

计算机组成和体系结构

第三讲

四川大学网络空间安全学院

2020年3月9日

封面来自Corsair.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

- 非冯-诺依曼模型
 - 单核非冯-诺依曼模型
 - 并行计算
 - 异构计算
- 数字电路基础回顾
 - 数据表示
 - 布尔代数和常见电路
- CPU基本知识和组织结构
 - 寄存器、ALU、控制单元
 - 总线、时钟、I/O子系统
 - 如何理解CPU时间计算公式

本期学习目标

- 计算机模型2
 - 内存概述
 - 体系结构
 - 交叉存储器
 - 中断及中断处理
- 计算机模型3: MARIE机器
 - 体系结构
 - 指令集
 - 寄存器传输表示
 - 汇编语言
 - 控制单元的实现
 - 现实世界中的指令集

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
RAM	Random Access Memory	随机访问内存
RTL	Register Transfer Language	寄存器传输语言
RTN	Register Transfer Notation	寄存器传输表示



计算机模型2: 内存、中断

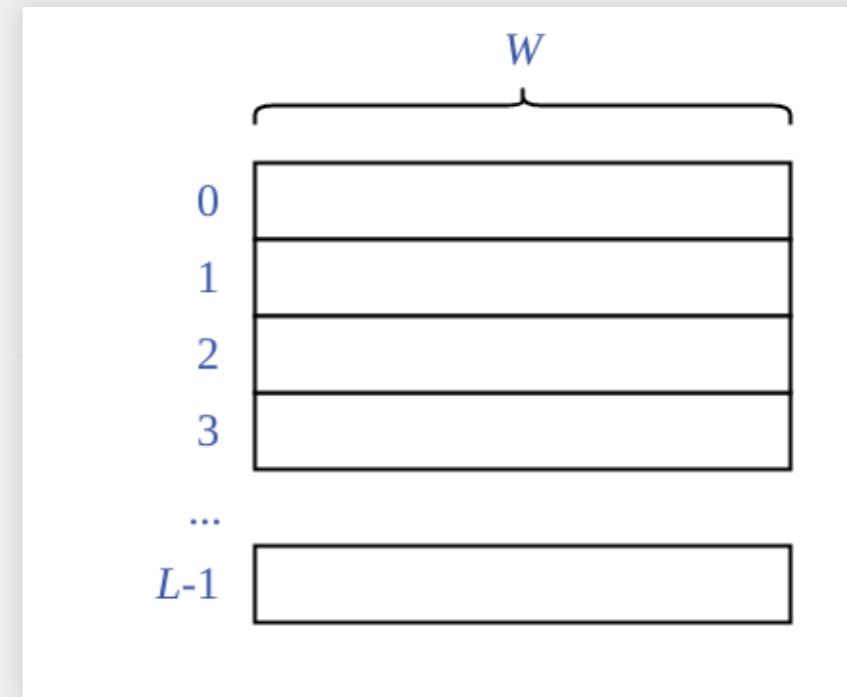
内存体系结构

内存体系结构

- 内存的逻辑表示

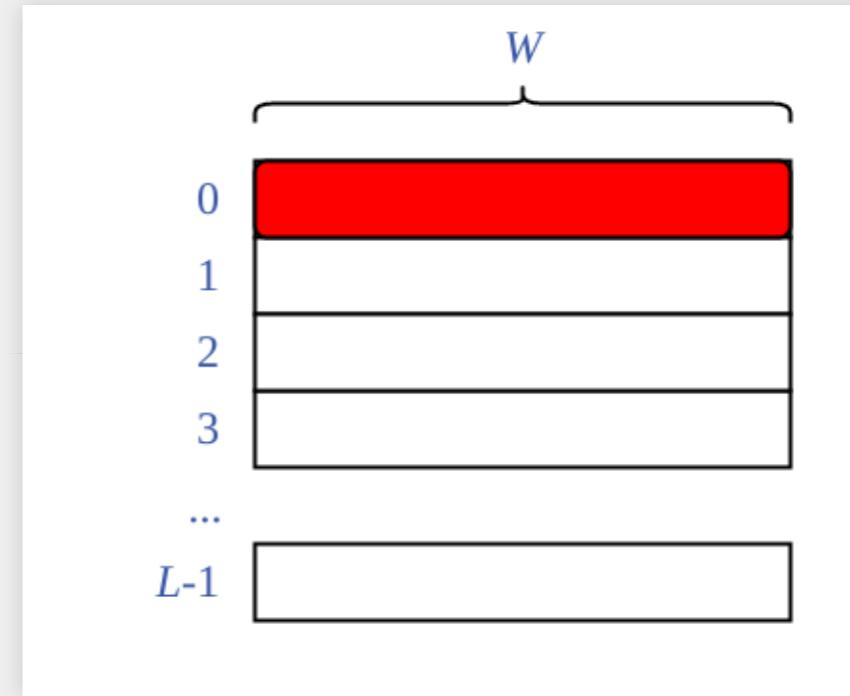
内存体系结构

- 内存的逻辑表示
 - 每个内存可以看作一个 $L \times W$ 的矩阵



内存体系结构

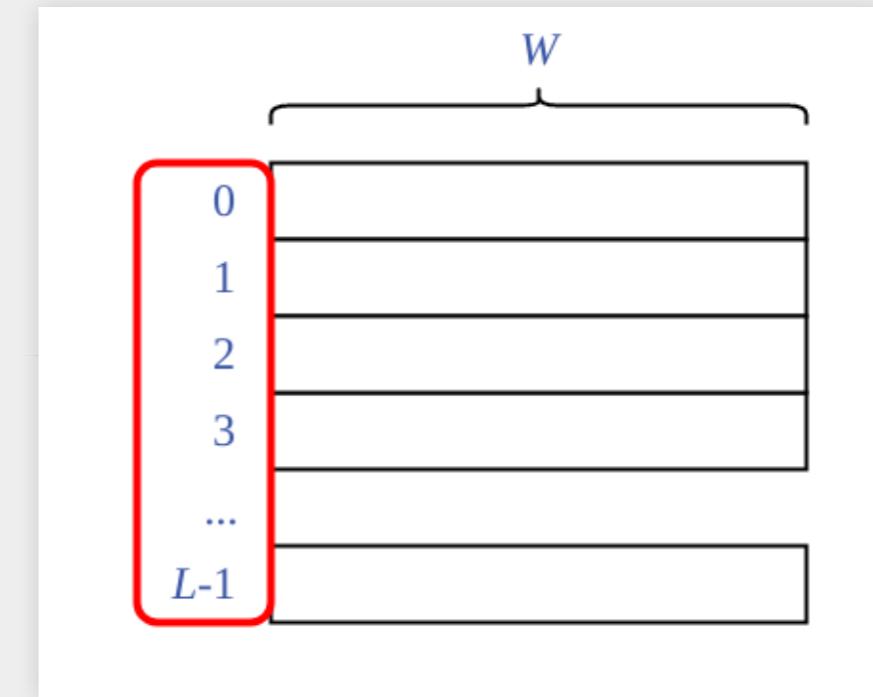
- 内存的逻辑表示
 - 每个内存可以看作一个 $L \times W$ 的矩阵
 - 每一行称为一个存储位置(memory location)，代表了一个**可寻址单元**，长度 W 即可寻址单元的大小



内存体系结构

- 内存的逻辑表示

- 每个内存可以看作一个 $L \times W$ 的矩阵
- 每一行称为一个存储位置(memory location)，代表了一个**可寻址单元**，长度 W 即可寻址单元的大小
- 每一个可寻址单元对应了一个唯一的地址编号，从0到 $L - 1$ 顺序递增。地址所需位数： $\lceil \log_2 L \rceil$ 。例：4M × 8的内存需要 $\lceil \log_2 4M \rceil = \lceil \log_2 2^{22} \rceil = 22$ 位

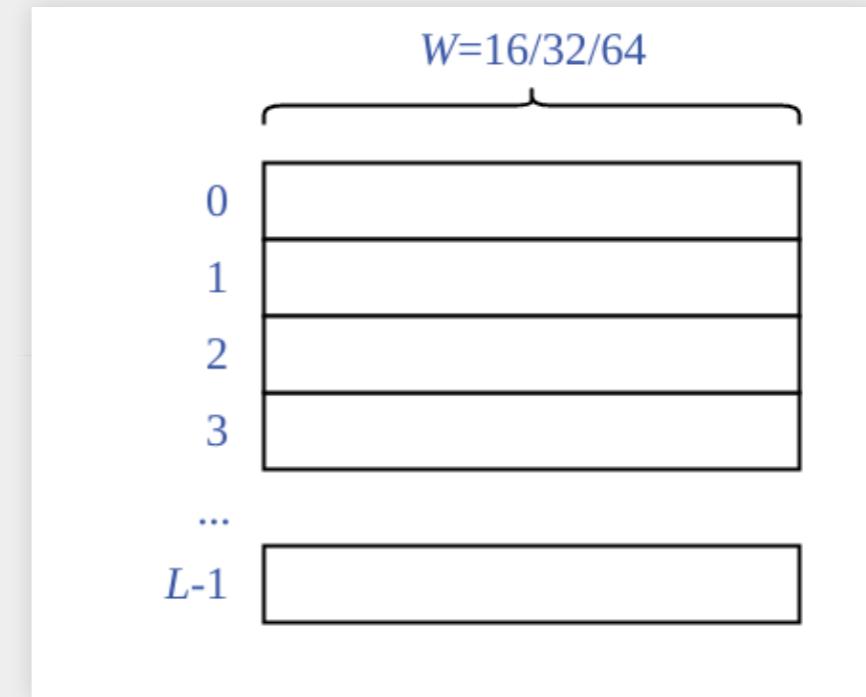


内存体系结构

- 内存的逻辑表示
 - 每个内存可以看作一个 $L \times W$ 的矩阵
 - 每一行称为一个存储位置(memory location)，代表了一个**可寻址单元**，长度 W 即可寻址单元的大小
 - 每一个可寻址单元对应了一个唯一的地址编号，从0到 $L - 1$ 顺序递增。地址所需位数： $\lceil \log_2 L \rceil$ 。例：4M × 8的内存需要 $\lceil \log_2 4M \rceil = \lceil \log_2 2^{22} \rceil = 22$ 位
- 常见寻址方式

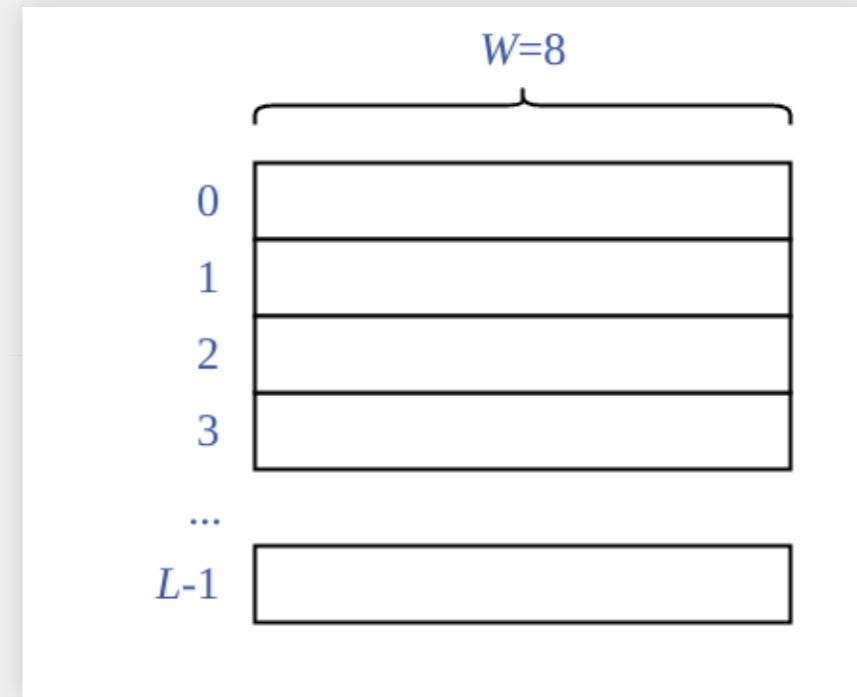
内存体系结构

- 内存的逻辑表示
 - 每个内存可以看作一个 $L \times W$ 的矩阵
 - 每一行称为一个存储位置(memory location)，代表了一个**可寻址单元**，长度 W 即可寻址单元的大小
 - 每一个可寻址单元对应了一个唯一的地址编号，从0到 $L - 1$ 顺序递增。地址所需位数： $\lceil \log_2 L \rceil$ 。例：4M × 8的内存需要 $\lceil \log_2 4M \rceil = \lceil \log_2 2^{22} \rceil = 22$ 位
- 常见寻址方式
 - 按字寻址(word-addressable)：可寻址单元大小为一个字(常见为16位、32位及64位)



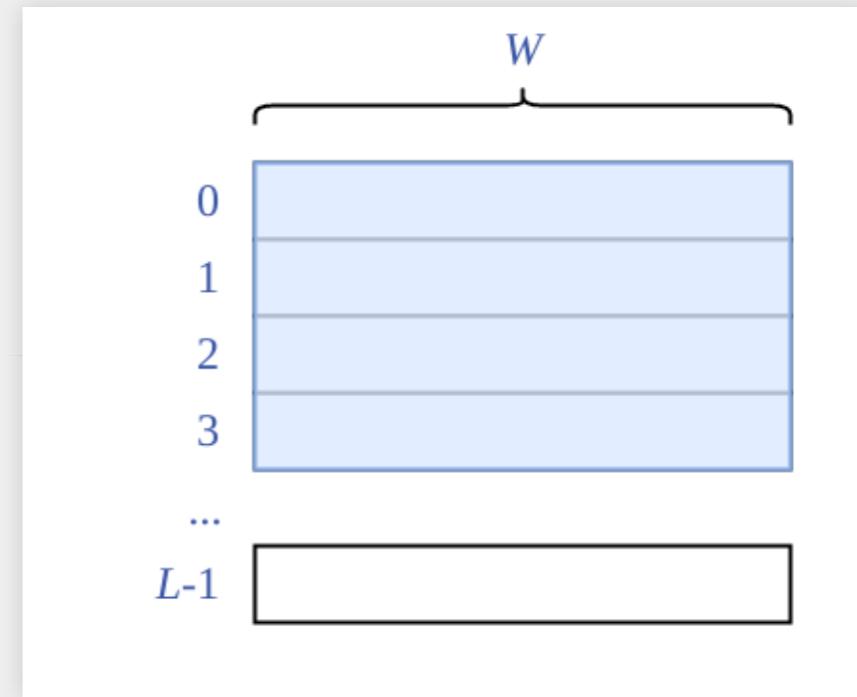
内存体系结构

- 内存的逻辑表示
 - 每个内存可以看作一个 $L \times W$ 的矩阵
 - 每一行称为一个存储位置(memory location)，代表了一个**可寻址单元**，长度 W 即可寻址单元的大小
 - 每一个可寻址单元对应了一个唯一的地址编号，从0到 $L - 1$ 顺序递增。地址所需位数： $\lceil \log_2 L \rceil$ 。例：4M × 8的内存需要 $\lceil \log_2 4M \rceil = \lceil \log_2 2^{22} \rceil = 22$ 位
- 常见寻址方式
 - 按字寻址(word-addressable)：可寻址单元大小为一个字(常见为16位、32位及64位)
 - 按字节寻址(byte-addressable)：可寻址单元大小为一个字节(8位)



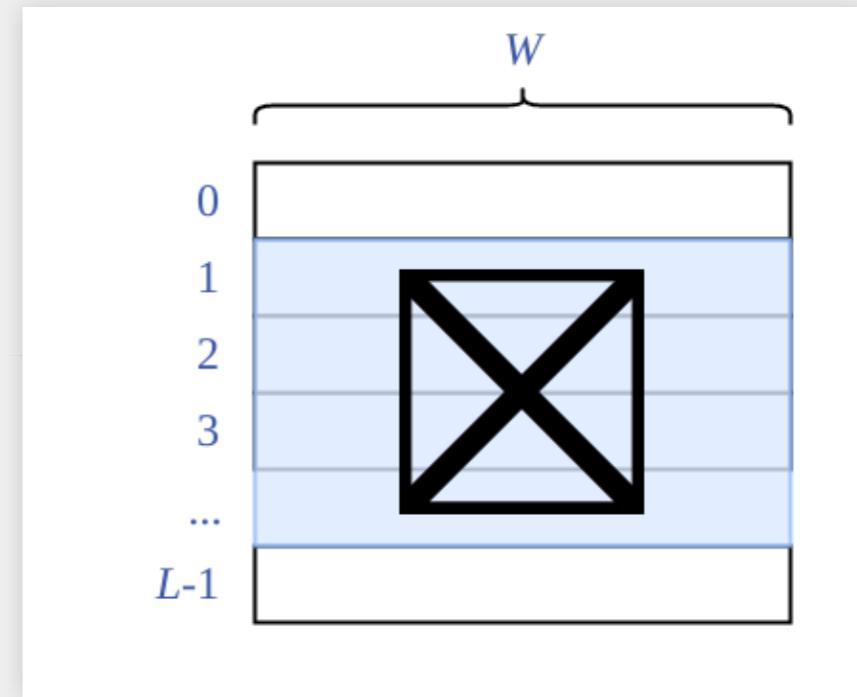
内存体系结构

- 内存的逻辑表示
 - 每个内存可以看作一个 $L \times W$ 的矩阵
 - 每一行称为一个存储位置(memory location)，代表了一个**可寻址单元**，长度 W 即可寻址单元的大小
 - 每一个可寻址单元对应了一个唯一的地址编号，从0到 $L - 1$ 顺序递增。地址所需位数： $\lceil \log_2 L \rceil$ 。例：4M × 8的内存需要 $\lceil \log_2 4M \rceil = \lceil \log_2 2^{22} \rceil = 22$ 位
- 常见寻址方式
 - 按字寻址(word-addressable)：可寻址单元大小为一个字(常见为16位、32位及64位)
 - 按字节寻址(byte-addressable)：可寻址单元大小为一个字节(8位)
 - 按字节寻址方式寻址一个字：用最低位字节的地址表示



内存体系结构

- 内存的逻辑表示
 - 每个内存可以看作一个 $L \times W$ 的矩阵
 - 每一行称为一个存储位置(memory location)，代表了一个**可寻址单元**，长度 W 即可寻址单元的大小
 - 每一个可寻址单元对应了一个唯一的地址编号，从0到 $L - 1$ 顺序递增。地址所需位数： $\lceil \log_2 L \rceil$ 。例：4M × 8的内存需要 $\lceil \log_2 4M \rceil = \lceil \log_2 2^{22} \rceil = 22$ 位
- 常见寻址方式
 - 按字寻址(word-addressable)：可寻址单元大小为一个字(常见为16位、32位及64位)
 - 按字节寻址(byte-addressable)：可寻址单元大小为一个字节(8位)
 - 按字节寻址方式寻址一个字：用最低位字节的地址表示
 - 对齐(alignment)问题：当字长为 2^k ($k \leq 3$)，采用按字节寻址方式，字的地址满足 $x \bmod 2^{k-3} = 0$ 。例：字长为32时，采用按字节寻址，一个字的地址为4的倍数



交叉存储器

为什么要使用交叉存储器：

交叉存储器

为什么要使用交叉存储器：用小的RAM芯片满足大的存储

交叉存储器

为什么要使用交叉存储器：用小的RAM芯片满足大的存储

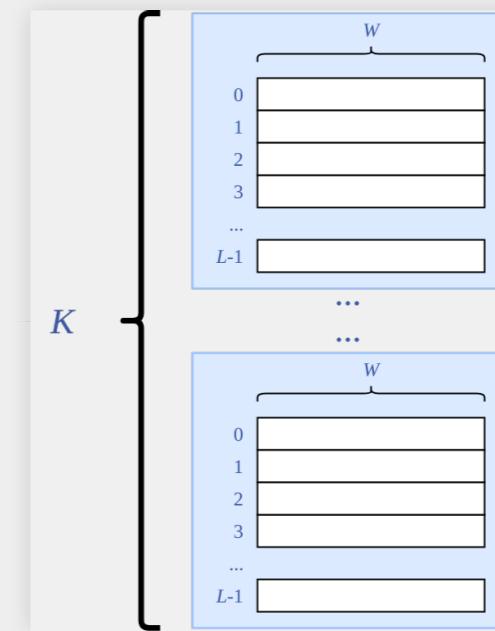
- 如何连接

交叉存储器

为什么要使用交叉存储器：用小的RAM芯片满足大的存储

- 如何连接

- 垂直方向： K 个 $L \times W$ 的RAM芯片构成一个 $KL \times W$ 的存储器

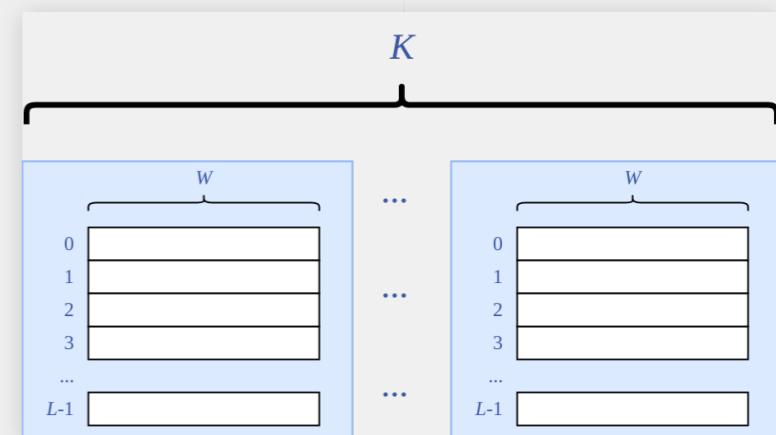
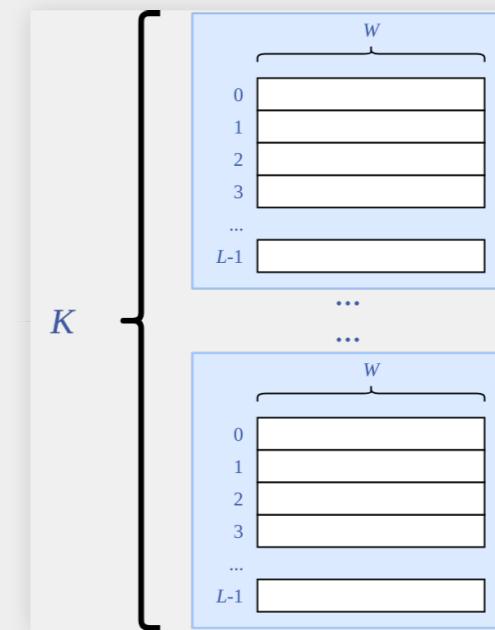


