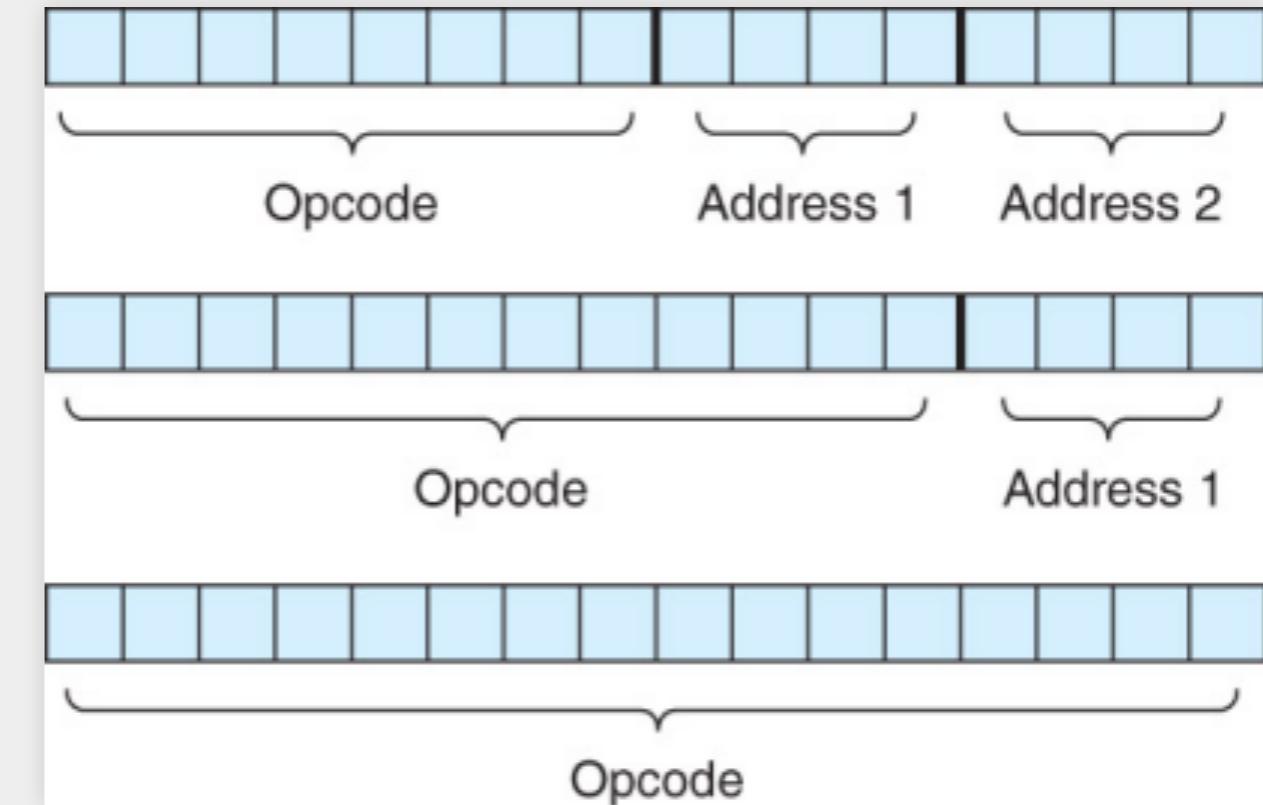
扩展操作码的示例

假设某体系结构中有16个寄存器和4K个可寻址存储单元，采用16位字长的指令

- 16个寄存器对应4位寄存器地址
- 4K个内存可寻址单元对应12位内存地址



采用固定操作码长度的指令类型



采用扩展操作码的指令类型

扩展操作码的编码

- 区别操作码的长度：采用**转义操作码**(Escape Opcode)
- 转义操作码的作用：表示操作码的编码超过当前长度

示例：假设某体系结构中地址长度为4，如何采用16位字长的指令编码下列指令：

- 15条三地址指令
- 14条双地址指令
- 31条单地址指令
- 16条零地址指令

0000 R1 R2 R3	15 three-address codes
...	
1110 R1 R2 R3	14 two-address codes
1111 - escape opcode	
1111 0000 R1 R2	31 one-address codes
...	
1111 1101 R1 R2	16 zero-address codes
1111 1110 - escape opcode	
1111 1110 0000 R1	16 zero-address codes
...	
1111 1111 1110 R1	16 zero-address codes
1111 1111 1111 - escape opcode	
1111 1111 1111 0000	16 zero-address codes
...	
1111 1111 1111 1111	16 zero-address codes

