中国科学院大学研究生课程

高等数字集成电路分析与设计

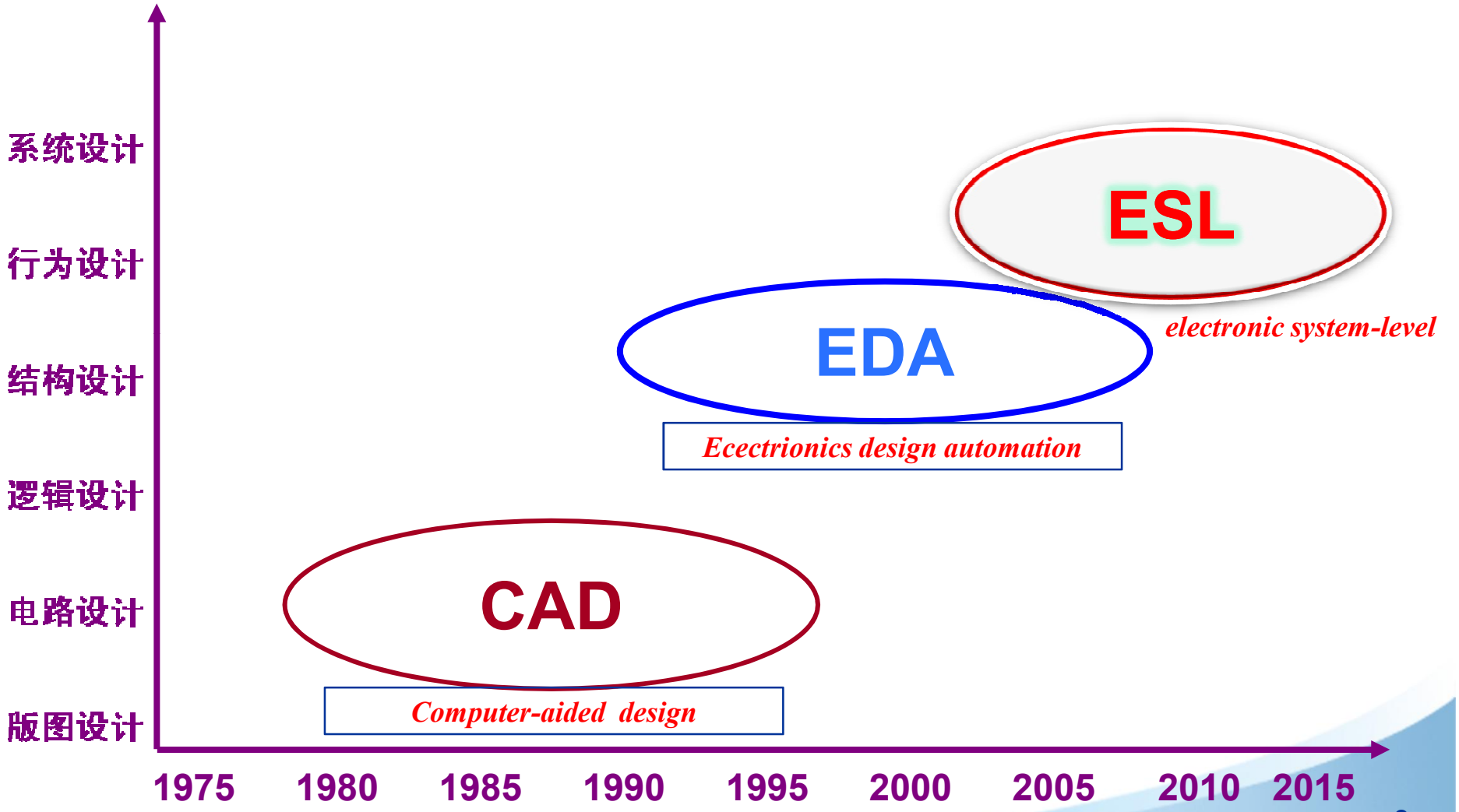
第八章 数字系统高层次综合

授课教师：王志君
电子邮件：wangzhijun@ime.ac.cn

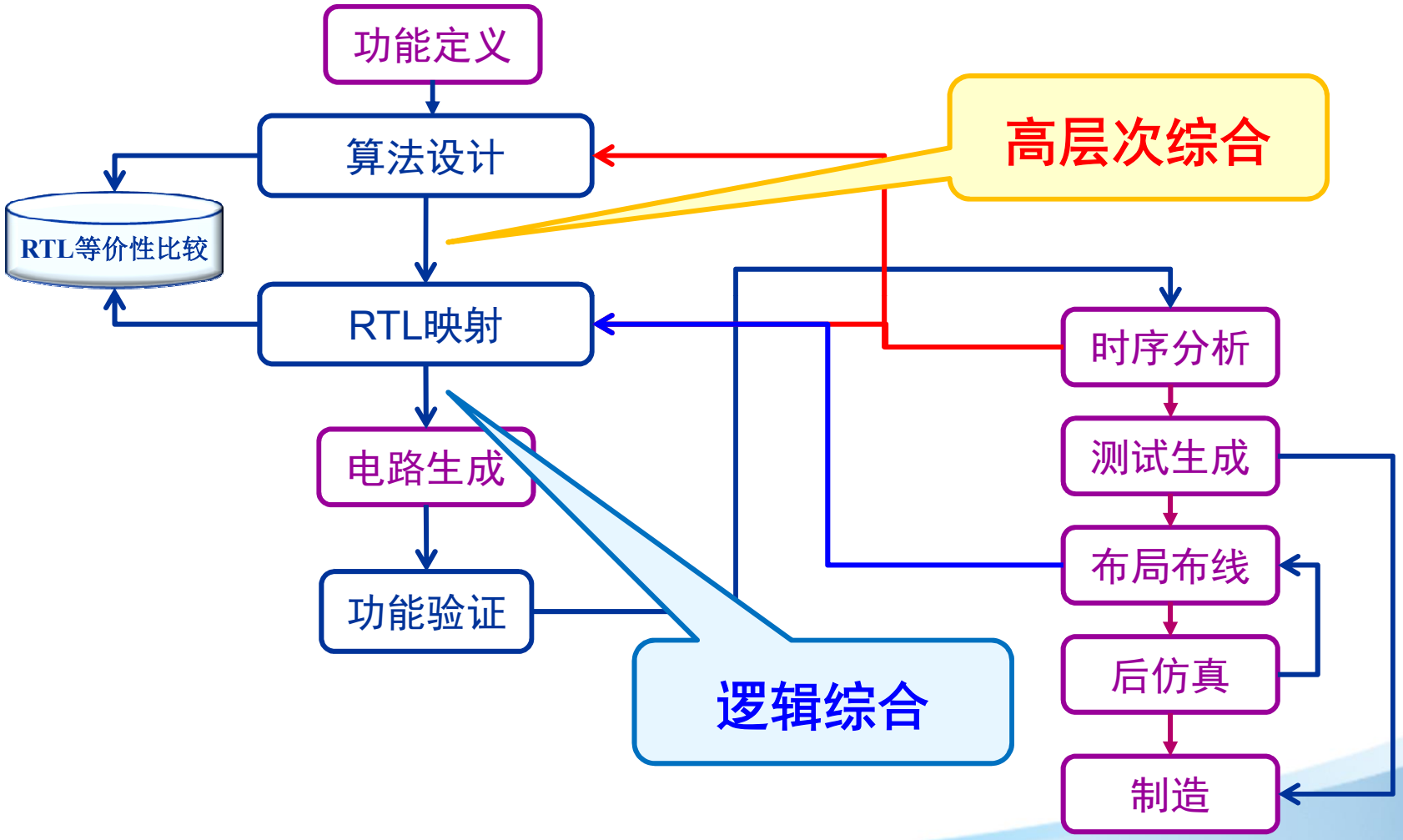
中国科学院微电子研究所

目录

集成电路设计方法学的发展



深亚微米集成电路设计流程



高层次综合的概念

- 高层次综合：将高层次语言转化成RTL代码的过程
  - 输入:
    - 高层次语言 (e.g., C/C++)
    - 高层次的逻辑描述或状态图表
  - 输出
    - RTL硬件描述语言
      - ✓ 数据路径部件
      - ✓ 控制路径部件
      - ✓ 互联结构
  - 高层次综合的约束:
    - 操作的前后相关性
    - 资源开销约束
    - 时间约束