交叉存储器

为什么要使用交叉存储器：用小的RAM芯片满足大的存储

- 如何连接

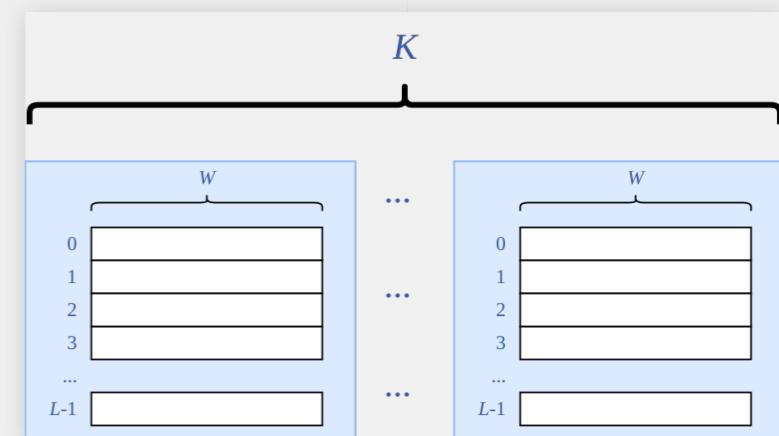
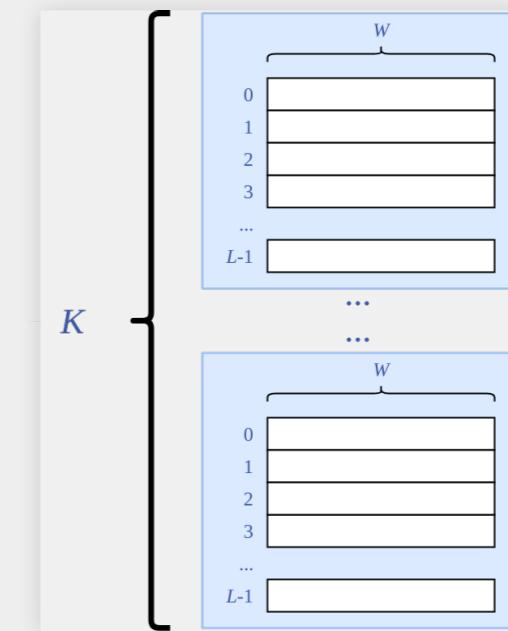
- 垂直方向： K 个 $L \times W$ 的RAM芯片构成一个 $KL \times W$ 的存储器
- 水平方向： K 个 $L \times W$ 的RAM芯片构成一个 $L \times KW$ 的存储器



交叉存储器

为什么要使用交叉存储器：用小的RAM芯片满足大的存储

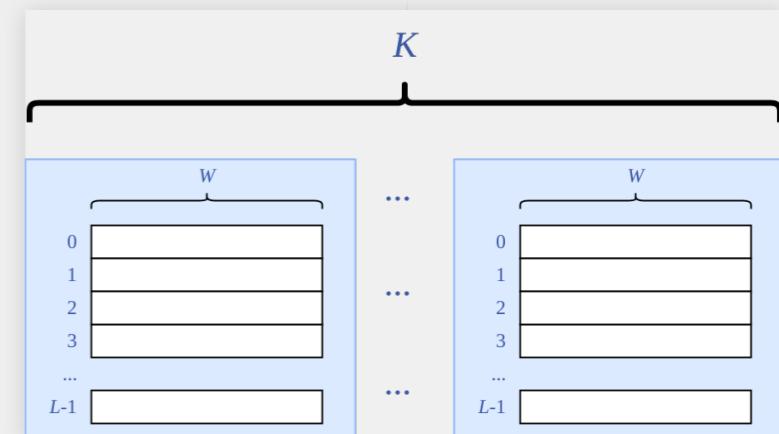
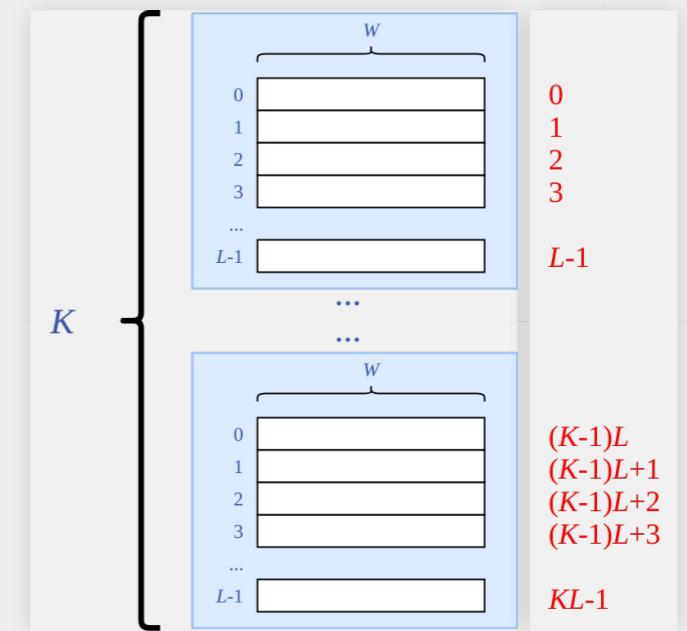
- 如何连接
 - 垂直方向： K 个 $L \times W$ 的RAM芯片构成一个 $KL \times W$ 的存储器
 - 水平方向： K 个 $L \times W$ 的RAM芯片构成一个 $L \times KW$ 的存储器
- 如何寻址(假设组合前后寻址方式不变)



交叉存储器

为什么要使用交叉存储器：用小的RAM芯片满足大的存储

- 如何连接
 - 垂直方向： K 个 $L \times W$ 的RAM芯片构成一个 $KL \times W$ 的存储器
 - 水平方向： K 个 $L \times W$ 的RAM芯片构成一个 $L \times KW$ 的存储器
- 如何寻址(假设组合前后寻址方式不变)
 - 垂直方向：和单个RAM芯片寻址方法一致，区别是长度变成 KL ，需要 $\lceil \log_2 (KL) \rceil$ 位地址。第 k 个芯片第 l 行对应的地址是 $((k - 1) \times L + (l - 1))$



交叉存储器

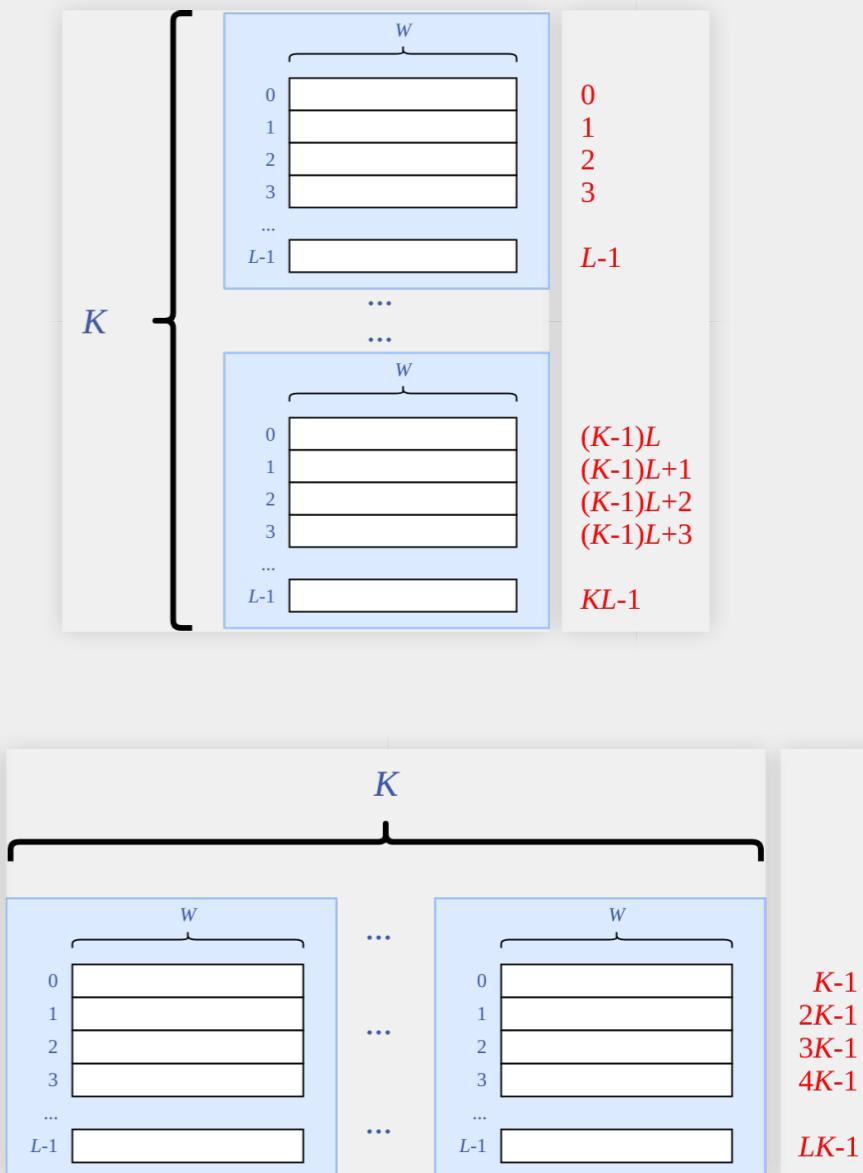
为什么要使用交叉存储器：用小的RAM芯片满足大的存储

- 如何连接

- 垂直方向： K 个 $L \times W$ 的RAM芯片构成一个 $KL \times W$ 的存储器
- 水平方向： K 个 $L \times W$ 的RAM芯片构成一个 $L \times KW$ 的存储器

- 如何寻址(假设组合前后寻址方式不变)

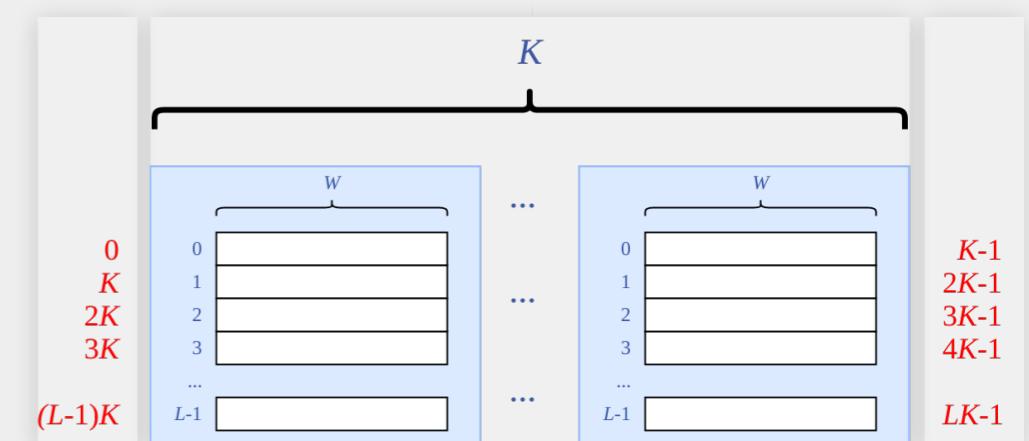
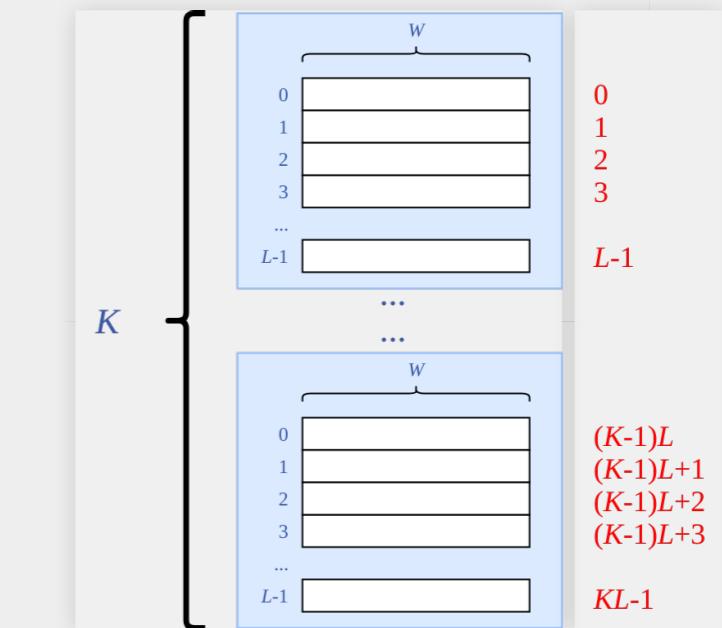
- 垂直方向：和单个RAM芯片寻址方法一致，区别是长度变成 KL ，需要 $\lceil \log_2 (KL) \rceil$ 位地址。第 k 个芯片第 l 行对应的地址是 $((k - 1) \times L + (l - 1))$
- 水平方向：可以看作两阶段的寻址，首先假设采用了长度为 KW 的按字寻址(共 L 个)，然后再从长度为 KW 的字中选出长度为 W 的字，需要地址数为 $\lceil \log_2 L \rceil + \lceil \log_2 K \rceil$ 。第 k 个芯片第 l 行对应的地址是 $(l - 1) * K + (k - 1)$



交叉存储器

为什么要使用交叉存储器：用小的RAM芯片满足大的存储

- 如何连接
 - 垂直方向： K 个 $L \times W$ 的RAM芯片构成一个 $KL \times W$ 的存储器
 - 水平方向： K 个 $L \times W$ 的RAM芯片构成一个 $L \times KW$ 的存储器
- 如何寻址(假设组合前后寻址方式不变)
 - 垂直方向：和单个RAM芯片寻址方法一致，区别是长度变成 KL ，需要 $\lceil \log_2 (KL) \rceil$ 位地址。第 k 个芯片第 l 行对应的地址是 $((k - 1) \times L + (l - 1))$
 - 水平方向：可以看作两阶段的寻址，首先假设采用了长度为 KW 的按字寻址(共 L 个)，然后再从长度为 KW 的字中选出长度为 W 的字，需要地址数为 $\lceil \log_2 L \rceil + \lceil \log_2 K \rceil$ 。第 k 个芯片第 l 行对应的地址是 $(l - 1) * K + (k - 1)$
- 实际中因为 K 和 L 都是2的次幂，模块号和行号(偏移值)从0开始编址，因此



交叉存储器

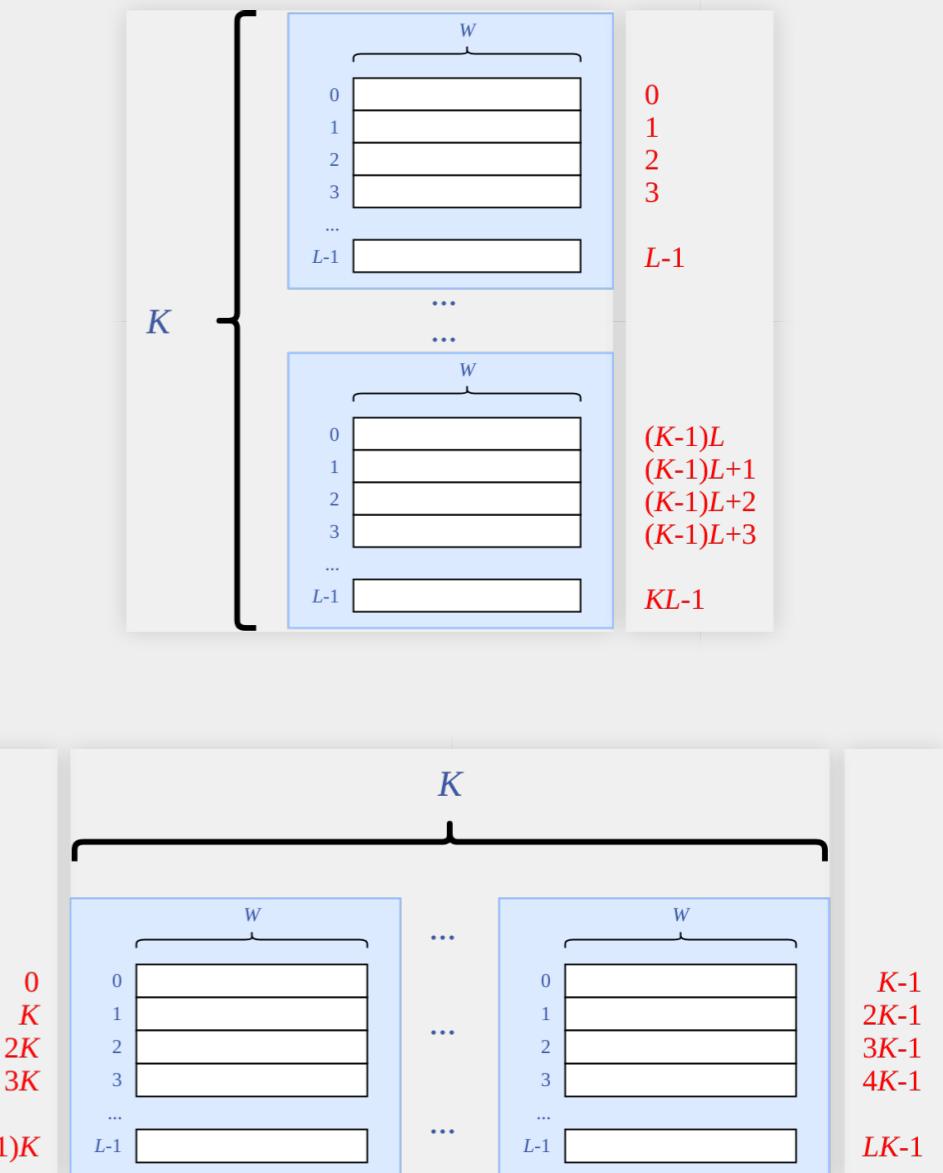
为什么要使用交叉存储器：用小的RAM芯片满足大的存储

- 如何连接
 - 垂直方向： K 个 $L \times W$ 的RAM芯片构成一个 $KL \times W$ 的存储器
 - 水平方向： K 个 $L \times W$ 的RAM芯片构成一个 $L \times KW$ 的存储器
- 如何寻址(假设组合前后寻址方式不变)
 - 垂直方向：和单个RAM芯片寻址方法一致，区别是长度变成 KL ，需要 $\lceil \log_2 (KL) \rceil$ 位地址。第 k 个芯片第 l 行对应的地址是 $((k - 1) \times L + (l - 1))$
 - 水平方向：可以看作两阶段的寻址，首先假设采用了长度为 KW 的按字寻址(共 L 个)，然后再从长度为 KW 的字中选出长度为 W 的字，需要地址数为 $\lceil \log_2 L \rceil + \lceil \log_2 K \rceil$ 。第 k 个芯片第 l 行对应的地址是 $(l - 1) * K + (k - 1)$
- 实际中因为 K 和 L 都是2的次幂，模块号和行号(偏移值)从0开始编址，因此

- 垂直方向多模块存储编址(**高位交叉编址**)：



$\lceil \log_2 K \rceil$ 位模块号(高位)+ $\lceil \log_2 L \rceil$ 位偏移值(低位)



交叉存储器

为什么要使用交叉存储器：用小的RAM芯片满足大的存储

- 如何连接

- 垂直方向： K 个 $L \times W$ 的RAM芯片构成一个 $KL \times W$ 的存储器
- 水平方向： K 个 $L \times W$ 的RAM芯片构成一个 $L \times KW$ 的存储器