扩展操作码的译码

- 按照编码模式分类后分别编码
- 示例：对于右边的编码，对应的译码方式如下所示

```
if (IR[15:12] != 1111) {  
    译码并执行对应的三地址指令  
} else (IR[15:9] != 1111 111) {  
    译码并执行对应的双地址指令  
} else (IR[15:4] != 1111 1111 1111) {  
    译码并执行对应的单地址指令  
} else {  
    译码并执行对应的零地址指令  
}
```

0000 R1 R2 R3	15 three-address codes
...	
1110 R1 R2 R3	
1111 - escape opcode	
1111 0000 R1 R2	14 two-address codes
...	
1111 1101 R1 R2	
1111 1110 - escape opcode	
1111 1110 0000 R1	31 one-address codes
...	
1111 1111 1110 R1	
1111 1111 1111 - escape opcode	
1111 1111 1111 0000	16 zero-address codes
...	
1111 1111 1111 1111	

利用位模式判断扩展操作码是否可行

- 判断是否采用 K 位可以编码一个使用扩展操作码的指令集：使用**位模式**
- 每一个位模式代表了一个特定的操作
- 编码长度要足够表示所有可能的操作组合
- 示例：假设地址长度为4,采用16位指令是否与编码右侧两个指令集

- 15条三地址指令：位模式数量

$$15 \times 2^4 \times 2^4 \times 2^4 = 61440$$

- 14条双地址指令：位模式数量 $14 \times 2^4 \times 2^4 = 3584$

- 31条单地址指令：位模式数量 $31 \times 2^4 = 496$

- 16条零地址指令：位模式数量16

合计 $65536 \leq 2^{16}$ 个位模式，**可以**被16位指令编码

- 15条三地址指令：位模式数量

$$15 \times 2^4 \times 2^4 \times 2^4 = 61440$$

- **15**条双地址指令：位模式数量 $15 \times 2^4 \times 2^4 = 3840$

- 31条单地址指令：位模式数量 $31 \times 2^4 = 496$

- 16条零地址指令：位模式数量16

合计 $65792 > 2^{16}$ 个位模式，**不能**被16位指令编码

指令类型

指令类型

- 数据传输

表示数据的传输，例如：

- 内存传送到寄存器：LOAD
- 寄存器之间数据传输：MOVE
- 寄存器传送到内存：STORE
- 内存/寄存器到栈：PUSH
- 栈到内存/寄存器：POP
- 原子操作：TEST, EXCHANGE等

指令类型

- 数据传输
 - 算术运算
- 表示算术运算，例如：
- 算术运算：ADD、SUBTRACT、MULTIPLY、DIVIDE等
 - 自增运算：INCREMENT等

指令类型

- 数据传输
 - 算术运算
 - 布尔逻辑运算
- 表示布尔逻辑运算，例如：
- 布尔代数运算符：AND，OR，XOR，NOT等
 - 比较函数：EQUAL，NEQUAL，LT，GT，LEQ，GEQ等

指令类型

- 数据传输
- 算术运算
- 布尔逻辑运算
- 位操作

将数据看作01数列进行按位操作的运算，例如

- 按位的逻辑运算：AND，OR，XOR等
- 算术逻辑移位：LSHIFT和RSHIFT
 - 符号位不移位，但是如果符号位上的移出位和符号位不同将导致算术溢出
 - 例：(10110000) = -80左移1位得到
(11100000) = -32，但是移出位0 ≠ 符号位1,因此发生算术溢出
 - 例：(11100000) = -32左移1位得到
(11000000) = -64,移出位1 = 符号位1,因此没有发生算术溢出
- 循环移位指令：LROTATE和RROTATE
 - 移出位填充到对侧
 - 例：(00000001)右移1位得到(00000000)，右循环移位1位得到(10000000)