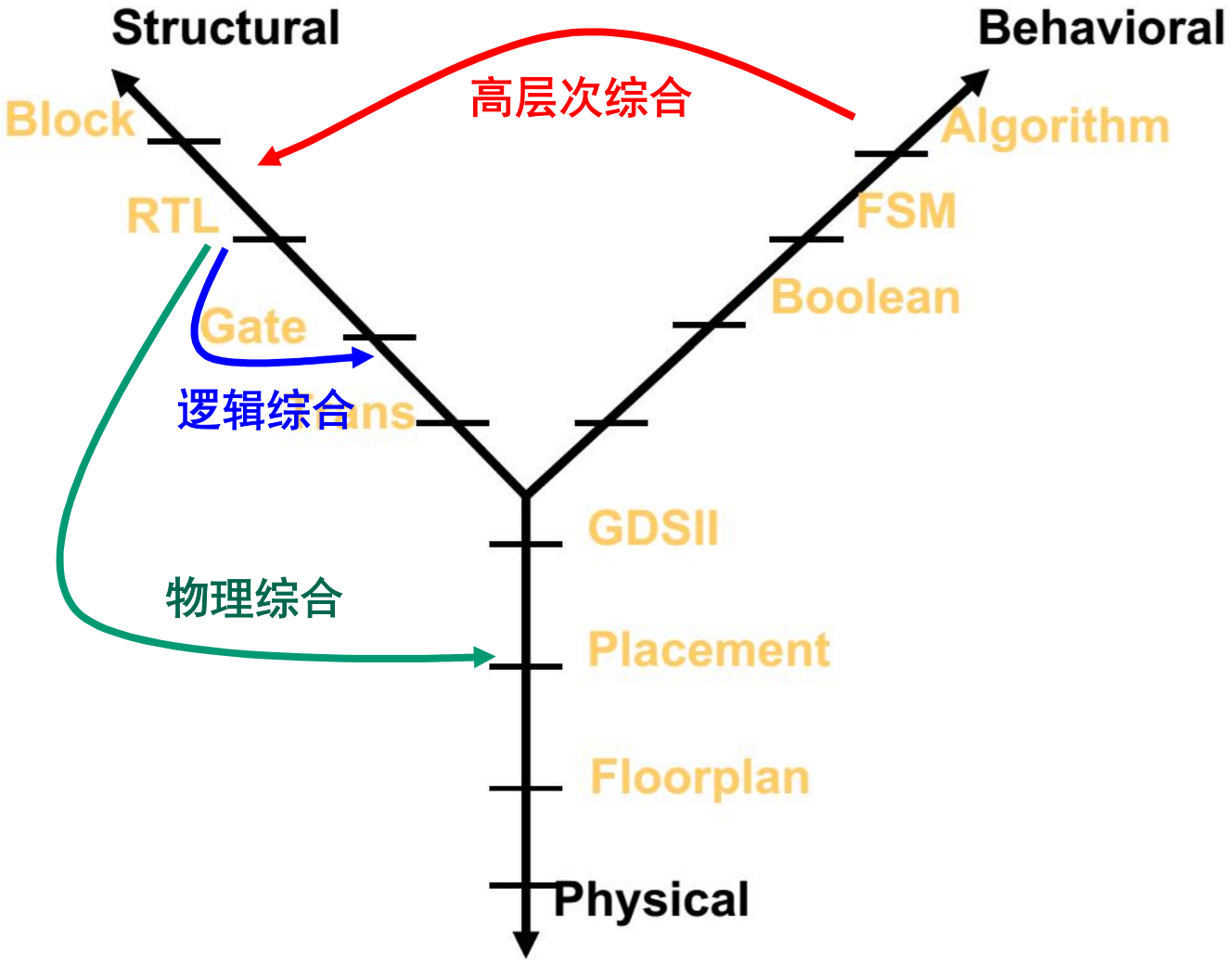
高层次综合的概念


描述方法
行为描述
结构描述
系统层
行为设计
处理器，存储器
开关网络
算法层
算法，
输入及输出
分系统设计
微架构层
功能单元
算术逻辑单元
状态机
逻辑层
逻辑方程组
门，触发器
电路层
转移功能表
晶体管网络

高层次综合的概念

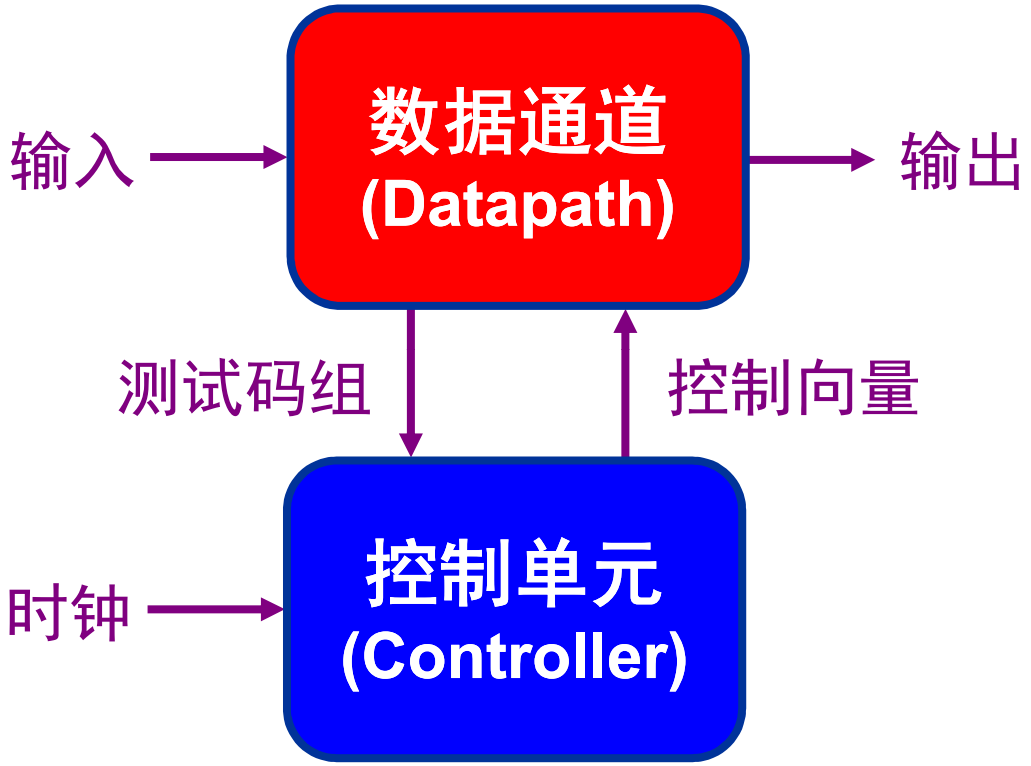
- 描述的抽象层次越高，与工艺的联系越少
- 描述通过模型来实现一个设计
- 设计本身要求描述所采用的模型要适合设计优化的要求；
- 工艺要求设计必须具备某种特点的风格而设计又试图通过风格来选定所需的实现工艺