- 如何寻址(假设组合前后寻址方式不变)

- 垂直方向：和单个RAM芯片寻址方法一致，区别是长度变成 KL ，需要 $\lceil \log_2 (KL) \rceil$ 位地址。第 k 个芯片第 l 行对应的地址是 $((k - 1) \times L + (l - 1))$
- 水平方向：可以看作两阶段的寻址，首先假设采用了长度为 KW 的按字寻址(共 L 个)，然后再从长度为 KW 的字中选出长度为 W 的字，需要地址数为 $\lceil \log_2 L \rceil + \lceil \log_2 K \rceil$ 。第 k 个芯片第 l 行对应的地址是 $(l - 1) * K + (k - 1)$

- 实际中因为 K 和 L 都是2的次幂，模块号和行号(偏移值)从0开始编址，因此

- 垂直方向多模块存储编址(**高位交叉编址**)：

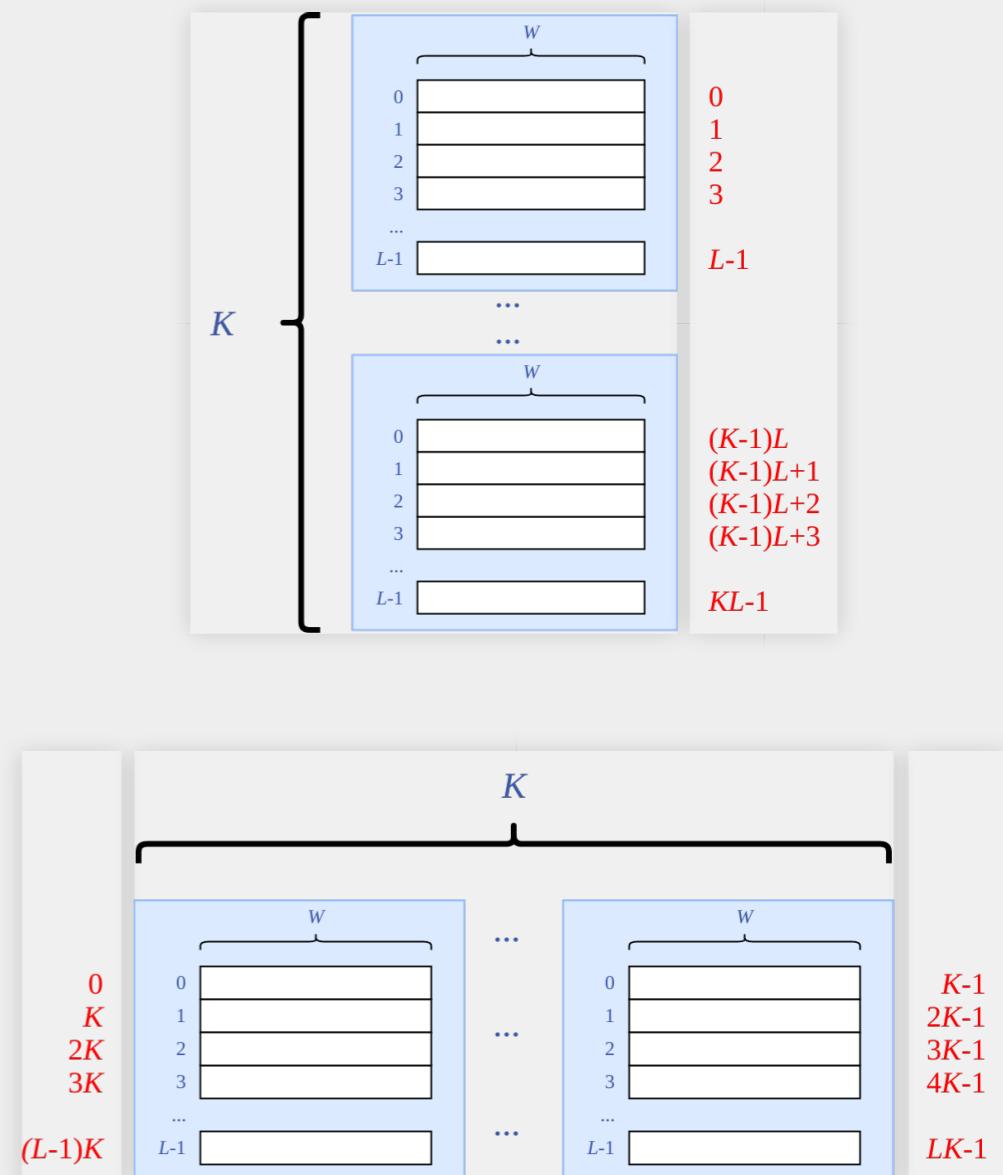


$\lceil \log_2 K \rceil$ 位模块号(高位)+ $\lceil \log_2 L \rceil$ 位偏移值(低位)

- 水平方向多模块存储编址(**低位交叉编址**)：

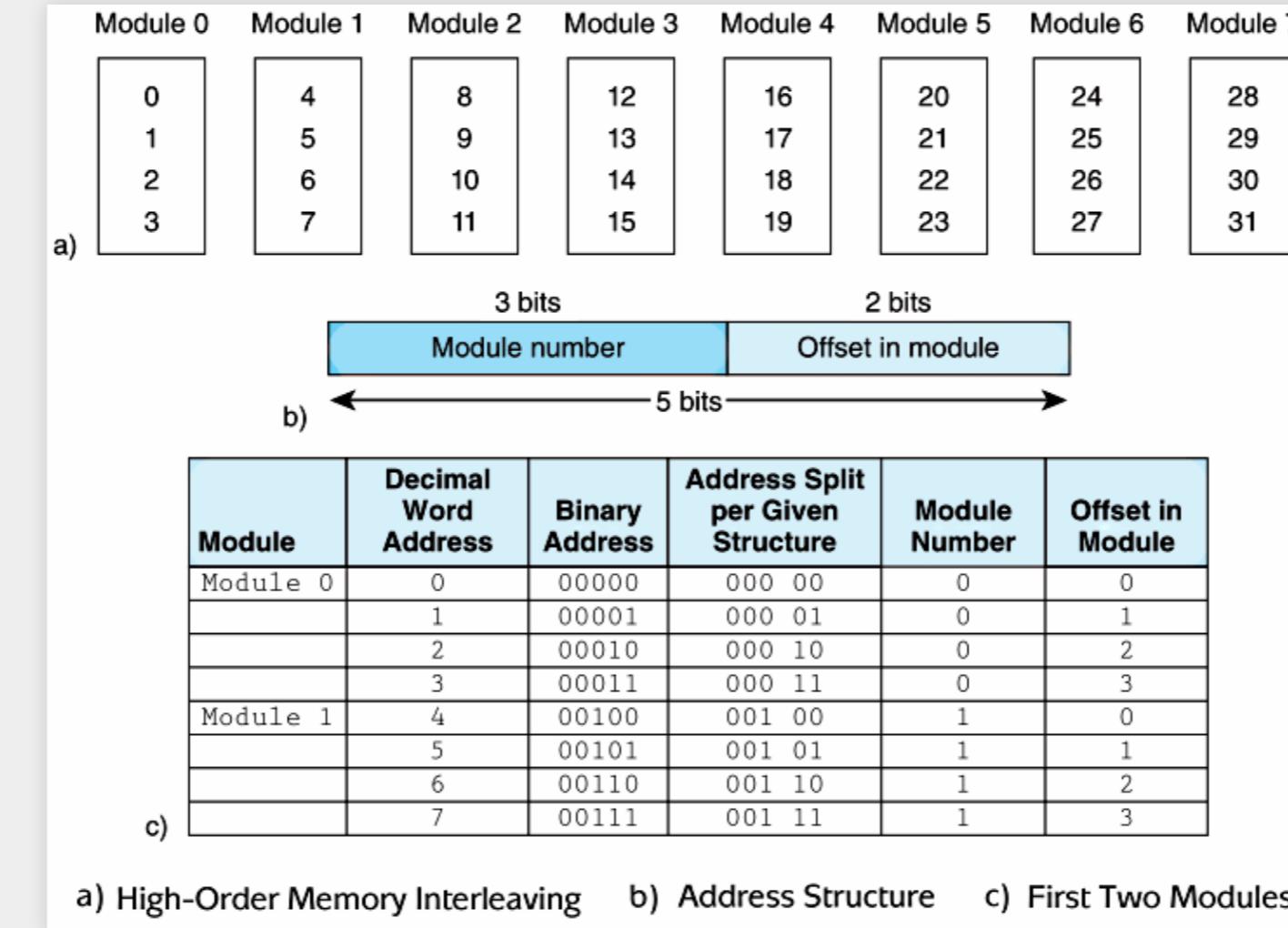


$\lceil \log_2 L \rceil$ 位偏移值(高位)+ $\lceil \log_2 K \rceil$ 位模块号(低位)



交叉编址示例

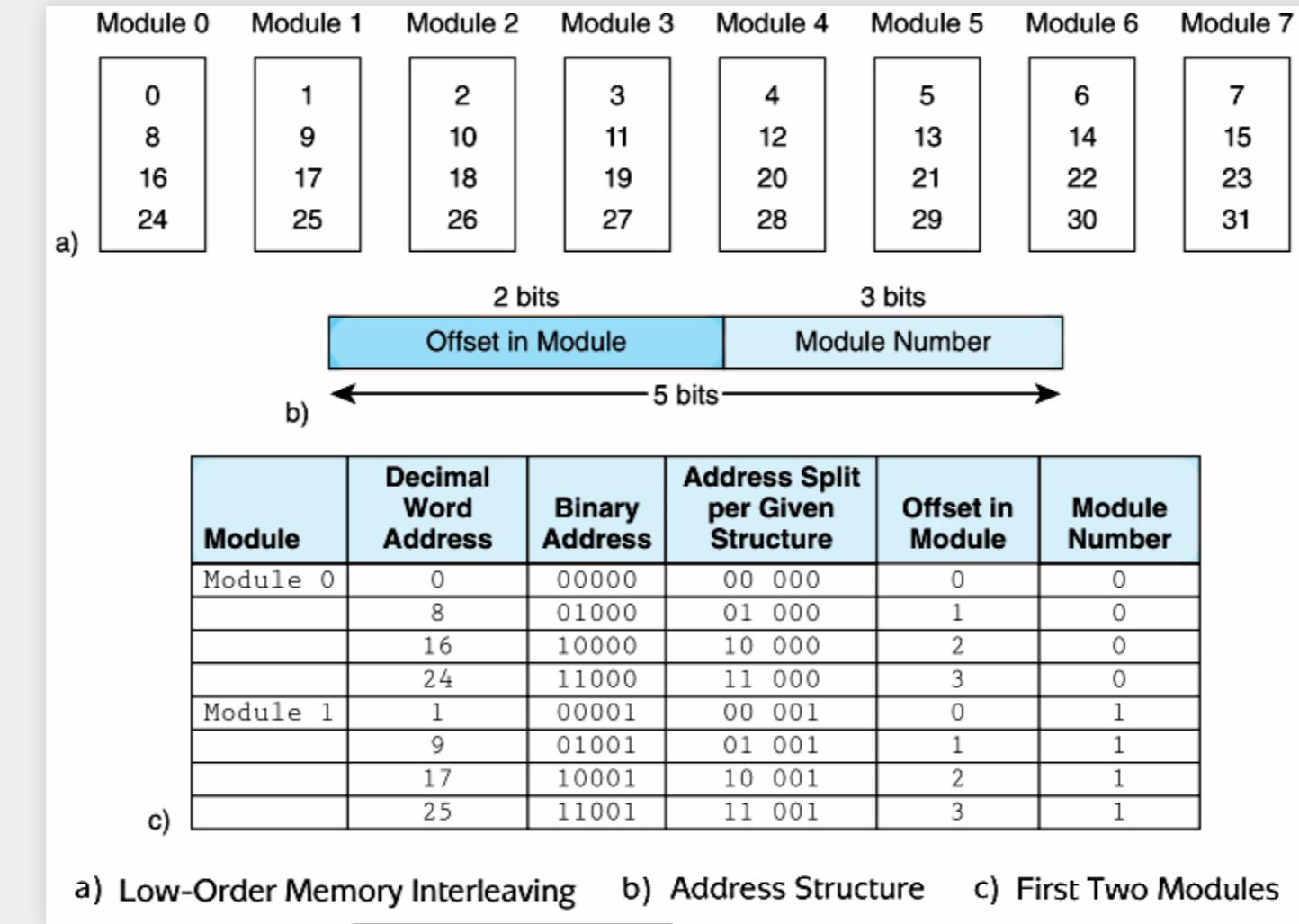
交叉编址示例



高位交叉编址(High-order Interleaving), $K = 8, L = 4$

图片来源：Linda Null, Julia Lobur, *The Essentials of Computer Organization and Architecture*, 4th Edition

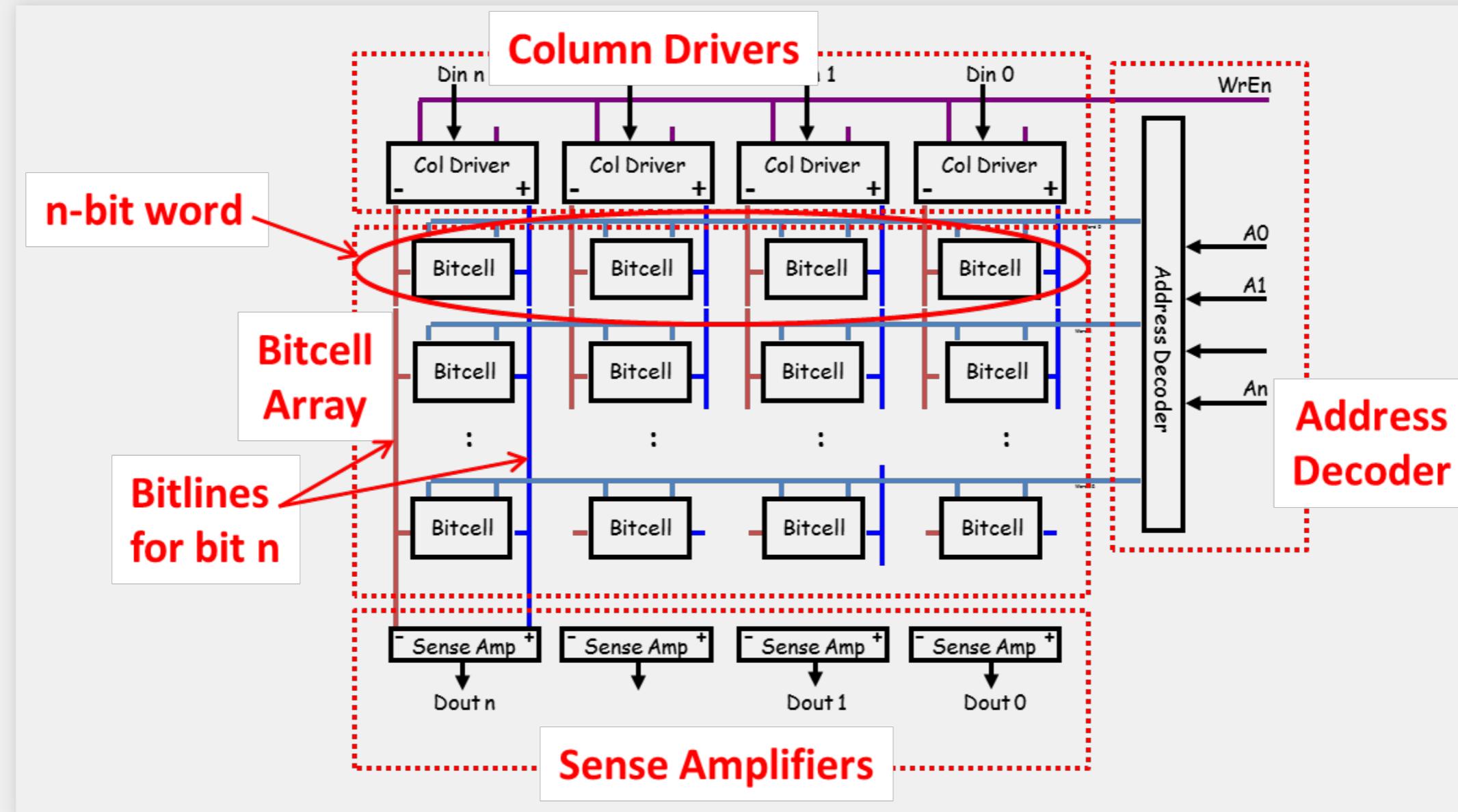
交叉编址示例



低位交叉编址(Low-order Interleaving), $K = 8, L = 4$

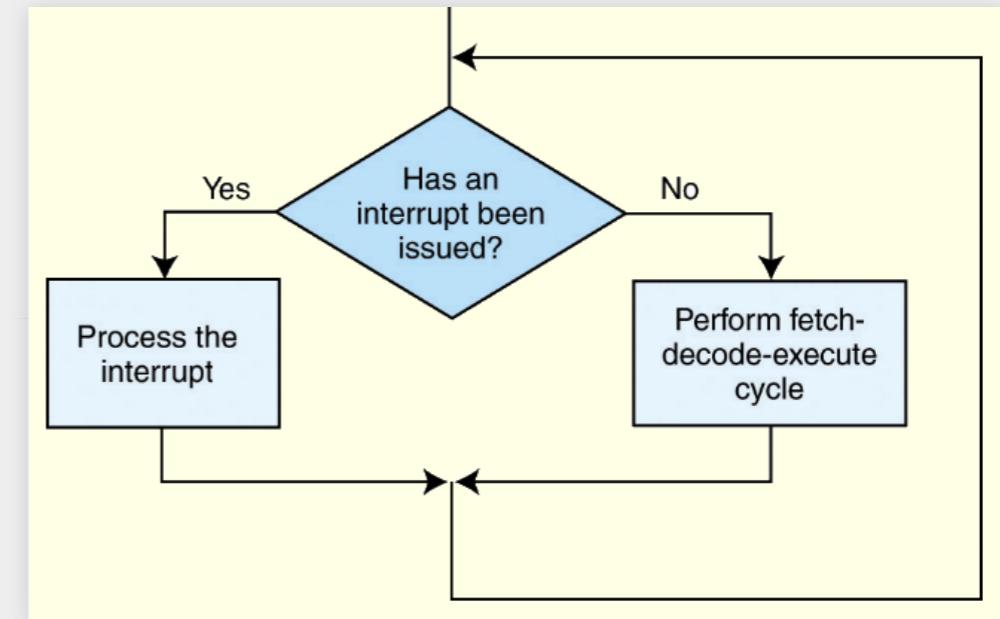
图片来源：Linda Null, Julia Lobur, *The Essentials of Computer Organization and Architecture*, 4th Edition

内存的电路实现*



中断

- 中断的作用：改变或停止系统中程序执行的事件
- 中断既可以由硬件触发也可以由软件触发。由软件触发的中断一般也称为陷阱(trap)或异常(exception)
- 常见的中断触发事件
 - I/O请求、算数错误、算数溢出、硬件故障、用户定义中断点、页面错误、非法指令等

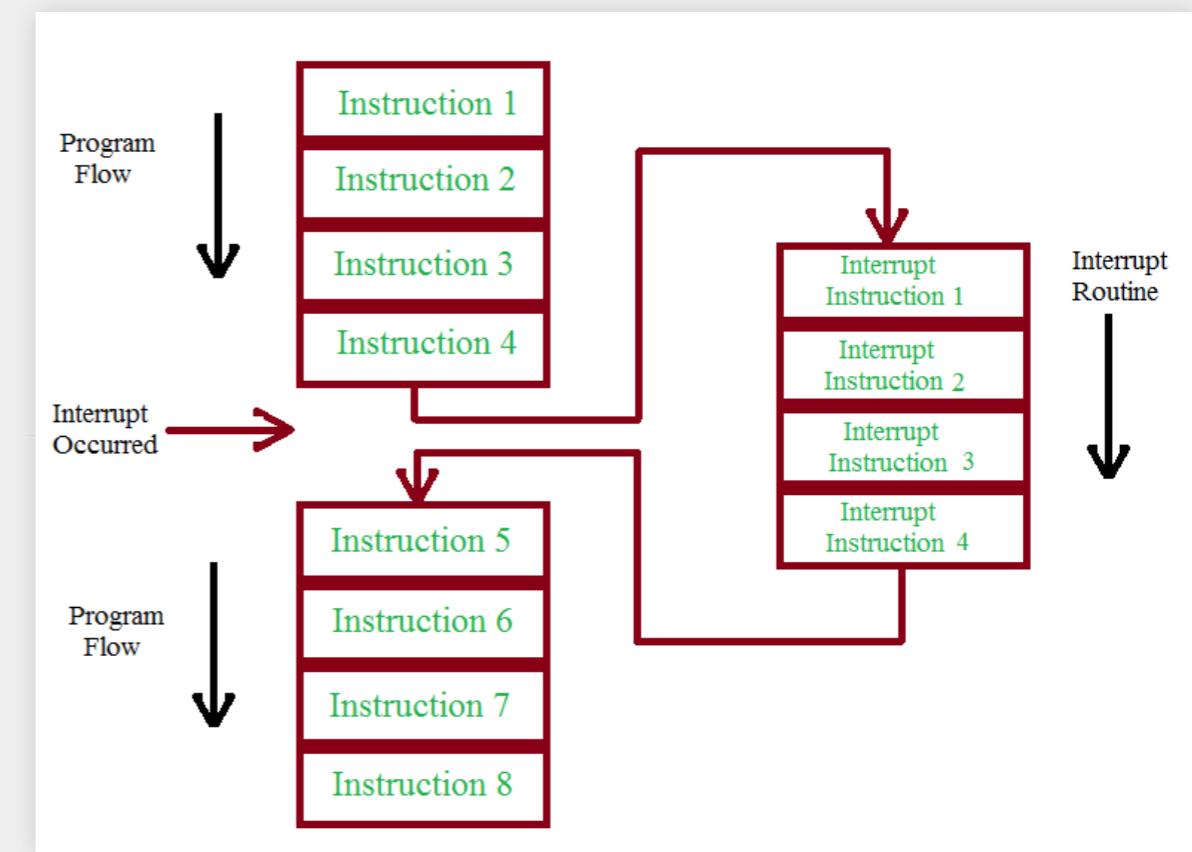


图片来源：Linda Null, Julia Lobur, *The Essentials of Computer Organization and Architectu*