指令类型

- 数据传输
- 算术运算
- 布尔逻辑运算
- 位操作
- 输入/输出指令

指令类型

- 数据传输
- 算术运算
- 布尔逻辑运算
- 位操作
- 输入/输出指令
- 传送控制指令

改变程序执行顺序的指令，例如：

- 跳转和条件跳转：JUMP、JE等
- 函数调用：CALL
- 中断：INT

指令类型

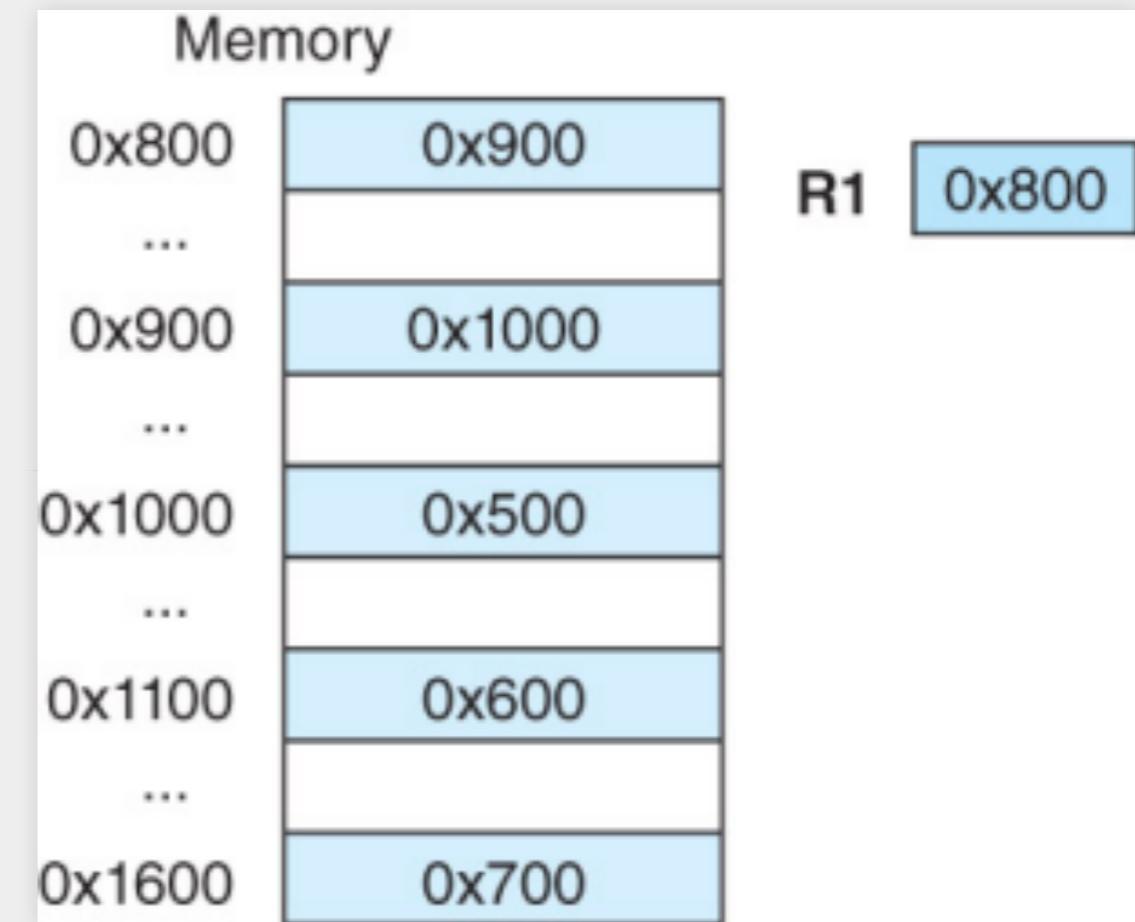
- 数据传输
 - 算术运算
 - 布尔逻辑运算
 - 位操作
 - 输入/输出指令
 - 传送控制指令
 - 专用指令
- 一些特殊的指令，例如：
- 无指令： NOP （用于流水线避免数据冲突）