高层次综合的概念

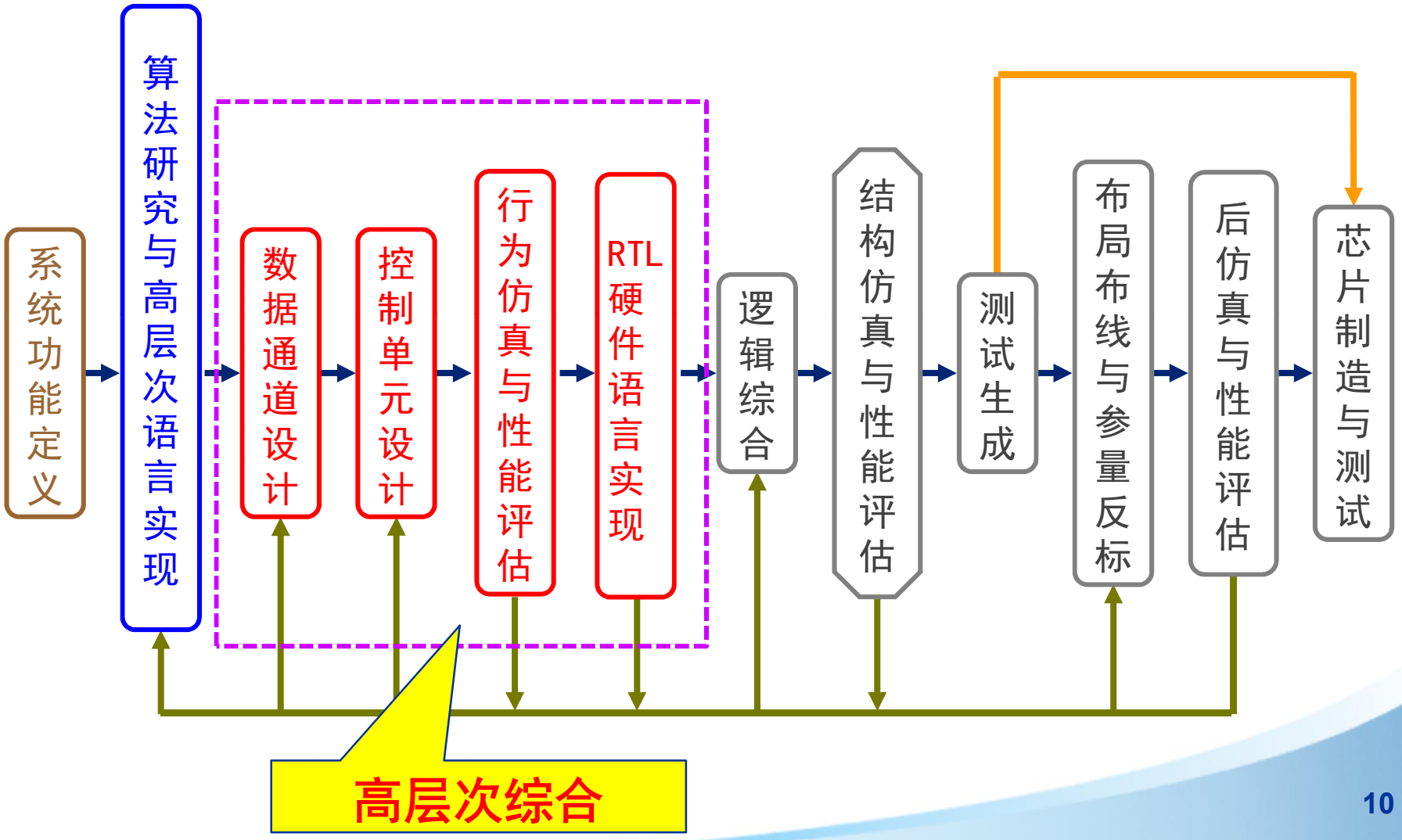

Structural
Behavioral
高层次综合
Block
Algorithm
RTL
FSM
Boolean
Gate
逻辑综合
GDSII
物理综合
Placement
Floorplan
Physical