中断处理

中断处理

- 中断的处理是通过对对应的**中断处理程序**(Interrupt Service Routine, ISR)来实现的



图片来源：<https://www.electronicshub.org/arm-interrupt-tutorial/>

中断处理

- 中断的处理是通过对对应的**中断处理程序**(Interrupt Service Routine, ISR)来实现的
- 中断的类型用中断号表示，中断号以及对应的中断处理程序存储在**中断向量表**中

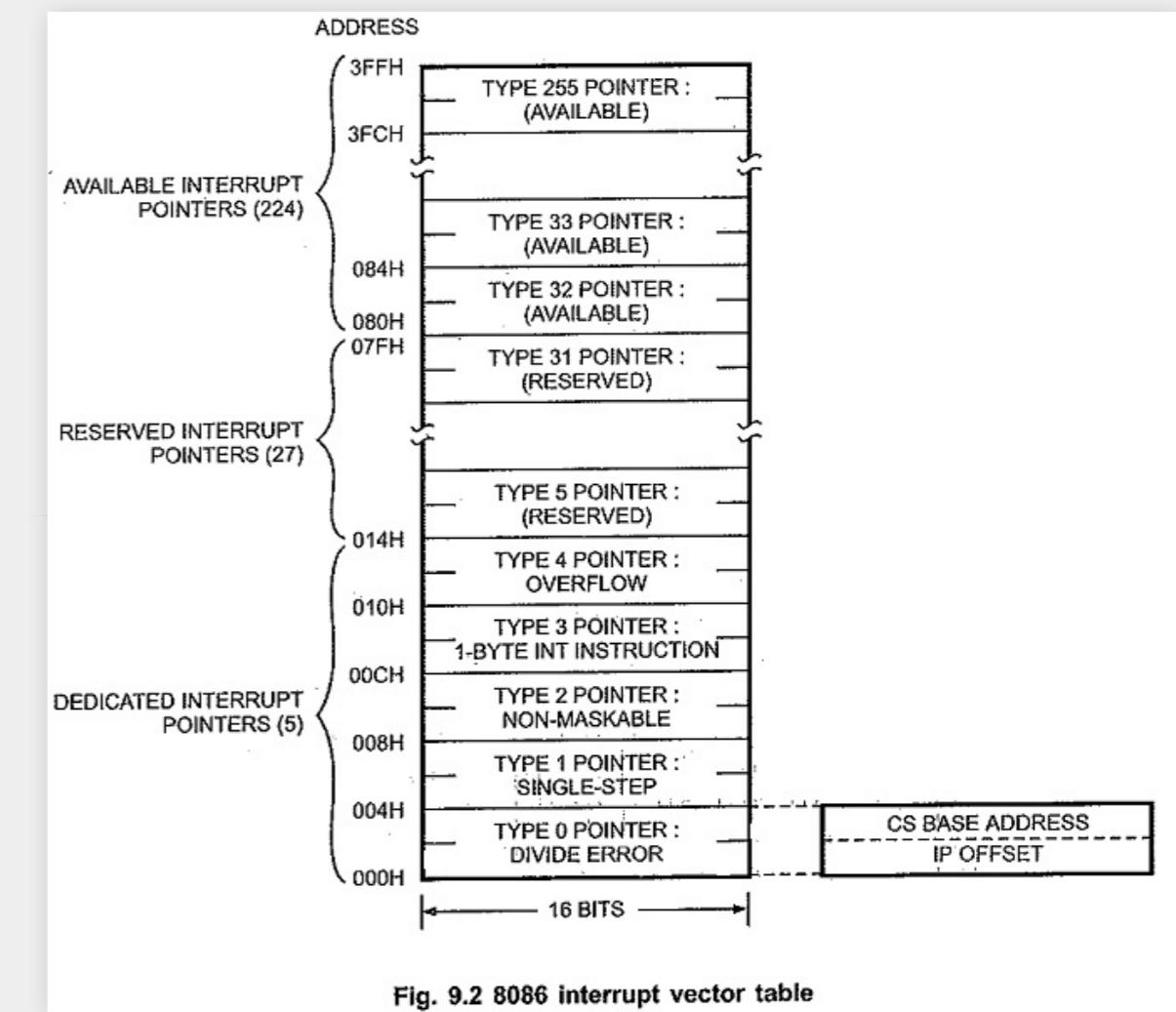
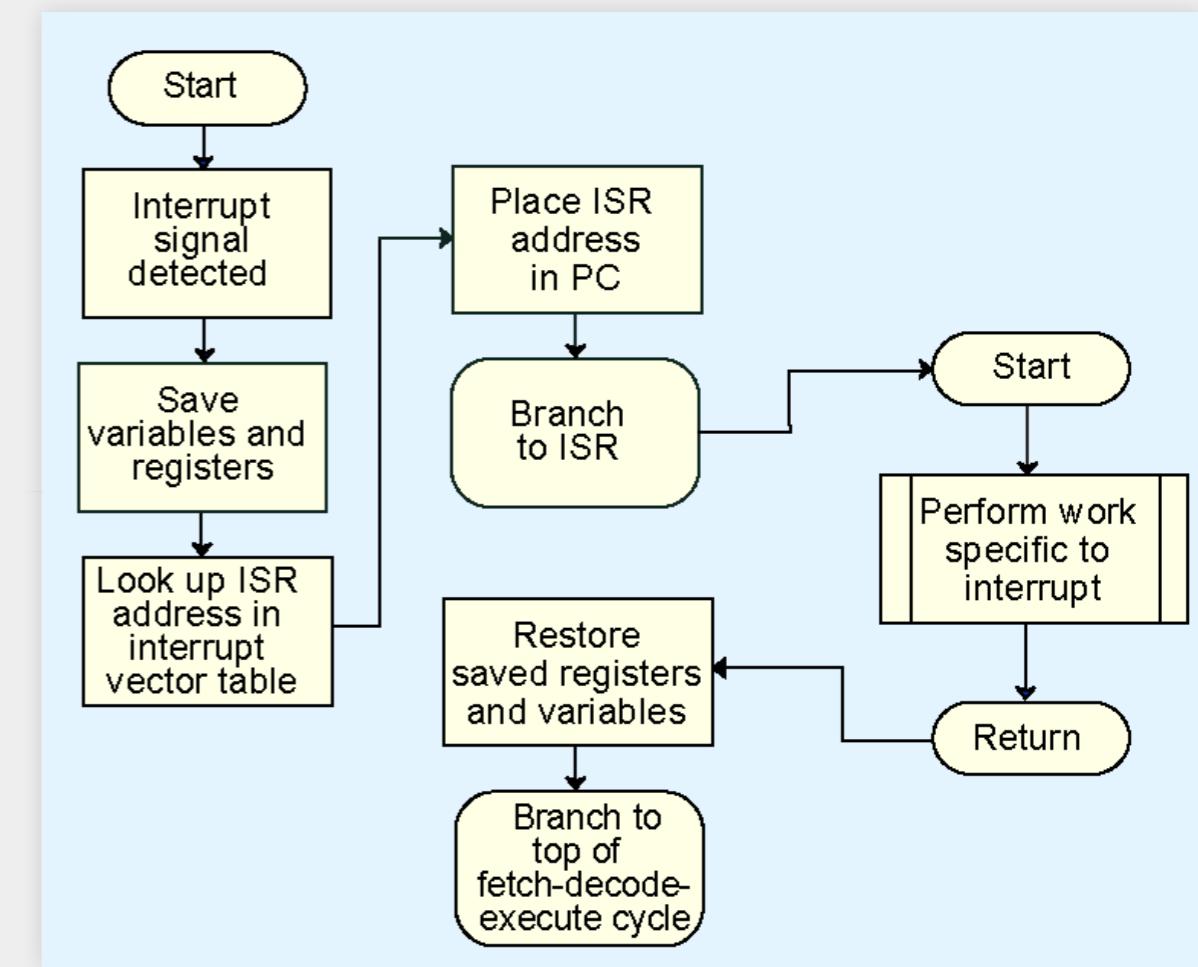


Fig. 9.2 8086 interrupt vector table

图片来源：<https://www.eeeguide.com/8086-interrupt/>

中断处理

- 中断的处理是通过对对应的**中断处理程序**(Interrupt Service Routine, ISR)来实现的
- 中断的类型用中断号表示，中断号以及对应的中断处理程序存储在**中断向量表**中
- 完整的中断处理过程



图片来源：Linda Null, Julia Lobur, *The Essentials of Computer Organization and Architectu*

中断处理

- 中断的处理是通过对对应的**中断处理程序**(Interrupt Service Routine, ISR)来实现的
- 中断的类型用中断号表示，中断号以及对应的中断处理程序存储在**中断向量表**中
- 完整的中断处理过程
- 中断的屏蔽(Mask)：在有些程序执行过程中（例如一个中断处理程序）不希望被中断，因此可以屏蔽掉另外的一些中断

计算机模型3: MARIE体系结构

MARIE

MARIE = the Machine Architecture that is Really Intuitive and Easy

MARIE模型的主要参数：

- 采用二进制编码，带符号整数采用2的补码表示
- 采用存储程序架构，采用16位固定字长的数据和指令
- 内存采用按字寻址，可寻址4K个存储位置
- 16位指令包括4位操作码(opcode)和12位地址
- 包含一个16位的算数单元
- 有7个寄存器

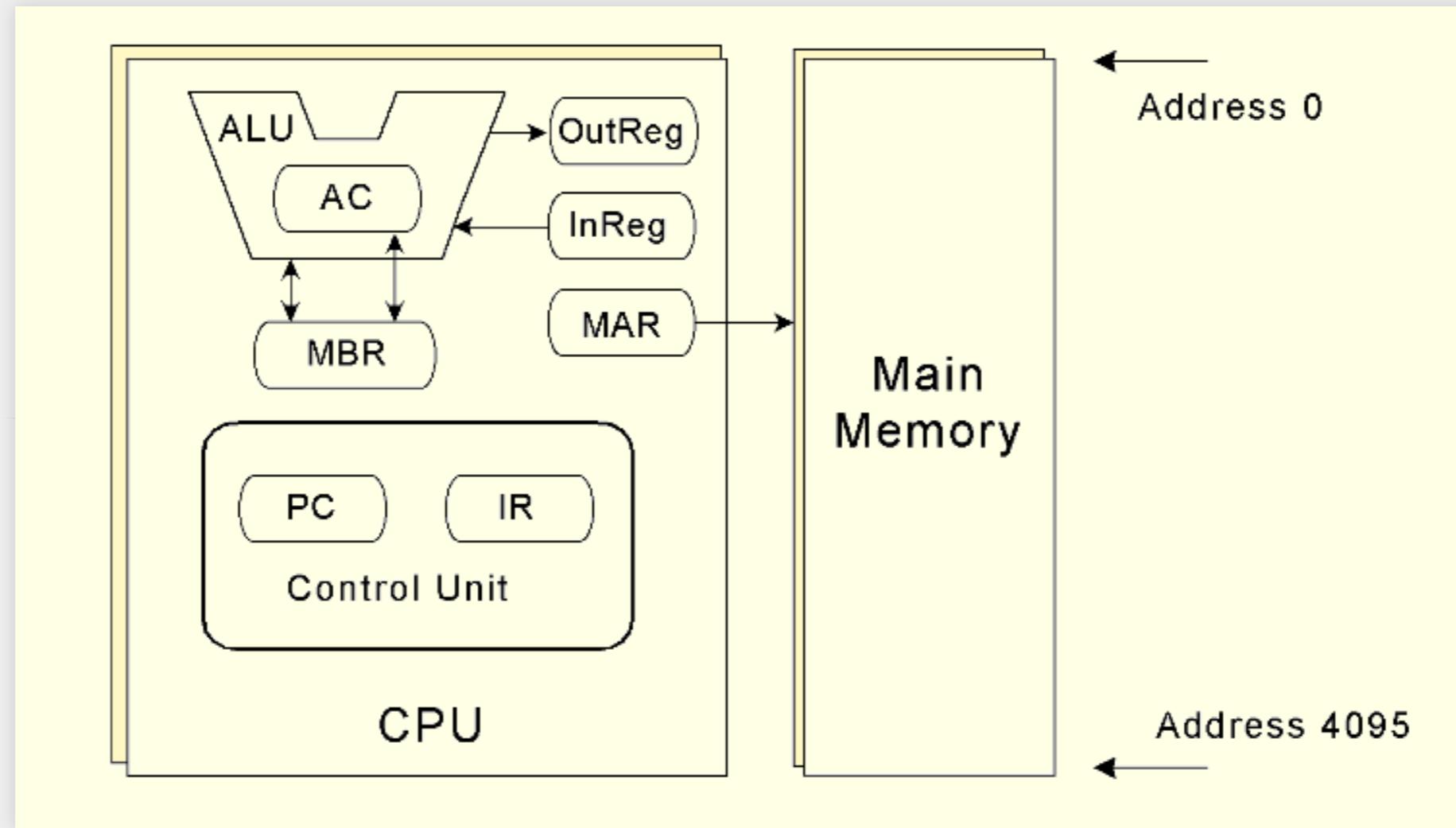
MARIE

MARIE = the Machine Architecture that is Really Intuitive and Easy

MARIE模型的主要参数：

- 采用二进制编码，带符号整数采用2的补码表示
- 采用存储程序架构，采用16位固定字长的数据和指令
- 内存采用按字寻址，可寻址4K个存储位置 **为什么是4K?**
- 16位指令包括4位操作码(opcode)和12位地址
- 包含一个16位的算数单元
- 有7个寄存器

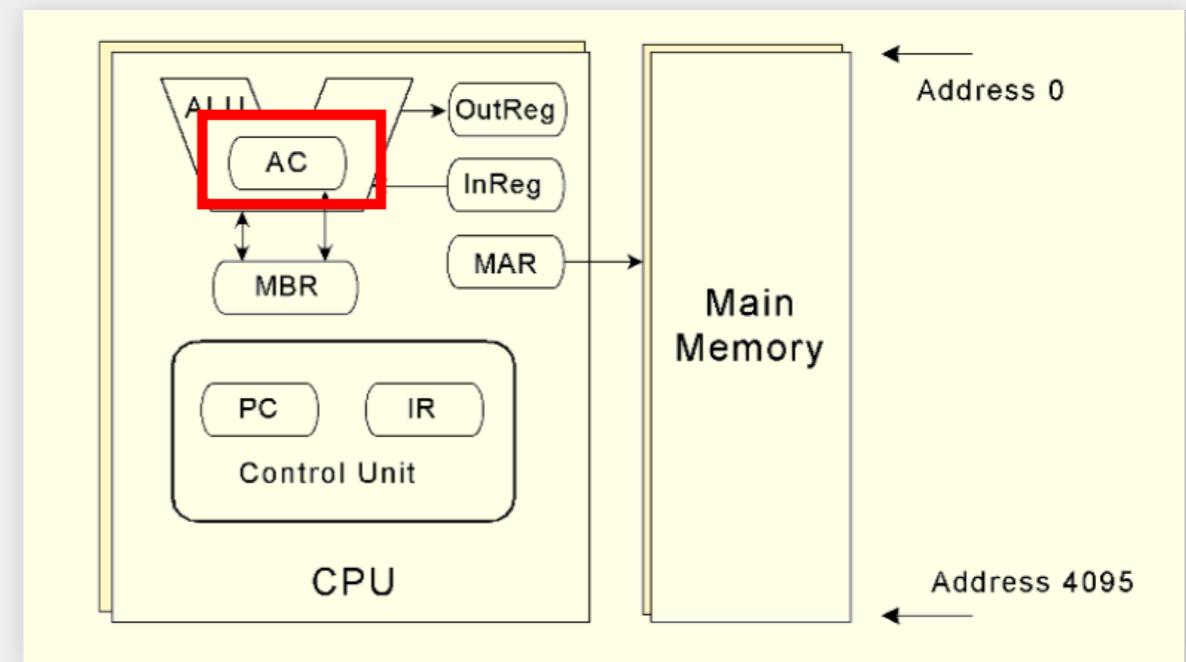
体系结构



寄存器

寄存器

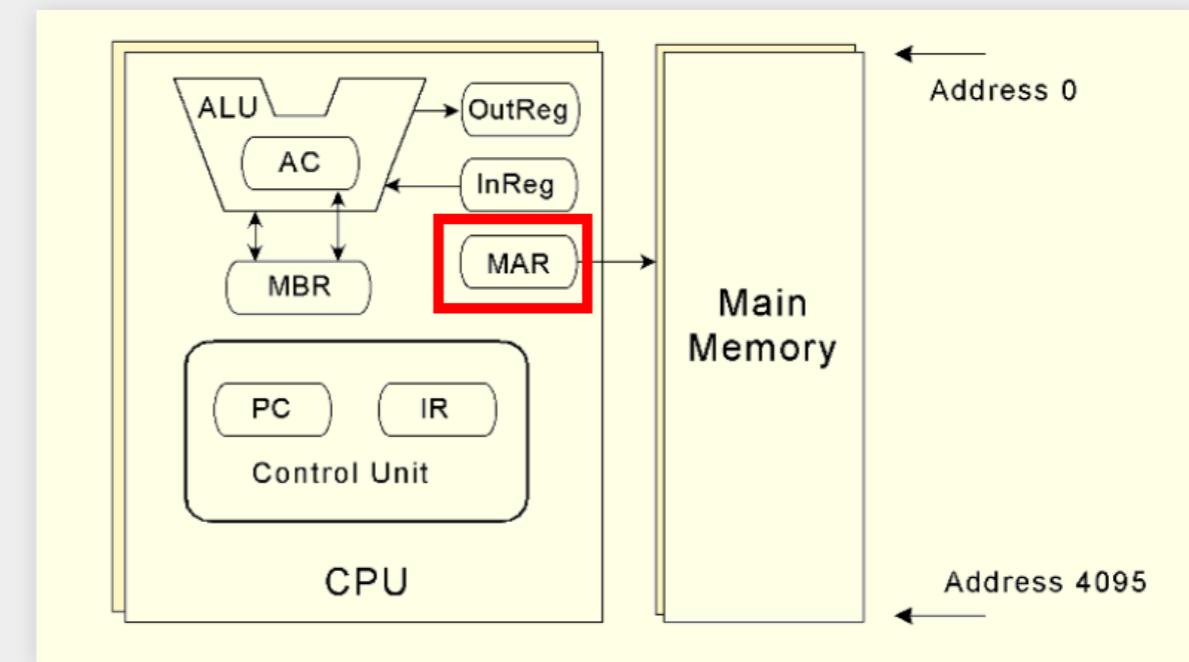
- 累加寄存器(Accumulator, AC): 16位寄存器, 储存指令运算数或运算结果



图片来源：Linda Null, Julia Lobur, *The Essentials of Computer Organization and Architecture*, 4t