寻址方式

常见的寻址方法：

- 立即寻址(Immediate Addressing)
- 直接寻址(Direct Addressing)
- 间接寻址(Indirect Addressing)
- 变址寻址(Indexed Addressing)
- 基址寻址(Based Addressing)
- 堆栈寻址(Stack Addressing)

寻址方法

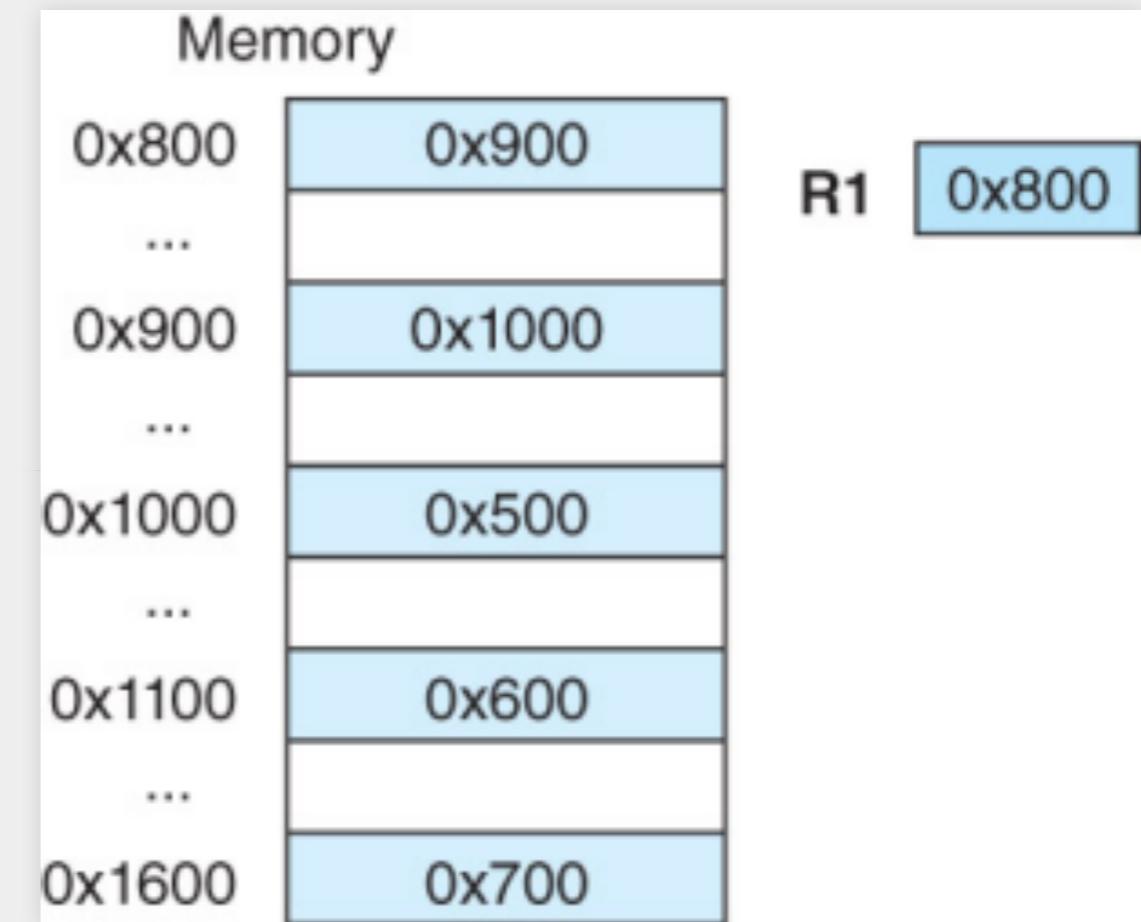