高层次综合的概念

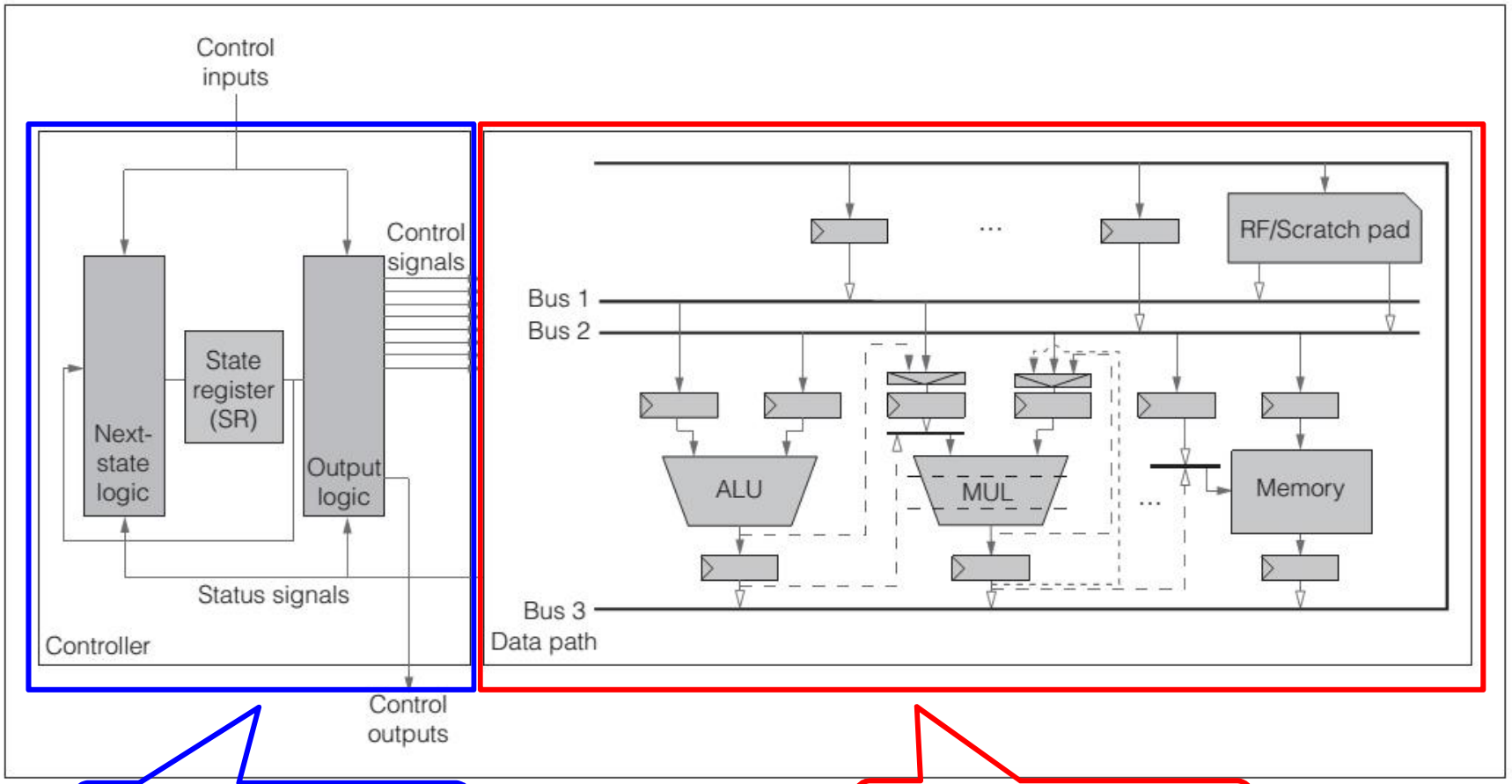
任意的数字电路
总是可以划分成
数据通道和
控制单元的组合



高层次综合的设计流程


系统功能定义
算法研究与高层次语言实现
数据通道设计
控制单元设计
行为仿真与性能评估
RTL硬件语言实现
逻辑综合
结构仿真与性能评估
测试生成
布局布线与参量反标
后仿真与性能评估
芯片制造与测试
高层次综合