寄存器

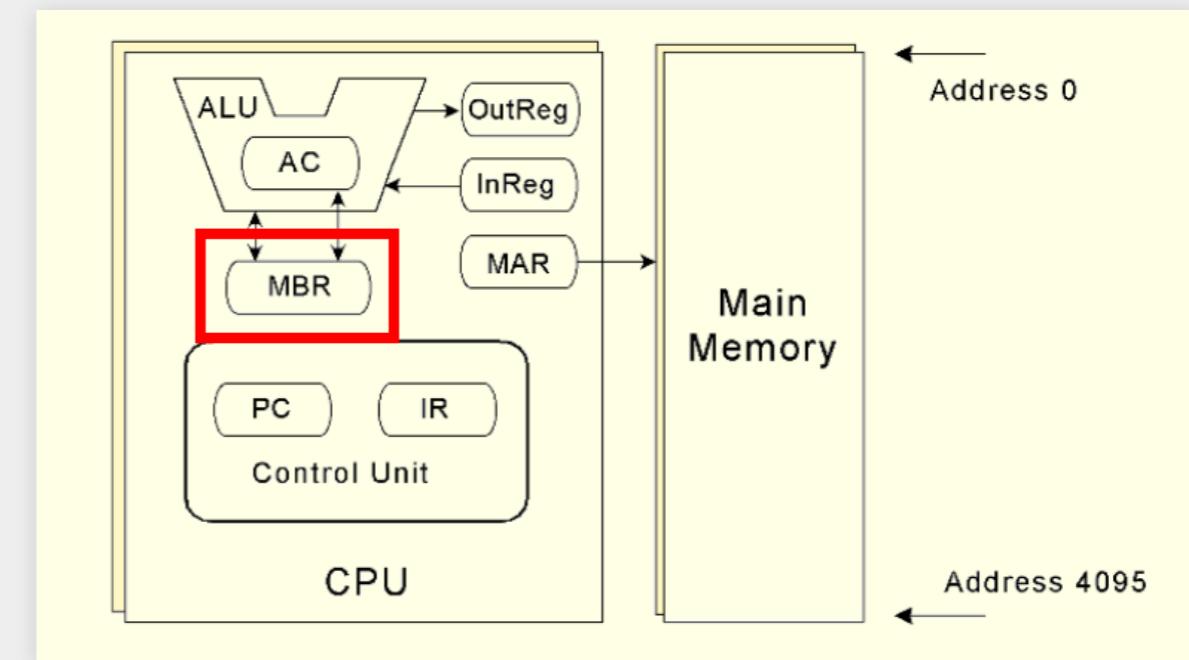
- 累加寄存器(Accumulator, AC): 16位寄存器, 储存指令运算数或运算结果
- 存储器地址寄存器(Memory Address Register, MAR): 12位寄存器, 用于保存引用数据的寄存器地址



图片来源：Linda Null, Julia Lobur, *The Essentials of Computer Organization and Architecture*, 4t

寄存器

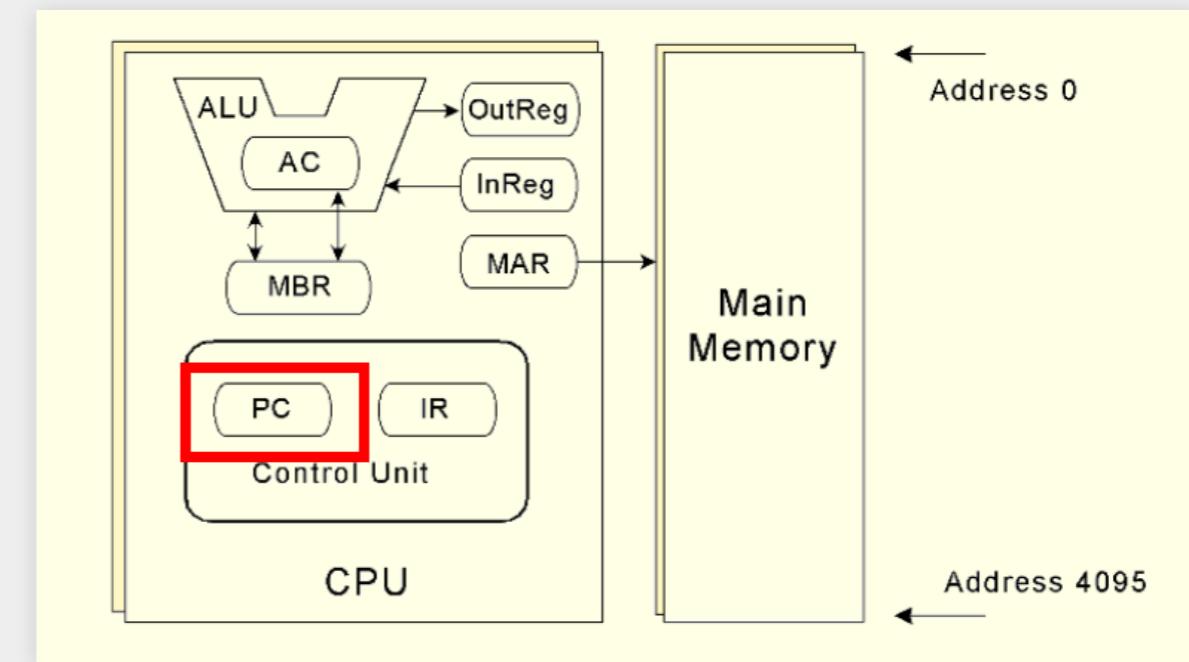
- 累加寄存器(Accumulator, AC): 16位寄存器, 储存指令运算数或运算结果
- 存储器地址寄存器(Memory Address Register, MAR): 12位寄存器, 用于保存引用数据的寄存器地址
- 存储器数据缓冲寄存器(Memory Buffer Register, MBR): 16位寄存器, 用于保存刚从内存中读取的数据或是即将放入内存的数据



图片来源：Linda Null, Julia Lobur, *The Essentials of Computer Organization and Architecture*, 4t

寄存器

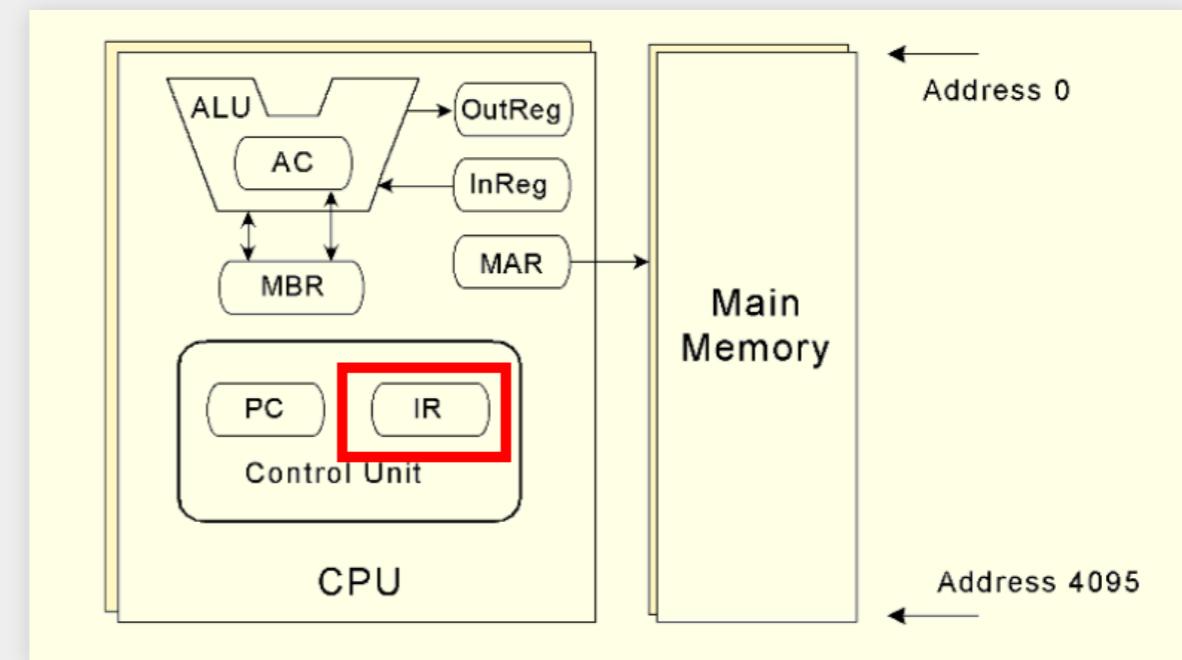
- 累加寄存器(Accumulator, AC): 16位寄存器, 储存指令运算数或运算结果
- 存储器地址寄存器(Memory Address Register, MAR): 12位寄存器, 用于保存引用数据的寄存器地址
- 存储器数据缓冲寄存器(Memory Buffer Register, MBR): 16位寄存器, 用于保存刚从内存中读取的数据或是即将放入内存的数据
- 程序计数器(Program Counter, PC): 12位寄存器, 用于保存将要执行的下一条指令的地址



图片来源：Linda Null, Julia Lobur, *The Essentials of Computer Organization and Architecture*, 4t

寄存器

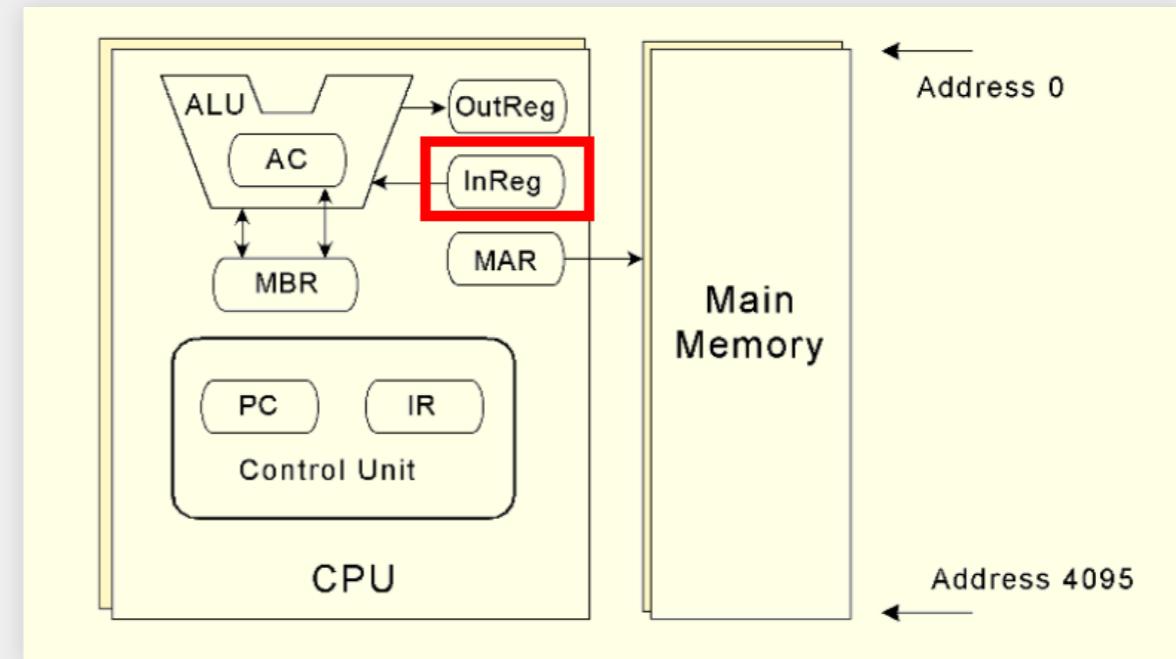
- 累加寄存器(Accumulator, AC): 16位寄存器, 储存指令运算数或运算结果
- 存储器地址寄存器(Memory Address Register, MAR): 12位寄存器, 用于保存引用数据的寄存器地址
- 存储器数据缓冲寄存器(Memory Buffer Register, MBR): 16位寄存器, 用于保存刚从内存中读取的数据或是即将放入内存的数据
- 程序计数器(Program Counter, PC): 12位寄存器, 用于保存将要执行的下一条指令的地址
- 指令寄存器(Instruction Register, IR): 16位寄存器, 用于保存将要执行的下一条指令



图片来源：Linda Null, Julia Lobur, *The Essentials of Computer Organization and Architecture*, 4t

寄存器

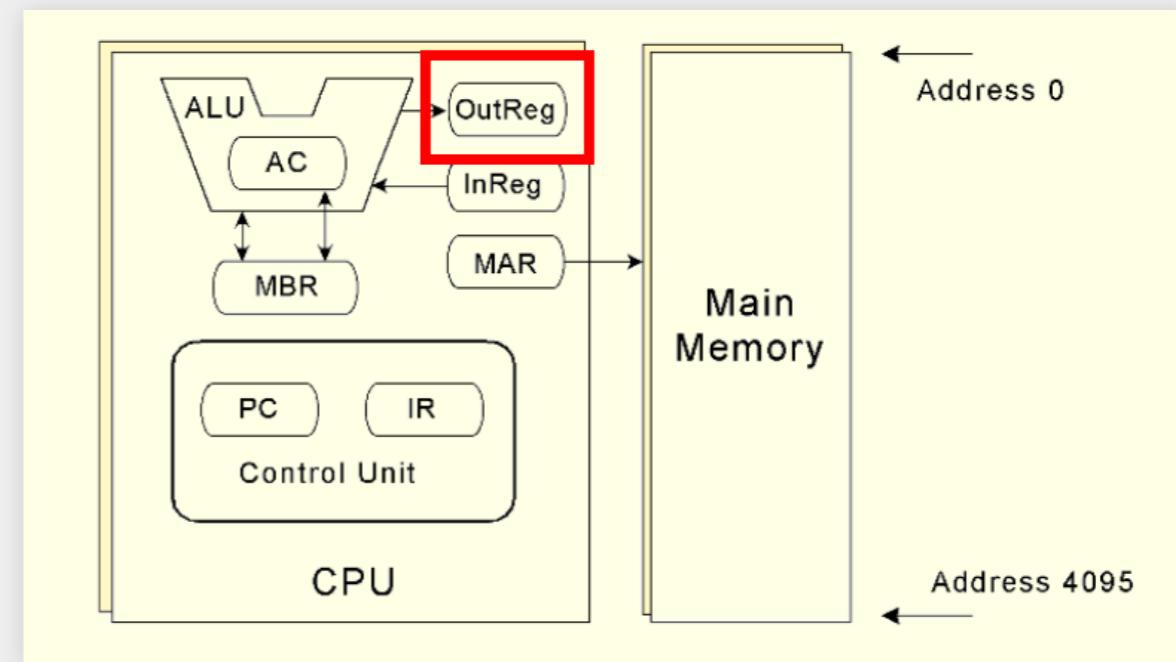
- 累加寄存器(Accumulator, AC): 16位寄存器, 储存指令运算数或运算结果
- 存储器地址寄存器(Memory Address Register, MAR): 12位寄存器, 用于保存引用数据的寄存器地址
- 存储器数据缓冲寄存器(Memory Buffer Register, MBR): 16位寄存器, 用于保存刚从内存中读取的数据或是即将放入内存的数据
- 程序计数器(Program Counter, PC): 12位寄存器, 用于保存将要执行的下一条指令的地址
- 指令寄存器(Instruction Register, IR): 16位寄存器, 用于保存将要执行的下一条指令
- 输入寄存器(Input Register, InReg): 8位寄存器, 代表输入设备读入的数据



图片来源：Linda Null, Julia Lobur, *The Essentials of Computer Organization and Architecture*, 4t

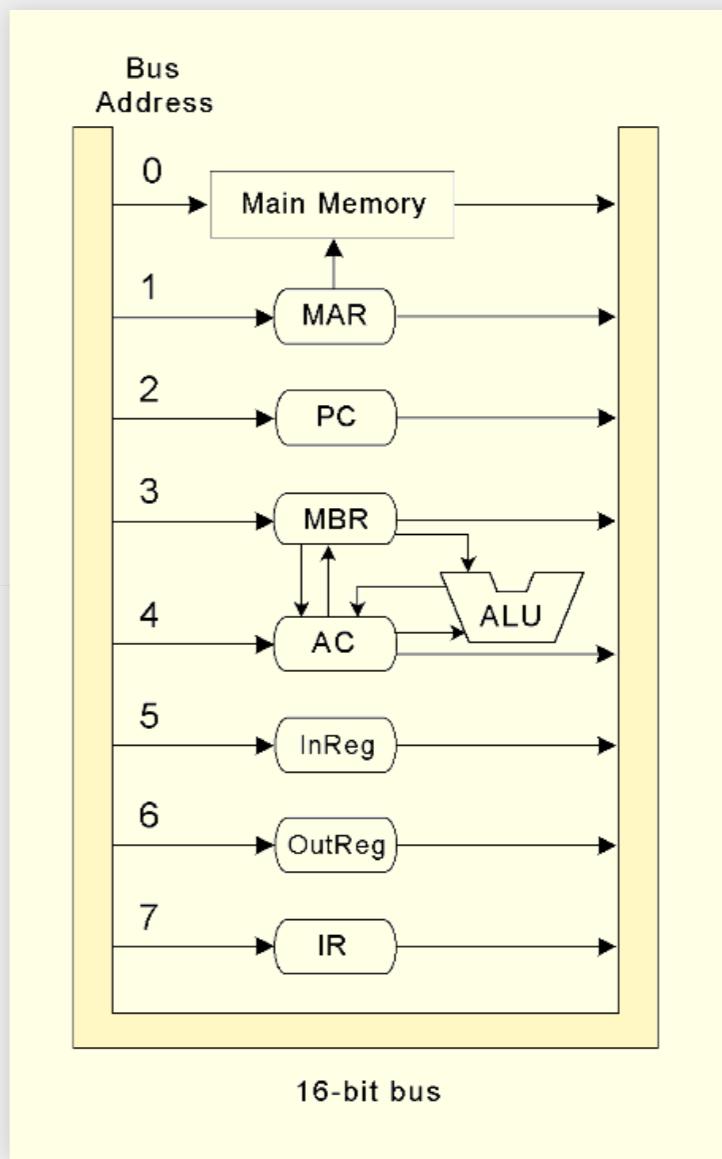
寄存器

- 累加寄存器(Accumulator, AC): 16位寄存器, 储存指令运算数或运算结果
- 存储器地址寄存器(Memory Address Register, MAR): 12位寄存器, 用于保存引用数据的寄存器地址
- 存储器数据缓冲寄存器(Memory Buffer Register, MBR): 16位寄存器, 用于保存刚从内存中读取的数据或是即将放入内存的数据
- 程序计数器(Program Counter, PC): 12位寄存器, 用于保存将要执行的下一条指令的地址
- 指令寄存器(Instruction Register, IR): 16位寄存器, 用于保存将要执行的下一条指令
- 输入寄存器(Input Register, InReg): 8位寄存器, 代表输入设备读入的数据
- 输出寄存器(Output Register, OutReg): 8位寄存器, 代表传输给输出设备的数据



图片来源：Linda Null, Julia Lobur, *The Essentials of Computer Organization and Architecture*, 4t

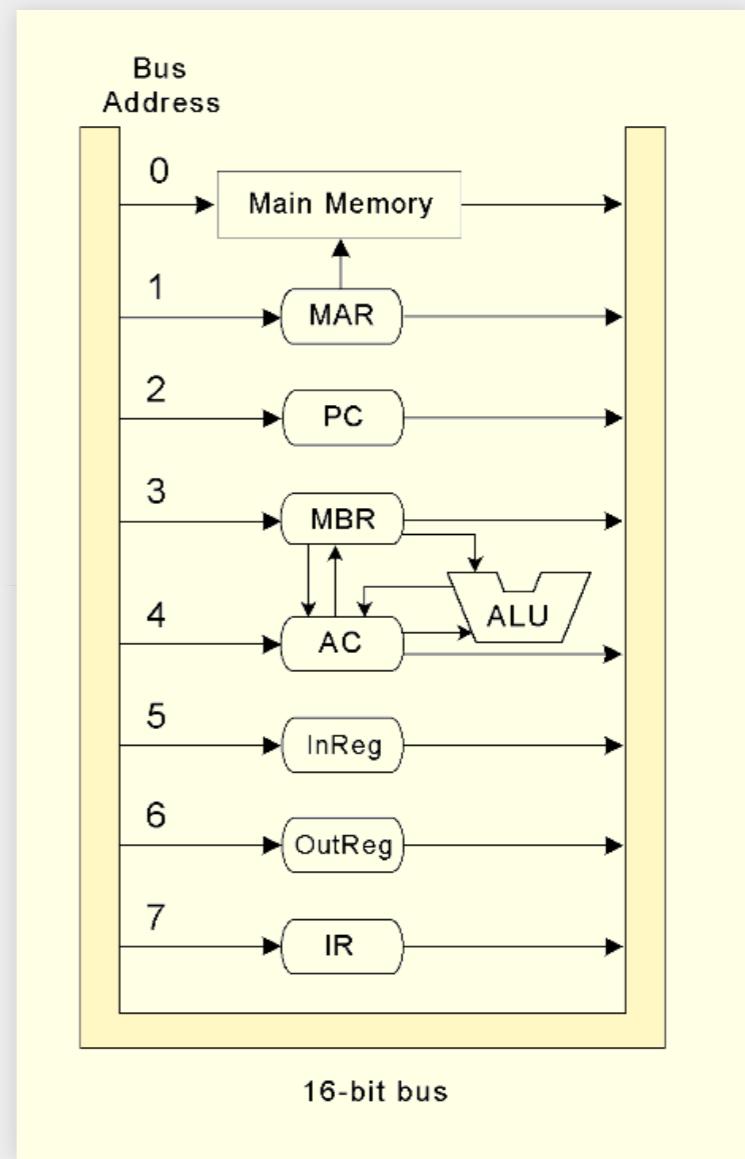
数据通路



图片来源：Linda Null, Julia Lobur, *The Essentials of Computer Organization and Architecture*, 5th Edition