寻址方法

立即寻址

指令中的地址直接指定操作数

例：对于右边的内存，采用立即寻址，执行指令LOAD
0x800之后寄存器AC的值是0x0800



寻址方法

直接寻址

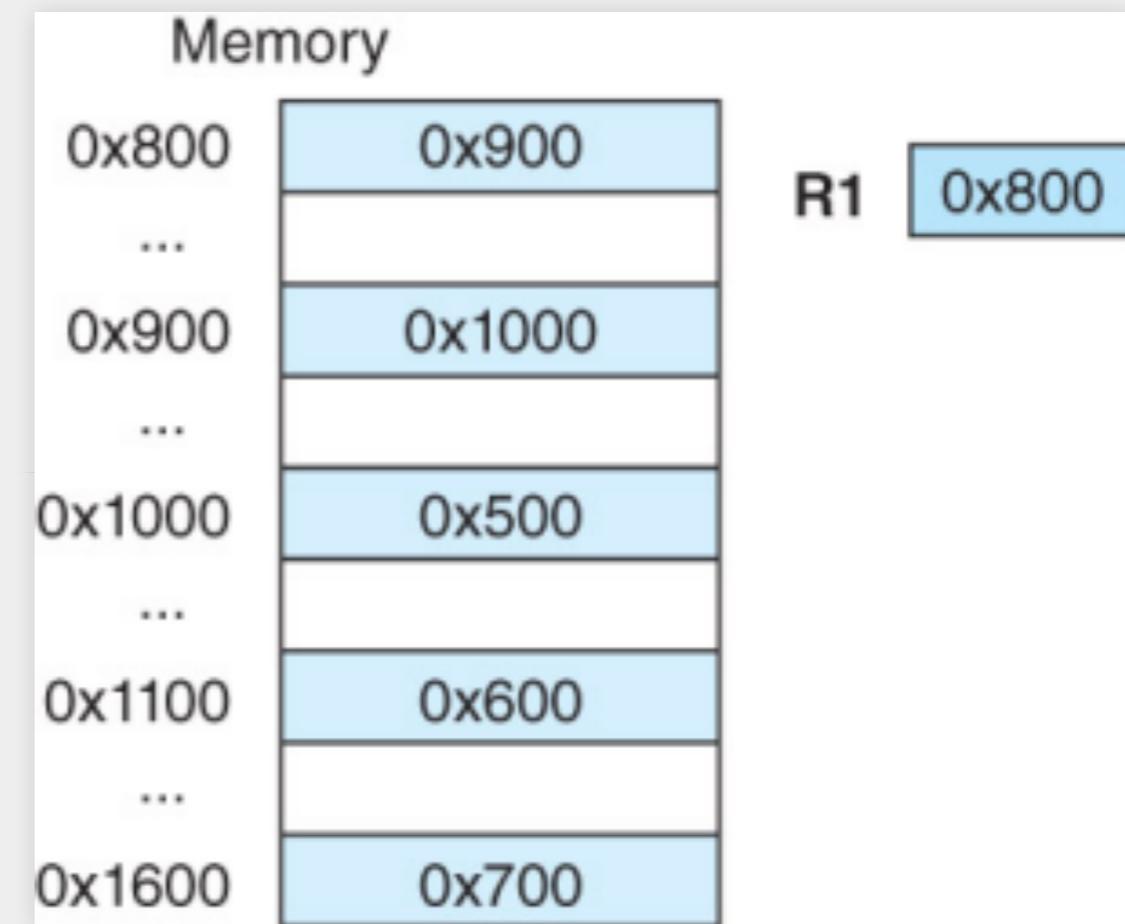
指令中的地址直接指向操作数的地址

指令中12位地址可寻址空间大小： 2^{12}

例：对于右边的内存，采用直接寻址，执行指令LOAD