高层次综合的典型架构

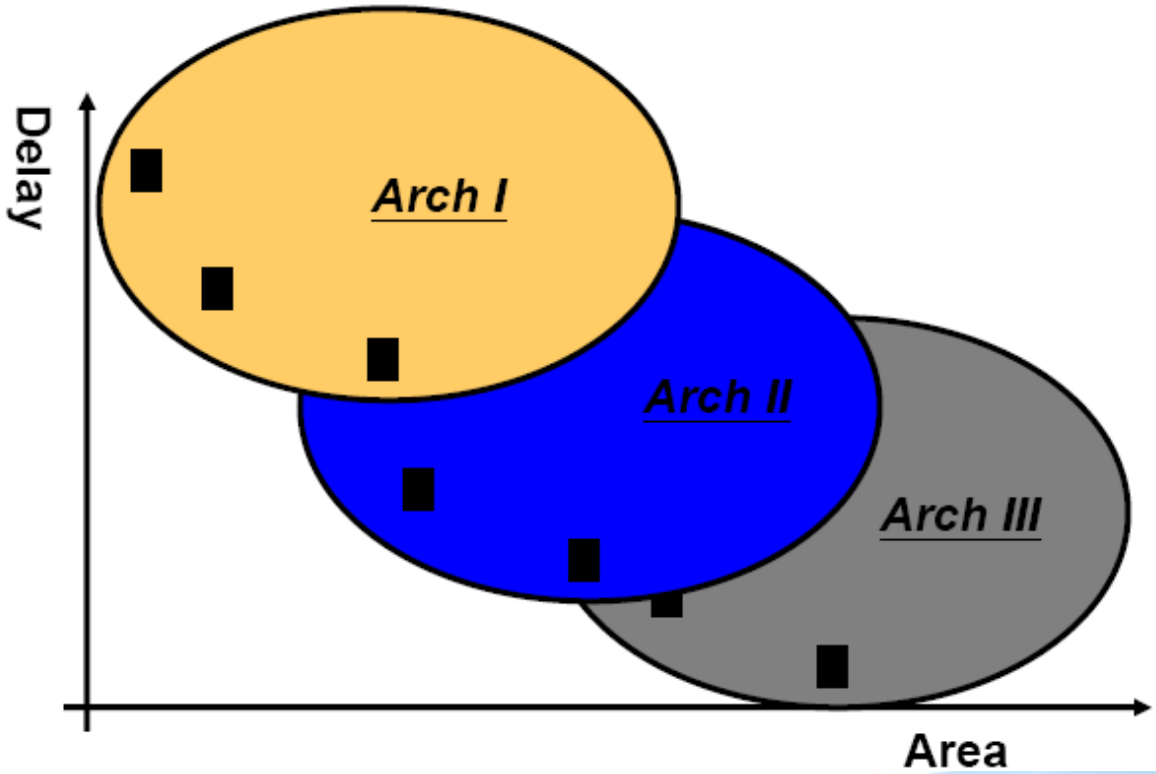


控制单元

数据通路

高层次综合的概念

- 设计空间的探索
  - 各种约束之间的权衡

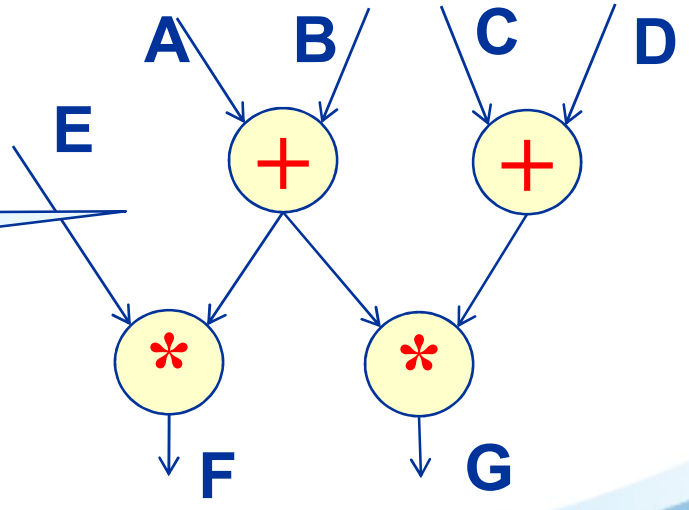


高层次综合举例

- 数据流分析
  - 共有项提取
  - 子模块检测
  - 初始流程图生成

```
int A, B, C, D, E, F, G;
F = E*(A+B);
G = (A+B)*(C+D);
…
```