数据通路

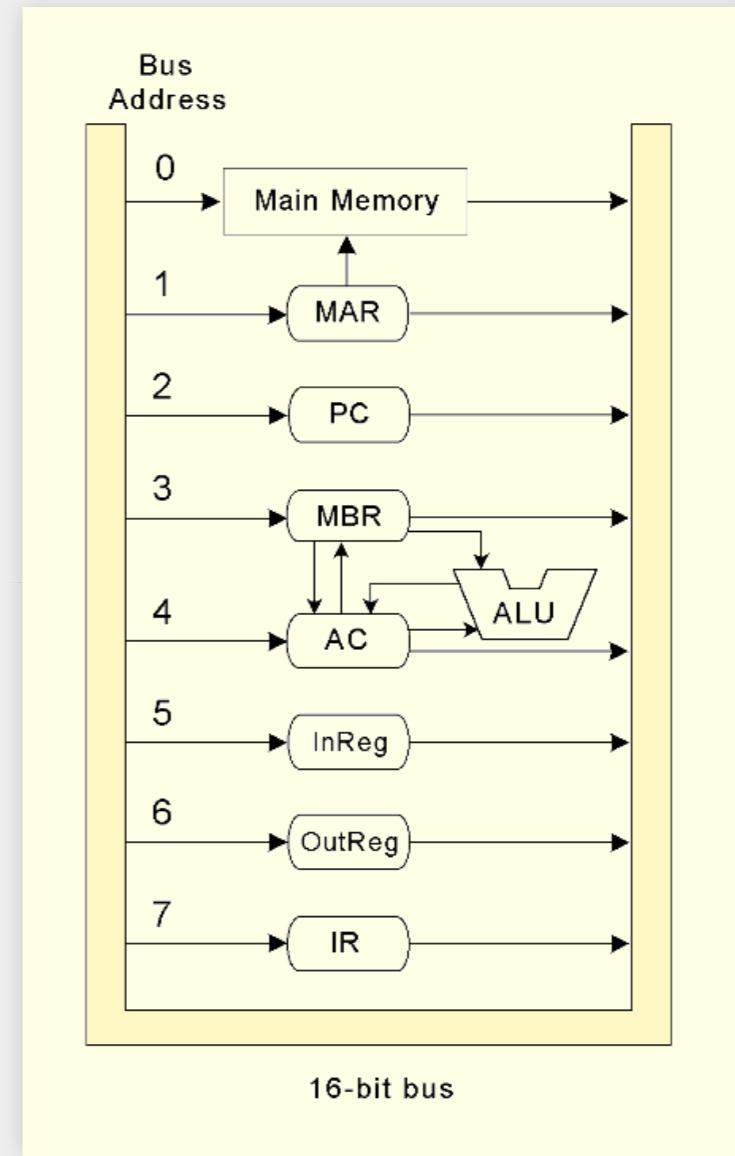
- 所有寄存器、内存共享一个16位总线



图片来源：Linda Null, Julia Lobur, *The Essentials of Computer Organization and Architecture*, 5th Edition

数据通路

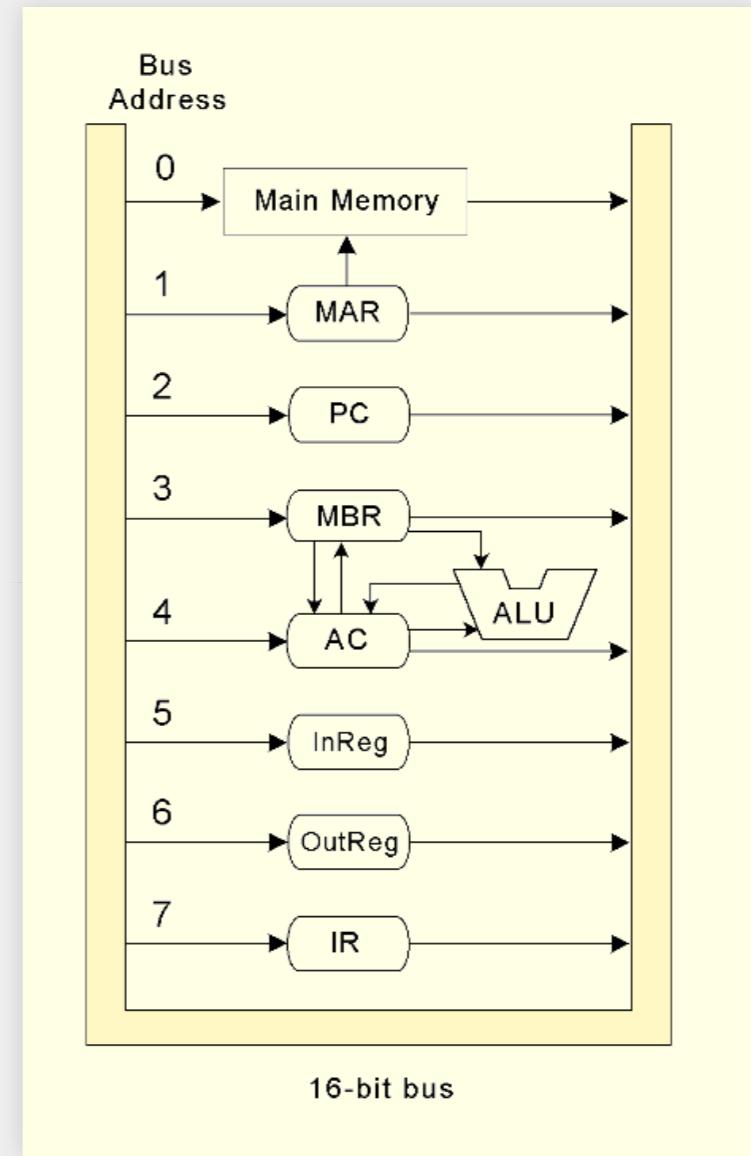
- 所有寄存器、内存共享一个16位总线
- 总线仲裁：每个总线上的设备对应了一个编号，只有当控制总线上的值对应了某个设备的编号时，该设备才可以使用总线



图片来源：Linda Null, Julia Lobur, *The Essentials of Computer Organization and Architecture*, 5th Edition

数据通路

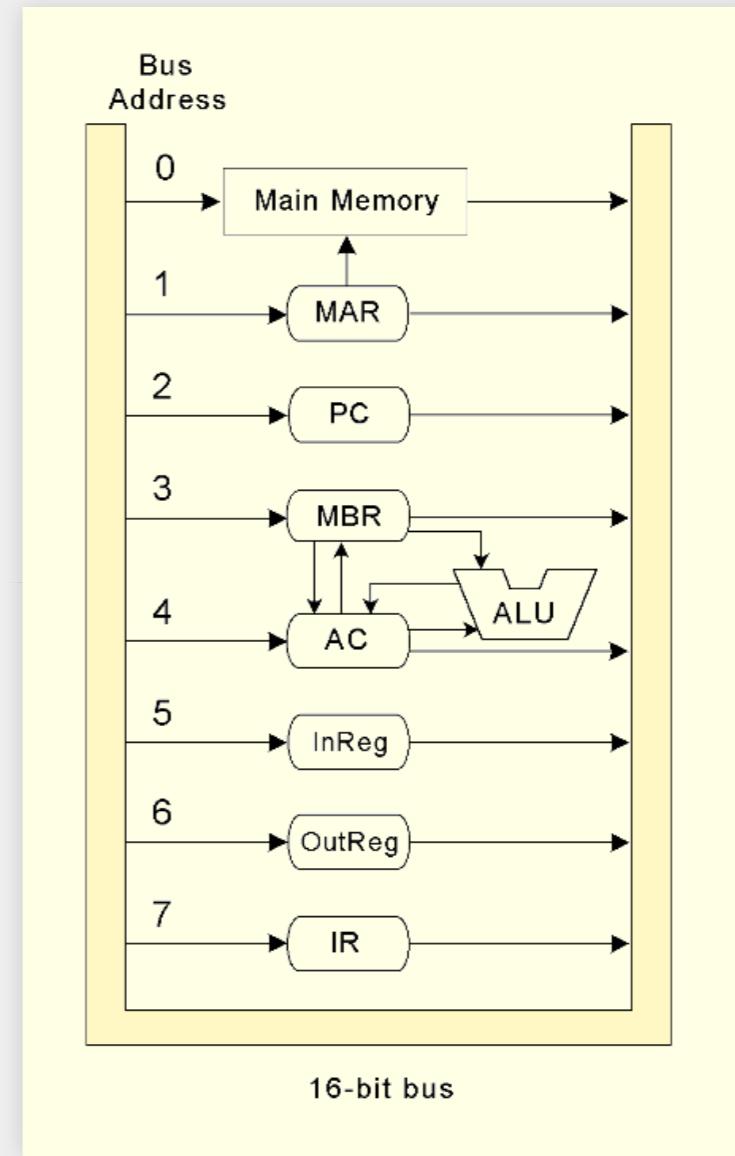
- 所有寄存器、内存共享一个16位总线
- 总线仲裁：每个总线上的设备对应了一个编号，只有当控制总线上的值对应了某个设备的编号时，该设备才可以使用总线
- MBR和AC、MBR和ALU、以及AC和ALU之间有特殊路径，可以与总线并行传输数据(注意：图中箭头代表的是信息流向，不是实际电路连接)



图片来源：Linda Null, Julia Lobur, *The Essentials of Computer Organization and Architecture*, 5th Edition

数据通路

- 所有寄存器、内存共享一个16位总线
- 总线仲裁：每个总线上的设备对应了一个编号，只有当控制总线上的值对应了某个设备的编号时，该设备才可以使用总线
- MBR和AC、MBR和ALU、以及AC和ALU之间有特殊路径，可以与总线并行传输数据(注意：图中箭头代表的是信息流向，不是实际电路连接)
- MARIE的总线采用了哪种仲裁方法？**



指令集

指令集

指令集架构(Instruction Set Architecture, ISA)

指令集

指令集架构(Instruction Set Architecture, ISA)

- 计算机可以执行的指令组成的集合以及指令编码格式

指令集

指令集架构(Instruction Set Architecture, ISA)

- 计算机可以执行的指令组成的集合以及指令编码格式
- 计算机软硬件的接口：用该指令集实现的程序可以运行在任何实现了这个指令集的硬件上

指令集

指令集架构(Instruction Set Architecture, ISA)

- 计算机可以执行的指令组成的集合以及指令编码格式
- 计算机软硬件的接口：用该指令集实现的程序可以运行在任何实现了这个指令集的硬件上

MARIE的指令集

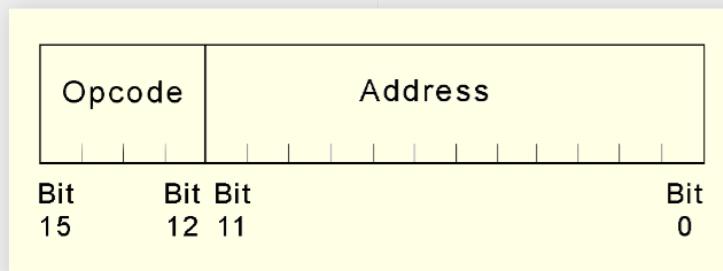
指令集

指令集架构(Instruction Set Architecture, ISA)

- 计算机可以执行的指令组成的集合以及指令编码格式
- 计算机软硬件的接口：用该指令集实现的程序可以运行在任何实现了这个指令集的硬件上

MARIE的指令集

- 指令集：其中0x1-0x9为基本指令，0x0、0xa-0xe为扩展指令。扩展指令实现了**间接寻址**(指针)



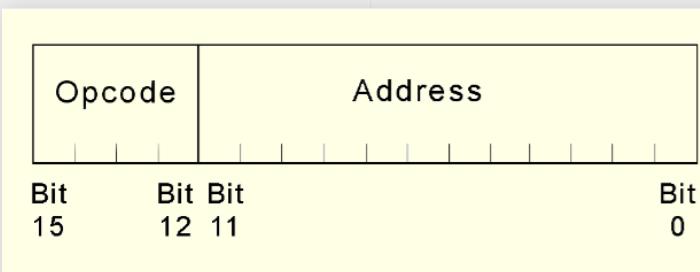
指令集

指令集架构(Instruction Set Architecture, ISA)

- 计算机可以执行的指令组成的集合以及指令编码格式
- 计算机软硬件的接口：用该指令集实现的程序可以运行在任何实现了这个指令集的硬件上

MARIE的指令集

- 指令集：其中0x1-0x9为基本指令，0x0、0xa-0xe为扩展指令。扩展指令实现了**间接寻址**(指针)
- 指令的编码：16位指令，4位操作码(Opcode)，12位地址



指令编码	指令	含义
0x0	JNS x	将PC写入地址为x的存储单元，并将x+1写入PC
0x1	LOAD x	将地址为x的存储单元中的值存入AC
0x2	STORE x	将AC中的值存储到地址为x的存储单元中
0x3	ADD x	将地址x中的值与AC中的值相加的结果存入AC
0x4	SUBT x	将AC中的值减去地址x中的值的结果存入AC
0x5	INPUT	从键盘输入一个数值到AC
0x6	OUTPUT	将AC中的数值输出到显示器
0x7	HALT	终止程序的运行
0x8	SKIPCOND	有条件地跳过下一条指令
0x9	JUMP x	将x的值装入PC
0xa	CLEAR	将AC中的值置为0
0xb	ADDI x	读取地址x中的地址，将其对应的值与AC中的值相加的结果存入AC
0xc	JUMPI x	读取地址x中的地址，将其对应的值写入PC
0xd	LOADI x	读取地址x中的地址，将其对应的值写入AC
0xe	STOREI x	读取地址x中的地址，将AC写入其对应的存储单元

寄存器传输表示/寄存器传输语言

寄存器传输表示/寄存器传输语言

- MERIE的每个指令涉及到多个寄存器和内存之间的信息传递

寄存器传输表示/寄存器传输语言

- MERIE的每个指令涉及到多个寄存器和内存之间的信息传递
- 每个细分的信息传递称为一个**微操作**
(Microoperation)

寄存器传输表示/寄存器传输语言

- MERIE的每个指令涉及到多个寄存器和内存之间的信息传递
- 每个细分的信息传递称为一个**微操作**(Microoperation)
- 我们采用寄存器传输表示(Register Transfer Notation, RTN)或者寄存器传输语言(Register Transfer Language, RTL)来形式化地描述这些微操作

寄存器传输表示/寄存器传输语言

- MERIE的每个指令涉及到多个寄存器和内存之间的信息传递
- 每个细分的信息传递称为一个**微操作**(Microoperation)
- 我们采用寄存器传输表示(Register Transfer Notation, RTN)或者寄存器传输语言(Register Transfer Language, RTL)来形式化地描述这些微操作

$A \leftarrow B$

- 代表：将 B 中的值写入 A
- A ：寄存器(用寄存器名字表示)或者是内存存储单元(用 $M[y]$ 代表地址为 y 中的值所对应的存储单元)
- B ：指令操作数 (用 x 表示)、寄存器(用寄存器名字表示)或者是内存存储单元(用 $M[y]$ 代表地址为 y 中的值所对应的存储单元)

寄存器传输表示/寄存器传输语言

- MERIE的每个指令涉及到多个寄存器和内存之间的信息传递
- 每个细分的信息传递称为一个**微操作**(Microoperation)
- 我们采用寄存器传输表示(Register Transfer Notation, RTN)或者寄存器传输语言(Register Transfer Language, RTL)来形式化地描述这些微操作

$$A \leftarrow B$$

- 代表：将B中的值写入A
- A：寄存器(用寄存器名字表示)或者是内存存储单元(用M[y]代表地址为y中的值所对应的存储单元)
- B：指令操作数 (用x表示)、寄存器(用寄存器名字表示)或者是内存存储单元(用M[y]代表地址为y中的值所对应的存储单元)

用RTN描述MARIE指令集

指令	RTN	指令	RTN
JNS x	MBR←PC MAR←x M[MAR]←MBR MBR←x AC←1 AC←AC+MBR PC←AC	SKIPCOND	IF x[11-10] = 00 AND AC < 0 THEN PC←PC + 1 ELSE IF x[11-10] = 01 AND AC = 0 THEN PC←PC + 1 ELSE IF x[11-10] = 10 AND AC > 0 THEN PC←PC + 1
LOAD x	MAR←x MBR←M[MAR] AC←MBR	JUMP x	PC←x
STORE x	MAR←x MBR←AC M[MAR]←MBR	CLEAR	AC←0
ADD x	MAR←x MBR←M[MAR] AC←AC + MBR	ADDI x	MAR←x MBR←M[MAR] MAR←MBR MBR←M[MAR] AC←AC + MBR
SUBT x	MAR←x MBR←M[MAR] AC←AC - MBR	JUMPI x	MAR←x MBR←M[MAR] PC←MBR
INPUT	AC←InReg	LOADI x	MAR←x MBR←M[MAR] MAR←MBR MBR←M[MAR] AC←MBR
OUTPUT	OutReg←AC	STOREI x	MAR←x MBR←M[MAR] MAR←MBR MBR←AC M[MAR]←MBR
HALT	-		