0x800之后寄存器AC的值是0x0900

寄存器(直接)寻址

指令中不用地址数，而是用一个寄存器名字，例：LOAD R1。寻址地址为对应的寄存器中存储的值，寻址方法和直接寻址一致



寻址方法

间接寻址

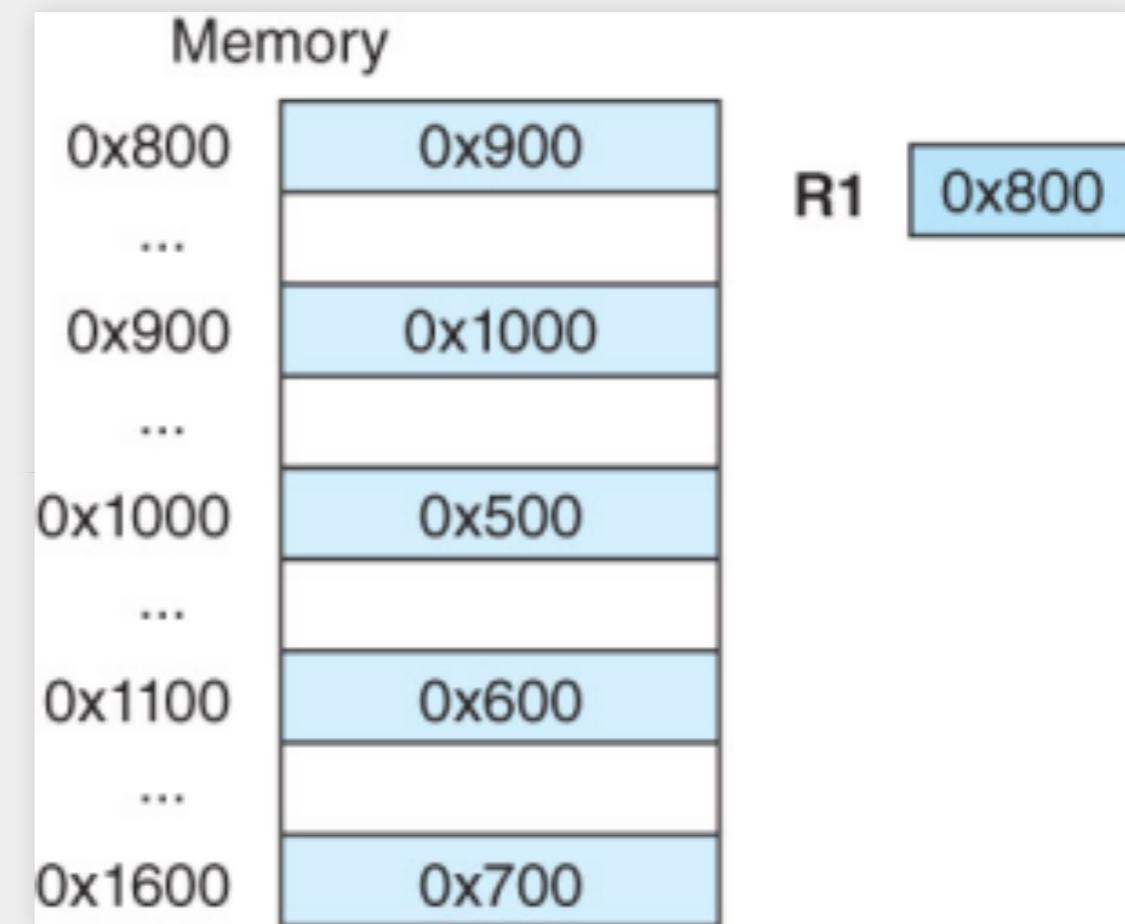
指令中的地址指向的地址保存了直接指向操作数的地址

指令中12位地址可寻址空间大小： 2^{16} (字长)