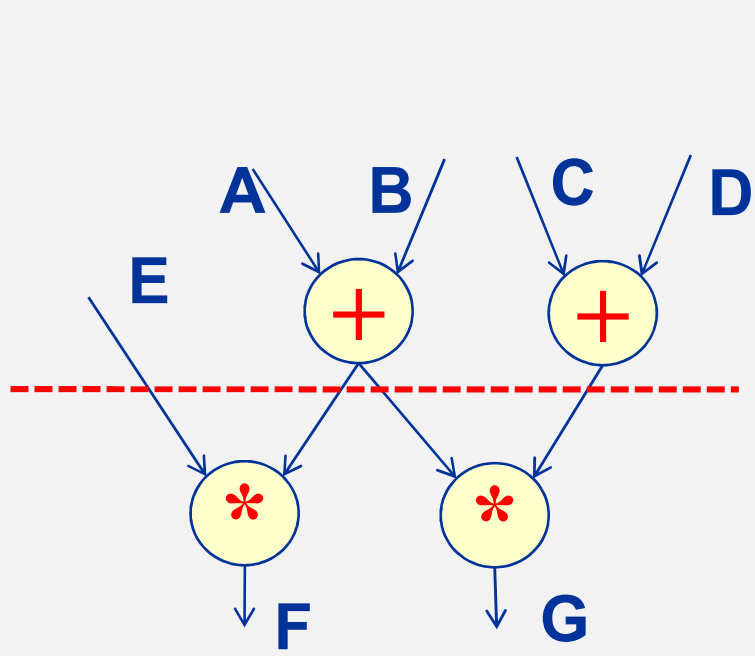
DFG：Data Flow Graph



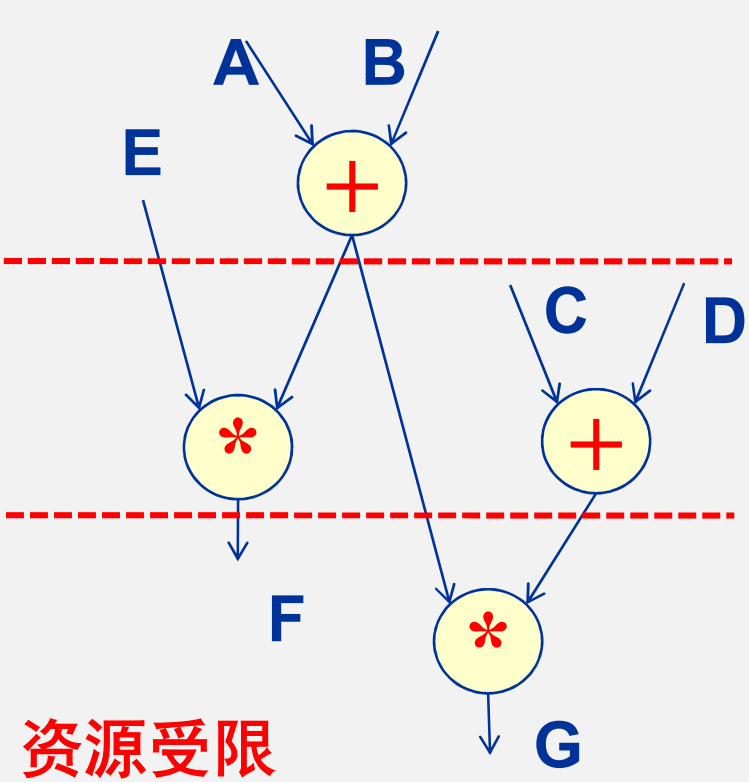
高层次综合举例

- 算子调度
  - 开销和速度的权衡
  - 流水划分

```
int A, B, C, D, E, F, G;
F = E*(A+B);
G = (A+B)*(C+D);
…
```