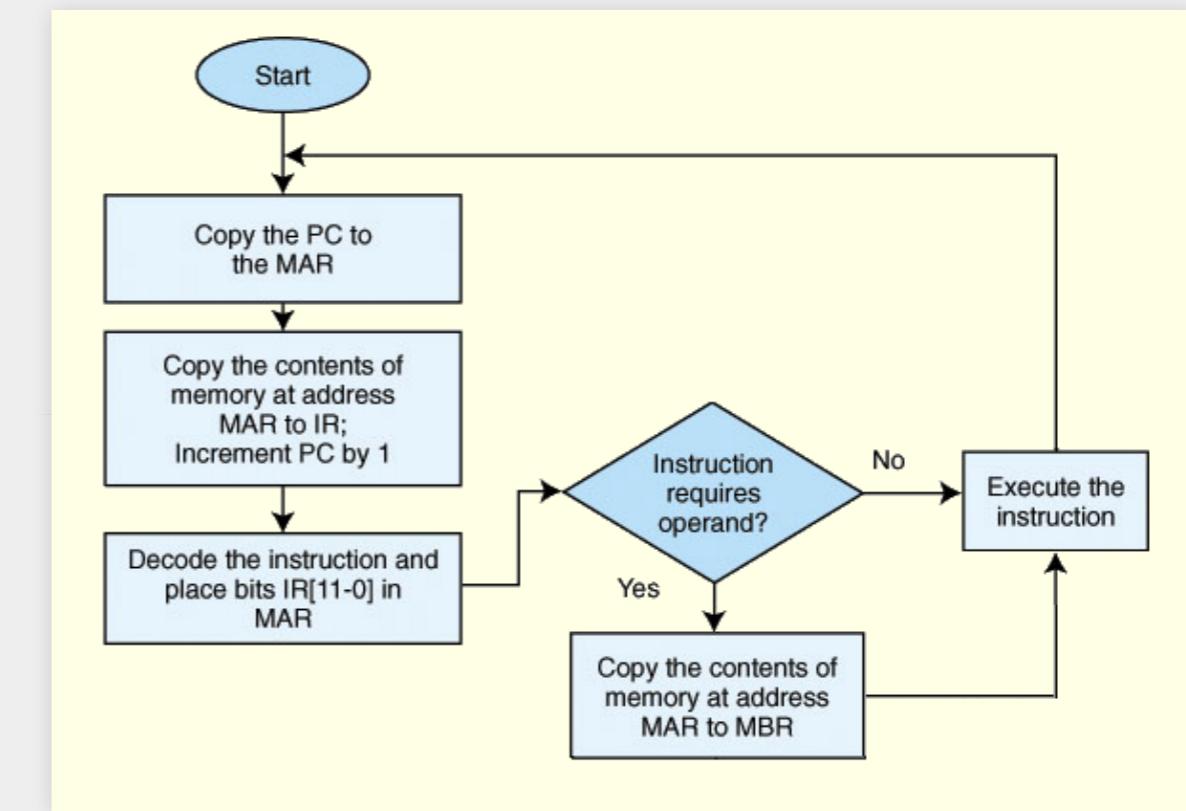
指令执行流程

指令执行流程

- 冯-诺依曼模型：取译码执行

指令执行流程

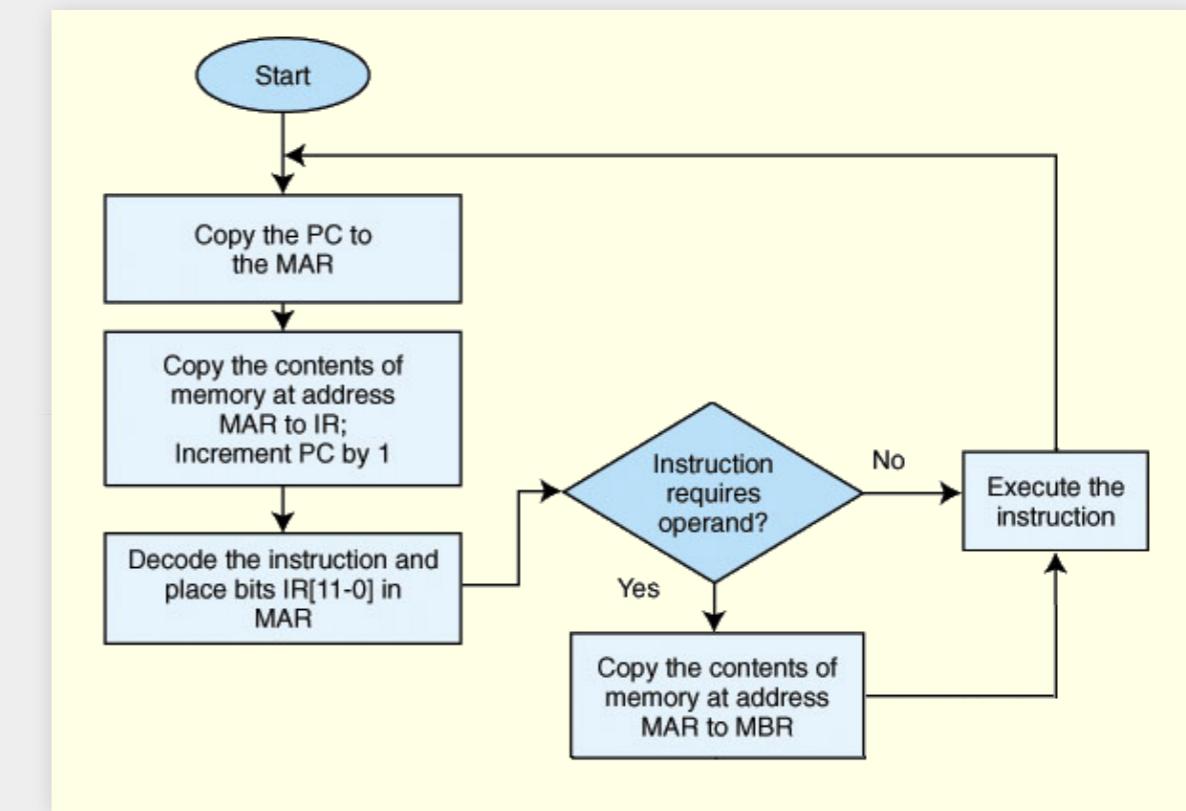
- 冯-诺依曼模型：取译码执行
- MARIE中的取译码执行过程



指令执行流程

- 冯-诺依曼模型：取译码执行
- MARIE中的取译码执行过程

用RTN描述MARIE的指令执行过程

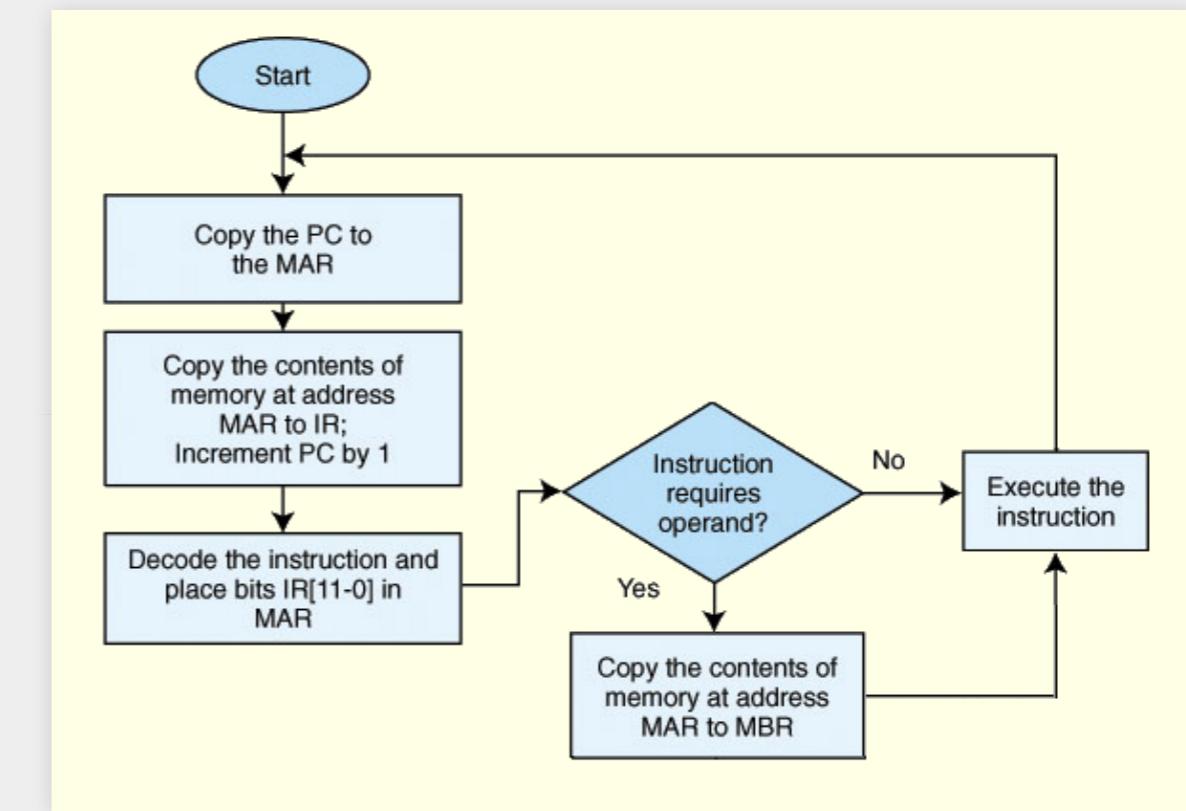


指令执行流程

- 冯-诺依曼模型：取译码执行
- MARIE中的取译码执行过程

用RTN描述MARIE的指令执行过程

1. MAR \leftarrow PC

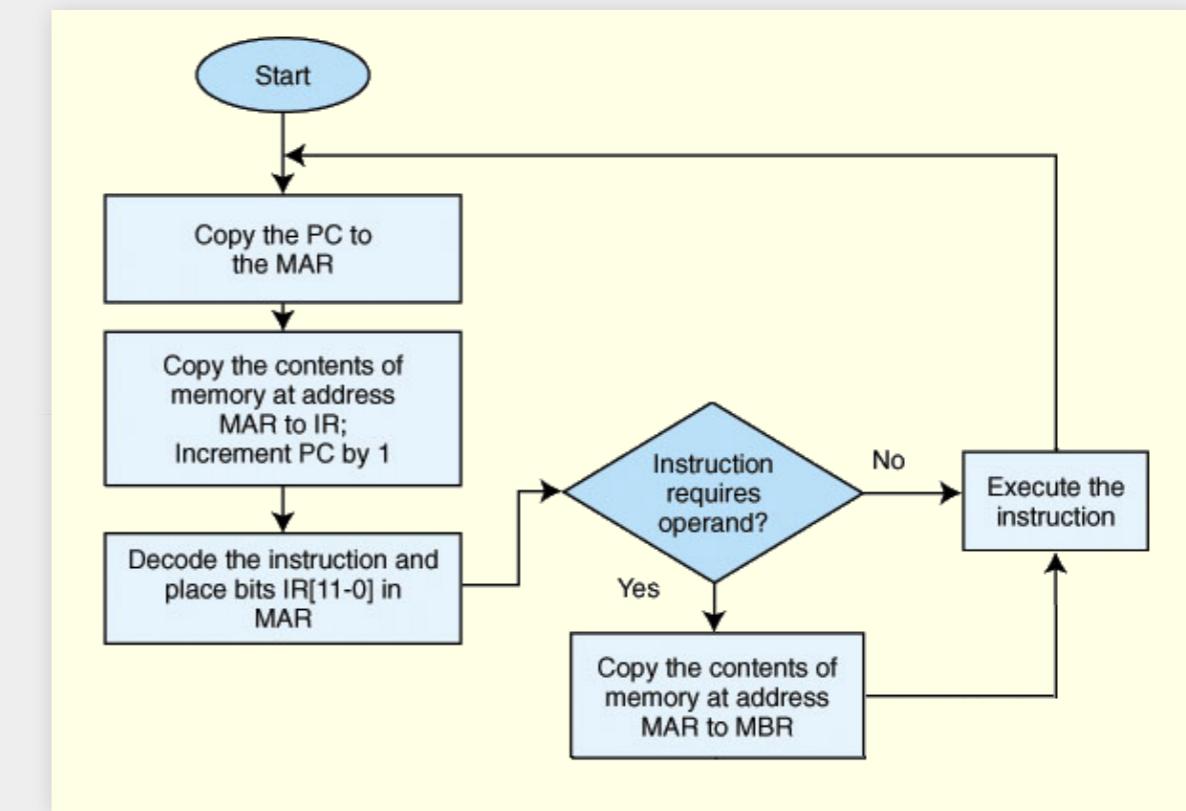


指令执行流程

- 冯-诺依曼模型：取译码执行
- MARIE中的取译码执行过程

用RTN描述MARIE的指令执行过程

1. $\text{MAR} \leftarrow \text{PC}$
2. $\text{IR} \leftarrow M[\text{MAR}], \text{PC} \leftarrow \text{PC} + 1$

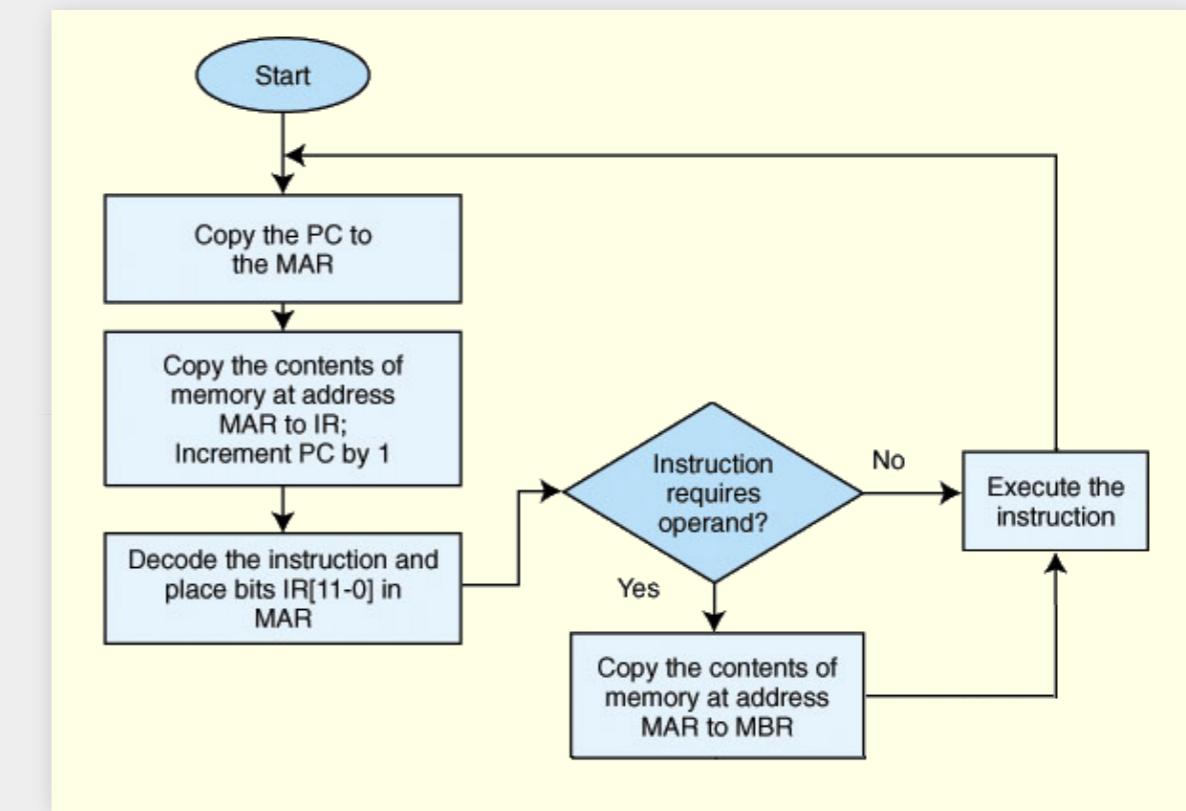


指令执行流程

- 冯-诺依曼模型：取译码执行
- MARIE中的取译码执行过程

用RTN描述MARIE的指令执行过程

1. $\text{MAR} \leftarrow \text{PC}$
2. $\text{IR} \leftarrow M[\text{MAR}], \text{PC} \leftarrow \text{PC} + 1$
3. $x \triangleq \text{IR}[11-0], \text{OP} \leftarrow \text{DECODE}(\text{IR}[15-12])$

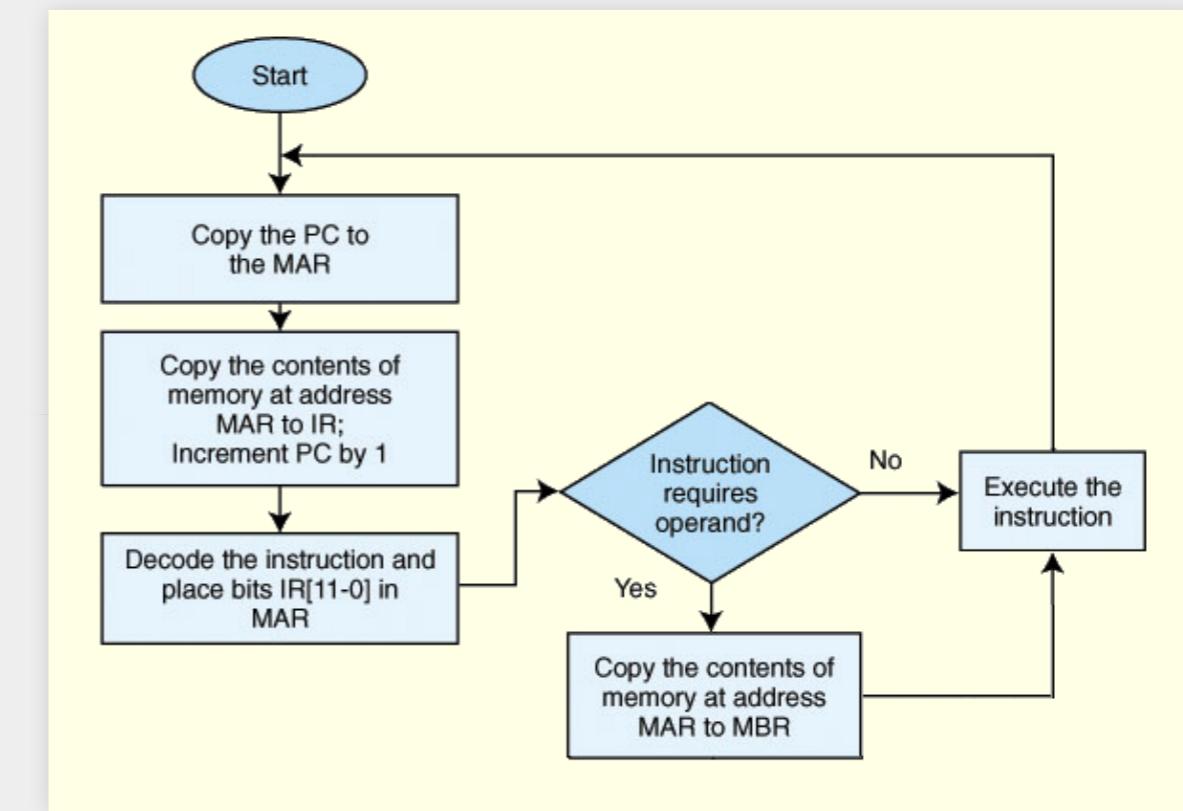


指令执行流程

- 冯-诺依曼模型：取译码执行
- MARIE中的取译码执行过程

用RTN描述MARIE的指令执行过程

1. $\text{MAR} \leftarrow \text{PC}$
2. $\text{IR} \leftarrow M[\text{MAR}], \text{PC} \leftarrow \text{PC} + 1$
3. $x \triangleq \text{IR}[11-0], \text{OP} \leftarrow \text{DECODE}(\text{IR}[15-12])$
4. $\text{EXEC}(\text{OP})$



指令执行流程

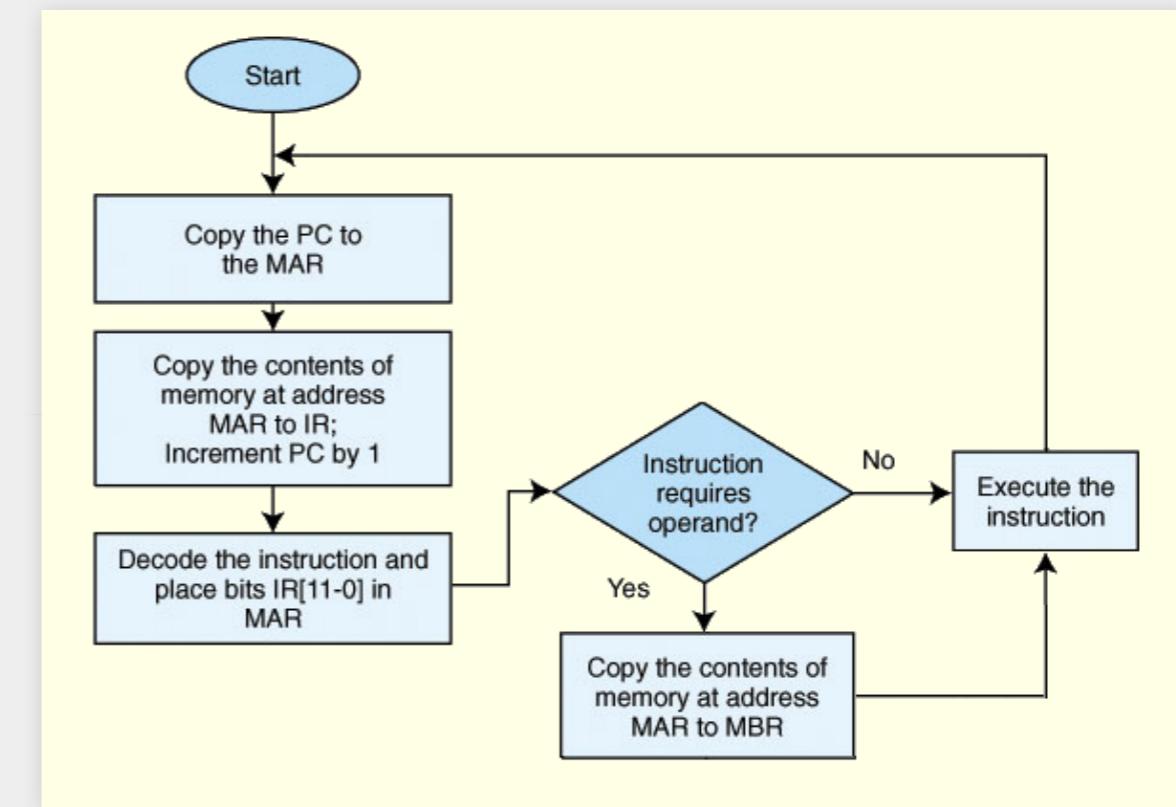
- 冯-诺依曼模型：取译码执行
- MARIE中的取译码执行过程

用RTN描述MARIE的指令执行过程

1. $\text{MAR} \leftarrow \text{PC}$
2. $\text{IR} \leftarrow M[\text{MAR}], \text{PC} \leftarrow \text{PC} + 1$
3. $x \triangleq \text{IR}[11-0], \text{OP} \leftarrow \text{DECODE}(\text{IR}[15-12])$
4. $\text{EXEC}(\text{OP})$

优化指令执行过程：

- 许多指令的第一步操作是 $\text{MAR} \leftarrow x$
- 译码同时将 x 预先载入 MAR 中，即 $\text{MAR} \leftarrow \text{IR}[11-0]$



中断处理和I/O

- MARIE不支持中断处理
- MARIE通过InReg和OutReg进行I/O处理
 - INPUT指令是**阻塞模式**: CPU等待, 直到InReg接收到用户输入
 - OUTPUT指令也是**阻塞模式**: 输出设备等待, 直到OutReg接收到CPU输出

样例程序

Address	Instruction	Binary Contents of Memory Address	Hex Contents of Memory
100	Load 104	0001000100000100	1104
101	Add 105	0011000100000101	3105
102	Store 106	0100000100000110	4106
103	Halt	0111000000000000	7000
104	0023	0000000000100011	0023
105	FFE9	111111111101001	FFE9
106	0000	0000000000000000	0000

样例程序

Address	Instruction	Binary Contents of Memory Address	Hex Contents of Memory
100	Load 104	0001000100000100	1104
101	Add 105	0011000100000101	3105
102	Store 106	0100000100000110	4106
103	Halt	0111000000000000	7000
104	0023	0000000000100011	0023
105	FFE9	111111111101001	FFE9
106	0000	0000000000000000	0000