例：对于右边的内存，采用间接寻址，执行指令LOAD
0x800之后寄存器AC的值是**0x1000**

寄存器间接寻址

指令中不用地址数，而是用一个寄存器名字，例：LOAD R1。寻址地址为对应的寄存器中存储的值，寻址方法和间接寻址一致



寻址方法

变址寻址

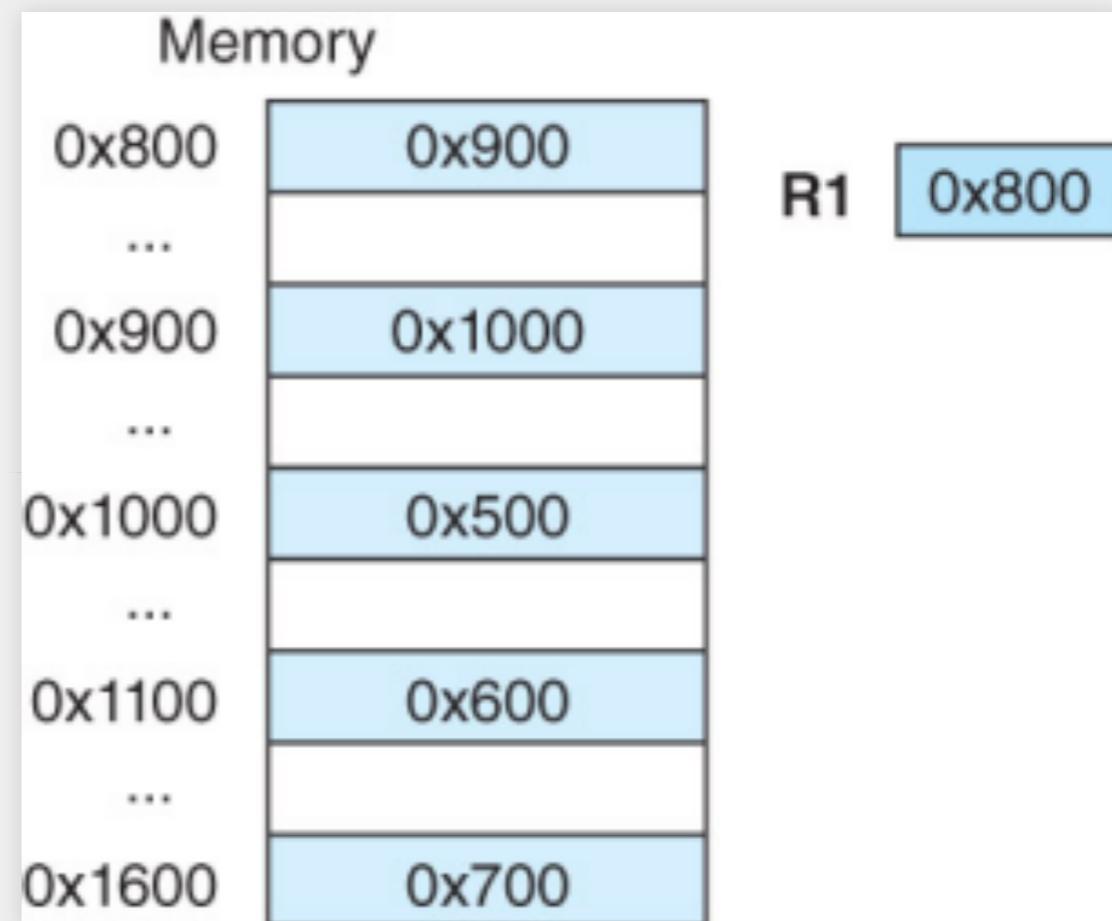
在一个特定的寄存器（称为**变址寄存器**）中保存了一个地址偏移值，通过指令中的地址加上变址寄存器中的偏移值得到实际的地址，进行**直接寻址**

基址寻址

在一个特定的寄存器（称为**基址寄存器**）中保存了一个地址偏移值，将指令中的地址作为偏移值加上基址寄存器中的地址得到实际的地址，进行**直接寻址**

指令中12位地址可寻址空间大小： $2^{16} + 2^{12}$

例：对于右边的内存，采用变址寻址和基址寻址，执行指令
LOAD 0x800之后寄存器AC的值是**0x0500**





第五讲结束

本期内容回顾

- 学习内容：教材第5章“仔细审视指令集架构”
 - 进一步加深理解指令集在整个计算机系统中的重要作用
 - 初步掌握正确解读现有指令集的方法
 - 初步了解指令集设计的方法
- 教材5.2节指令格式
 - 学习掌握指令格式设计中需要考虑的因素和现有的解决方案
- 教材5.3节指令类型
 - 学习掌握常见的指令类型
- 教材5.4节寻址
 - 学习掌握常见的寻址类型

扩展阅读



Q & A