- 内存位置100-103对应了四条指令，实现了一个加法操作

样例程序

Address	Instruction	Binary Contents of Memory Address	Hex Contents of Memory
100	Load 104	0001000100000100	1104
101	Add 105	0011000100000101	3105
102	Store 106	0100000100000110	4106
103	Halt	0111000000000000	7000
104	0023	0000000000100011	0023
105	FFE9	111111111101001	FFE9
106	0000	0000000000000000	0000

- 内存位置100-103对应了四条指令，实现了一个加法操作
- 内存位置104-106对应了三个16位数据，其中两个储存了输入操作数，一个放置输出结果

样例程序

Address	Instruction	Binary Contents of Memory Address	Hex Contents of Memory
100	Load 104	0001000100000100	1104
101	Add 105	0011000100000101	3105
102	Store 106	0100000100000110	4106
103	Halt	0111000000000000	7000
104	0023	0000000000100011	0023
105	FFE9	111111111101001	FFE9
106	0000	0000000000000000	0000

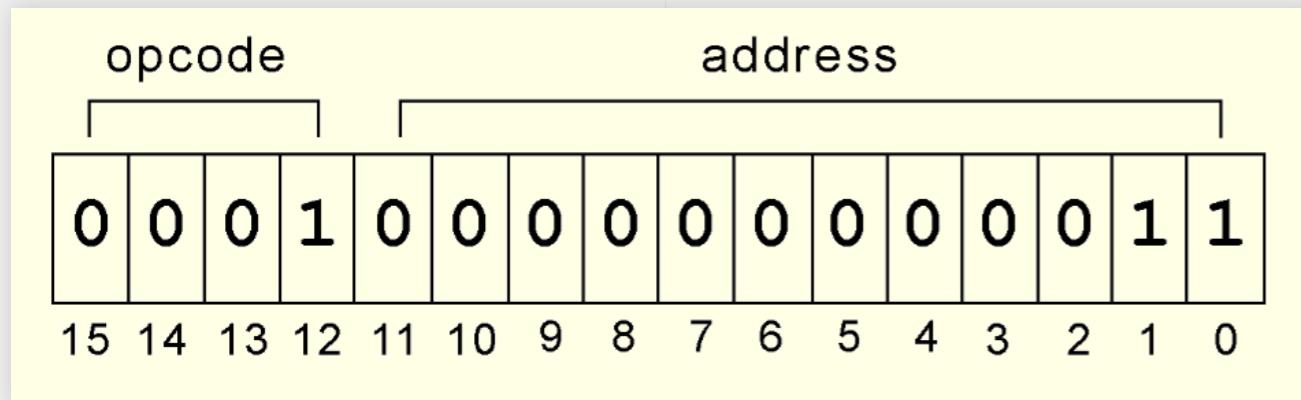
描述程序的运行状态：**程序踪迹**(Program Trace)

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		100	-	-	-	-
Fetch	MAR ← PC	100	-	100	-	-
	IR ← M[MAR]	100	1104	100	-	-
	PC ← PC + 1	101	1104	100	-	-
Decode	MAR ← IR[11-0]	101	1104	104	-	-
	(Decode IR[15-12])	101	1104	104	-	-
Get operand	MBR ← M[MAR]	101	1104	104	0023	-
Execute	AC ← MBR	101	1104	104	0023	0023

- 内存位置100-103对应了四条指令，实现了一个加法操作
- 内存位置104-106对应了三个16位数据，其中两个储存了输入操作数，一个放置输出结果

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		101	1104	104	0023	0023
Fetch	MAR ← PC	101	1104	101	0023	0023
	IR ← M[MAR]	101	3105	101	0023	0023
	PC ← PC + 1	102	3105	101	0023	0023
Decode	MAR ← IR[11-0]	102	3105	105	0023	0023
	(Decode IR[15-12])	102	3105	105	0023	0023
Get operand	MBR ← M[MAR]	102	3105	105	FFE9	0023
Execute	AC ← AC + MBR	102	3105	105	FFE9	000C

汇编语言



LOAD 3对应的二进制指令编码

- 采用二进制表示的指令称为**机器指令**：(0001 0000 0000 0011)
 - 采用符号表示的指令称为**汇编语言指令**(Assembly)：LOAD 3
 - 对应表示操作码的符号称为**助记符**(Mnemonic Instruction)：LOAD

- **汇编程序**根据汇编语言生成对应的机器指令
 - 简化指令操作码：采用**助记符**。例：LOAD, ADD, STORE
 - 简化地址和数据定位：采用**标记符号**。例：X, Y, Z
 - 在MARIE中，标记符号后必须加“,”
 - 简化数据表示：采用特定的**汇编指令**。例：DEC, HEX
 - 仅在编译时使用，不会编译成机器指令

Address	Instruction		
100	Load	X	
101	Add	Y	
102	Store	Z	
103	Halt		
104 X,	DEC	35	
105 Y,	DEC	-23	
106 Z,	HEX	0000	

汇编语言的编译过程

第一次通读(Pass): 建立符号表，翻译操作码

1	X
3	Y
2	Z
7	0 0 0

X	104
Y	105
Z	106

第二次通读: 将符号替换为对应的值

1	1	0	4
3	1	0	5
2	1	0	6
7	0	0	0
0	0	2	3
F	F	E	9
0	0	0	0

指令译码和执行：控制单元的实现

指令译码和执行：控制单元的实现

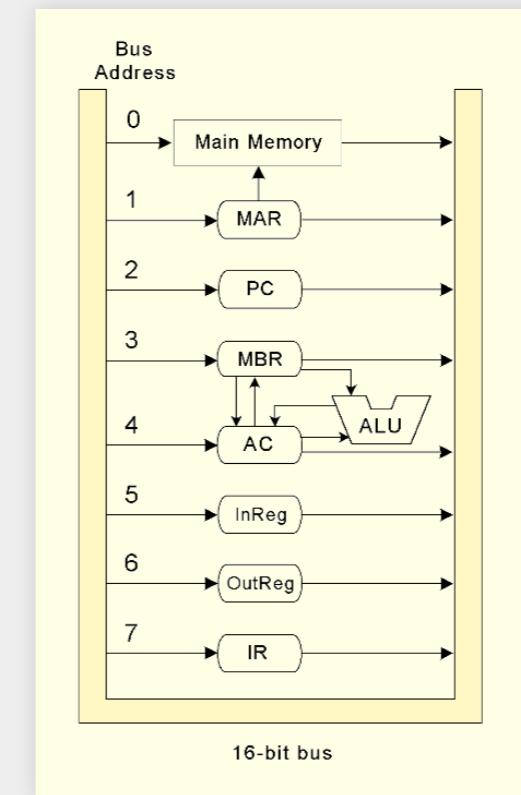
- 为了保证CPU的正确运行，需要通过**控制信号**来协调各部件的功能

指令译码和执行：控制单元的实现

- 为了保证CPU的正确运行，需要通过**控制信号**来协调各部件的功能
- MARIE架构中需要哪些控制信号？

指令译码和执行：控制单元的实现

- 为了保证CPU的正确运行，需要通过**控制信号**来协调各部件的功能
- MARIE架构中需要哪些控制信号？
 - 总线
- 控制总线的读和写：每个时钟周期**总线**可以读取一个设备的值，并写入一个设备
- 各需要3个控制信号：读取信号(P_0, P_1, P_2)和写入信号(P_3, P_4, P_5)
 - 例： $(P_2 P_1 P_0)_2 = 7_{10}$ 时，总线在读IR寄存器的值(0-11位)， $(P_5 P_4 P_3)_2 = 1_{10}$ 时，总线在写入MAR寄存器
 - 上面的例子实现了 $\text{MAR} \leftarrow \text{IR}[11-0]$



指令译码和执行：控制单元的实现

- 为了保证CPU的正确运行，需要通过**控制信号**来协调各部件的功能
- MARIE架构中需要哪些控制信号？
 - 总线: $P_0, P_1, P_2, P_3, P_4, P_5$
 - ALU
- 控制ALU中的多路复用器

ALU Control Signals		ALU Response
A_1	A_0	
0	0	Do Nothing
0	1	$AC \leftarrow AC + MBR$
1	0	$AC \leftarrow AC - MBR$
1	1	$AC \leftarrow 0$ (Clear)

指令译码和执行：控制单元的实现

- 为了保证CPU的正确运行，需要通过**控制信号**来协调各部件的功能
- MARIE架构中需要哪些控制信号？
 - 总线: $P_0, P_1, P_2, P_3, P_4, P_5$
 - ALU: A_0, A_1
 - 时钟
- 每个指令的完整执行需要多个时钟周期，需要根据当前的指令和对应的时钟周期计数决定对应的微操作信号
 - MARIE的时钟周期最多8个: $T_0 - T_7$
 - 采用3-8译码器实现，每个时钟周期仅有一个值为1
 - 以ADD x为例，整个取译码执行总共需要6个时钟周期
 - C_r 信号用于重置时钟周期计数

指令译码和执行：控制单元的实现

- 为了保证CPU的正确运行，需要通过**控制信号**来协调各部件的功能
- MARIE架构中需要哪些控制信号？
 - 总线: $P_0, P_1, P_2, P_3, P_4, P_5$
 - ALU: A_0, A_1
 - 时钟: T_0, \dots, T_7, C_r
 - PC
- PC有两个输入源：总线（JNS、JUMP、JUMPI）和计数器（其它）
- IncrPC：值为1的时候PC读取计数器的值，否则从总线读入

指令译码和执行：控制单元的实现

- 为了保证CPU的正确运行，需要通过**控制信号**来协调各部件的功能
- MARIE架构中需要哪些控制信号？
 - 总线: $P_0, P_1, P_2, P_3, P_4, P_5$
 - ALU: A_0, A_1
 - 时钟: T_0, \dots, T_7, C_r
 - PC: IncrPC
 - 内存
- 控制内存的读写模式
 - M_W : 值为1的时候写入内存，否则不写
 - M_R : 值为1的时候从内存读入总线，否则不读

指令译码和执行：控制单元的实现

- 为了保证CPU的正确运行，需要通过**控制信号**来协调各部件的功能
- MARIE架构中需要哪些控制信号？
 - 总线: $P_0, P_1, P_2, P_3, P_4, P_5$
 - ALU: A_0, A_1
 - 时钟: T_0, \dots, T_7, C_r
 - PC: IncrPC
 - 内存: M_W, M_R
 - 替代数据源
- 控制AC和MBR等具有替代输入源的寄存器
- L_{alt} : 值为1的时候从替代源读入数据，否则从总线读入

指令译码和执行：控制单元的实现

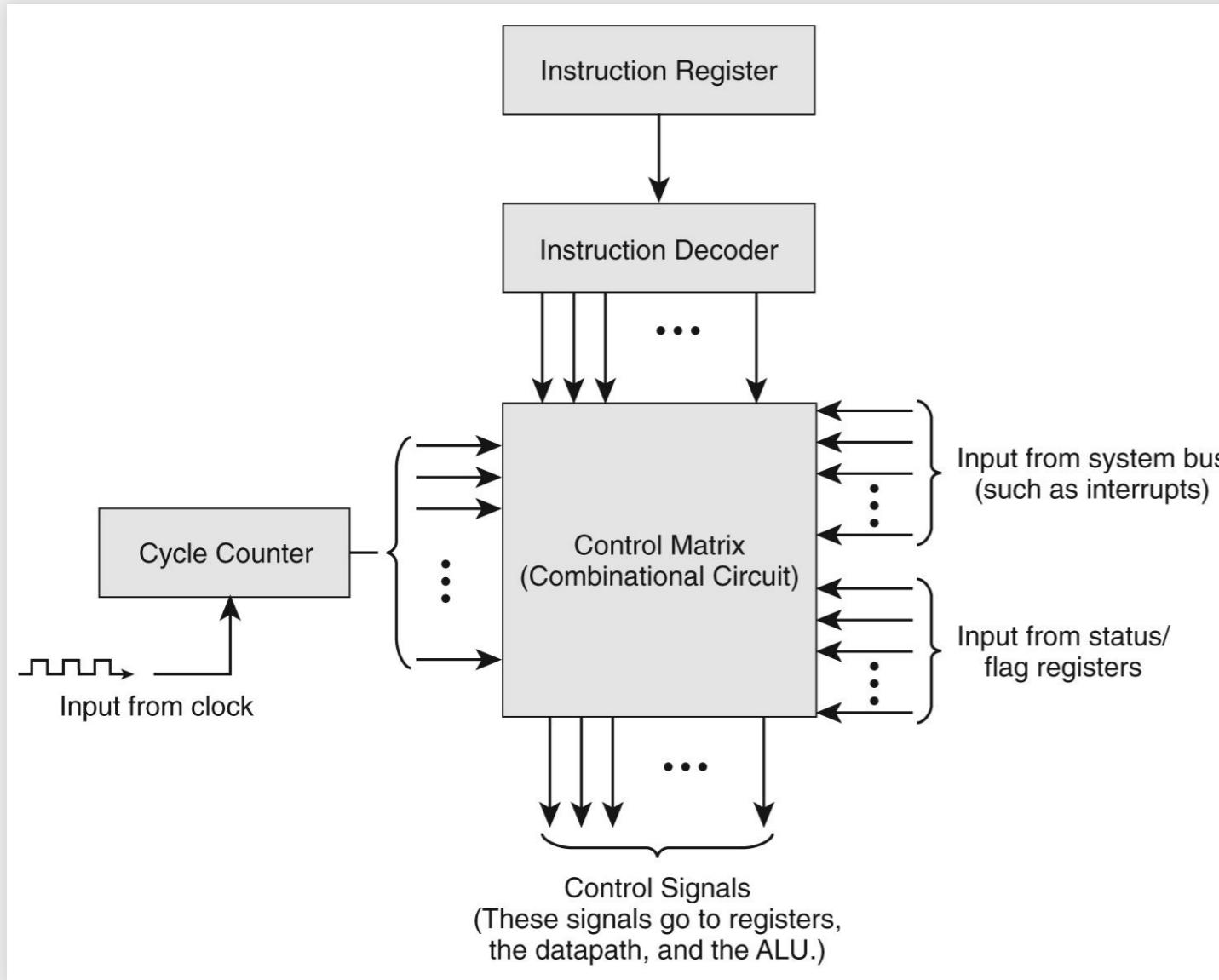
- 为了保证CPU的正确运行，需要通过**控制信号**来协调各部件的功能
- MARIE架构中需要哪些控制信号？
 - 总线: $P_0, P_1, P_2, P_3, P_4, P_5$
 - ALU: A_0, A_1
 - 时钟: T_0, \dots, T_7, C_r
 - PC: IncrPC
 - 内存: M_W, M_R
 - 替代数据源: L_{alt}
- 如何设置控制信号？

指令译码和执行：控制单元的实现

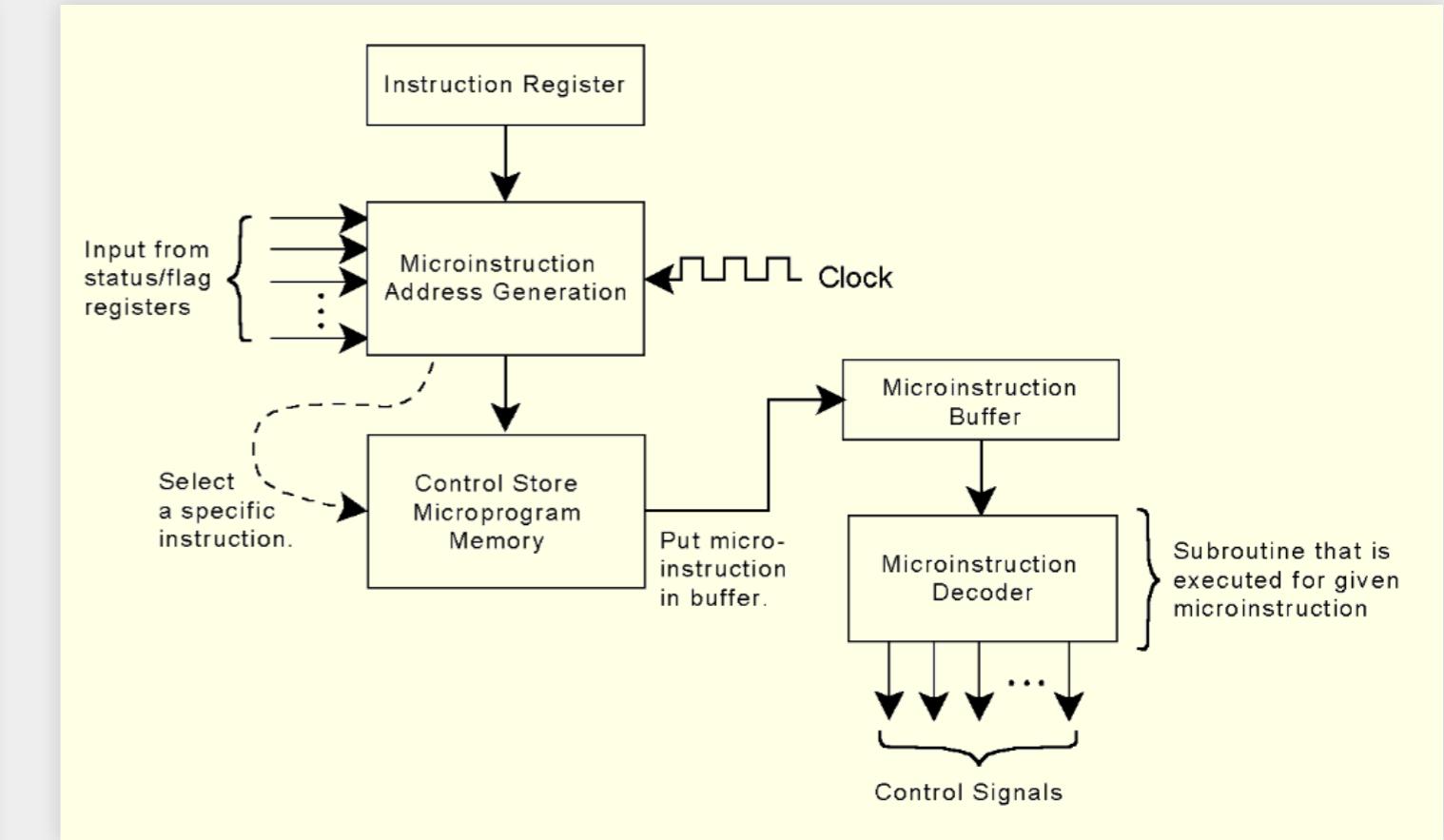
- 为了保证CPU的正确运行，需要通过**控制信号**来协调各部件的功能
- MARIE架构中需要哪些控制信号？
 - 总线: $P_0, P_1, P_2, P_3, P_4, P_5$
 - ALU: A_0, A_1
 - 时钟: T_0, \dots, T_7, C_r
 - PC: IncrPC
 - 内存: M_W, M_R
 - 替代数据源: L_{alt}
- 如何设置控制信号？
 - 在RTN中加入**信号模式**

信号模式	微操作
$P_3 P_2 P_1 P_0 T_3$	$\text{MAR} \leftarrow x$
$P_4 P_3 T_4 M_R$	$\text{MBR} \leftarrow M[\text{MAR}]$
$C_r A_0 P_5 T_5 L_{\text{alt}}$	$\text{AC} \leftarrow \text{AC} + \text{MBR}$

控制单元的实现



硬连线控制



微程序控制

现实中的体系结构

按照指令集设计分类

现实中的体系结构

按照指令集设计分类

- 复杂指令集计算机(Complex Instruction Set Computer, CISC): 指令数量多、长度各异

现实中的体系结构

按照指令集设计分类

- 复杂指令集计算机(Complex Instruction Set Computer, CISC): 指令数量多、长度各异
 - CISC代表: Intel体系结构(x86体系结构)

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

图片来源: <https://www.computing.dcu.ie/~humphrys/Notes/OS/hardware.html>

现实中的体系结构

按照指令集设计分类

- 复杂指令集计算机(Complex Instruction Set Computer, CISC): 指令数量多、长度各异
 - CISC代表: Intel体系结构(x86体系结构)
- 精简指令集计算机(Reduced Instruction Set Computer, RISC): 指令数量少, 所有指令长度相同

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

图片来源: <https://www.computing.dcu.ie/~humphrys/Notes/OS/hardware.html>

现实中的体系结构

按照指令集设计分类

- 复杂指令集计算机(Complex Instruction Set Computer, CISC): 指令数量多、长度各异
 - CISC代表: Intel体系结构(x86体系结构)
- 精简指令集计算机(Reduced Instruction Set Computer, RISC): 指令数量少, 所有指令长度相同
 - RISC代表: MIPS、ARM

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

图片来源: <https://www.computing.dcu.ie/~humphrys/Notes/OS/hardware.html>



第三讲结束

本期内容回顾

- 计算机模型2
 - 内存概述
 - 体系结构
 - 交叉存储器
 - 中断及中断处理
- 计算机模型3: MARIE机器
 - 体系结构
 - 指令集
 - 寄存器传输表示
 - 汇编语言
 - 控制单元的实现
 - 现实世界中的指令集

扩展阅读

- 内存的电路实现
 - <http://semiengineering.com/embedded-memory-impact-power-grids/>



Q & A