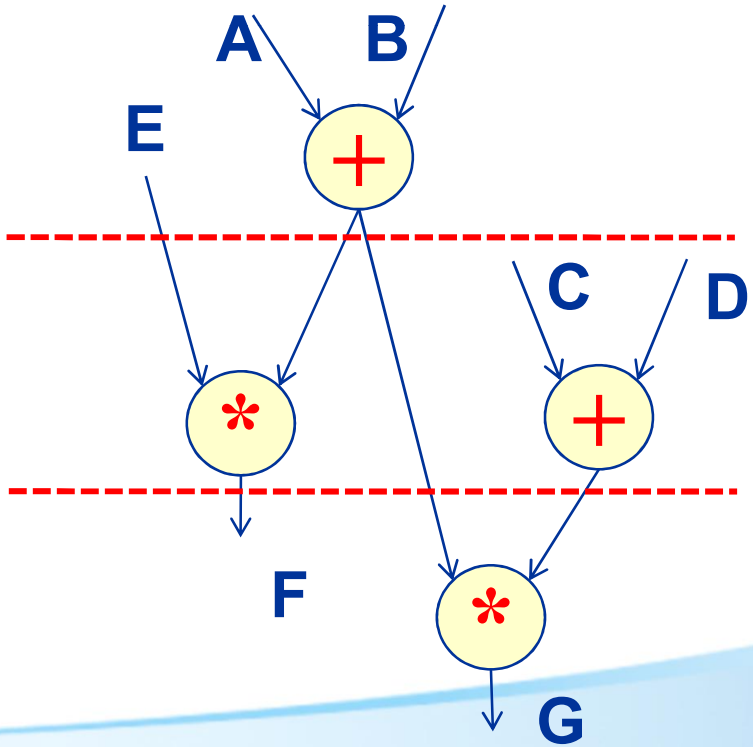
资源不受限

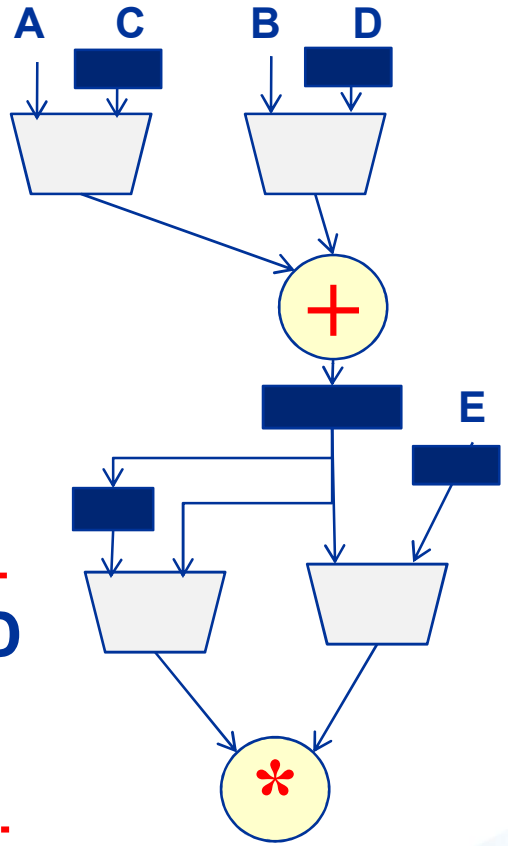


资源受限

高层次综合举例

- 数据路径规划
  - 操作数选取
  - 存储分配（寄存器、memory）
  - 内部互联生成





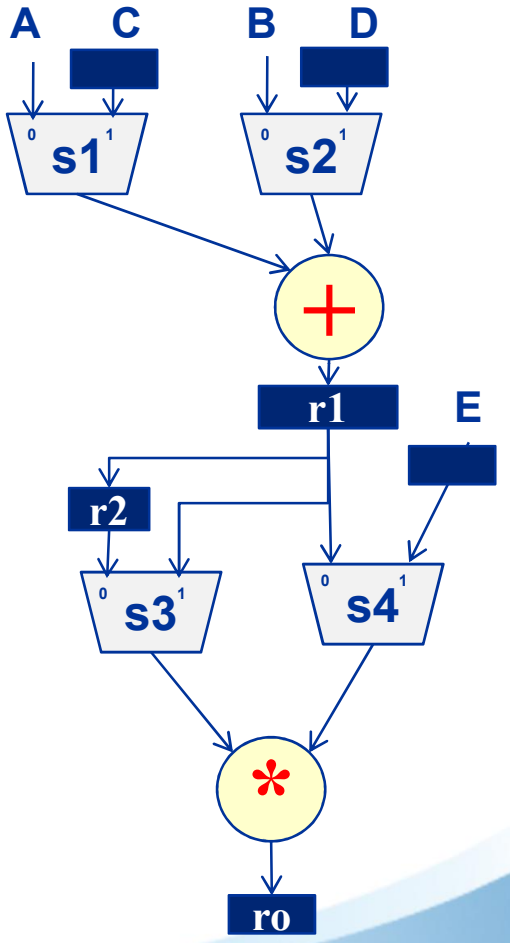

```
int A, B, C, D, E, F, G;
F = E*(A+B);
G = (A+B)*(C+D);
…
```

高层次综合举例

❖数据路径规划

➢控制码生成

第一拍：s1(0), s2(0), s3(x), s4(x)
第二拍： s1(1), s2(1), s3(1), s4(1)
第三拍： s1(0), s2(0), s3(0), s4(0)， ro(F)
第四拍： s1(1), s2(1), s3(1), s4(1)， ro(G)

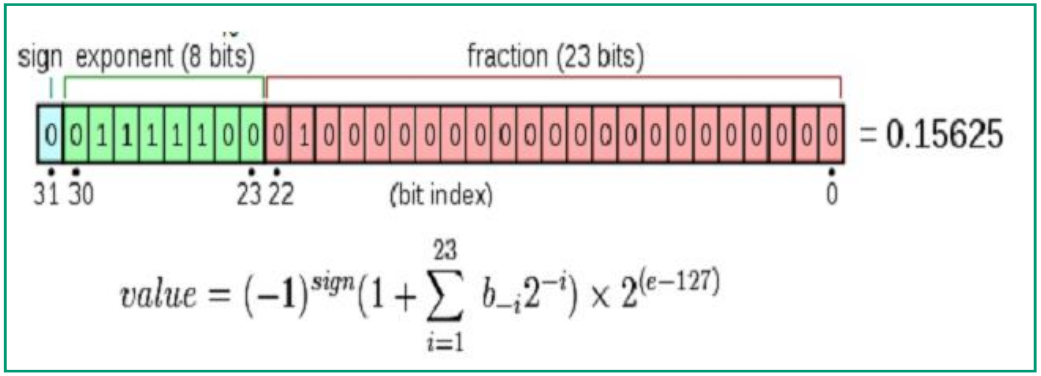


高层次综合前的准备

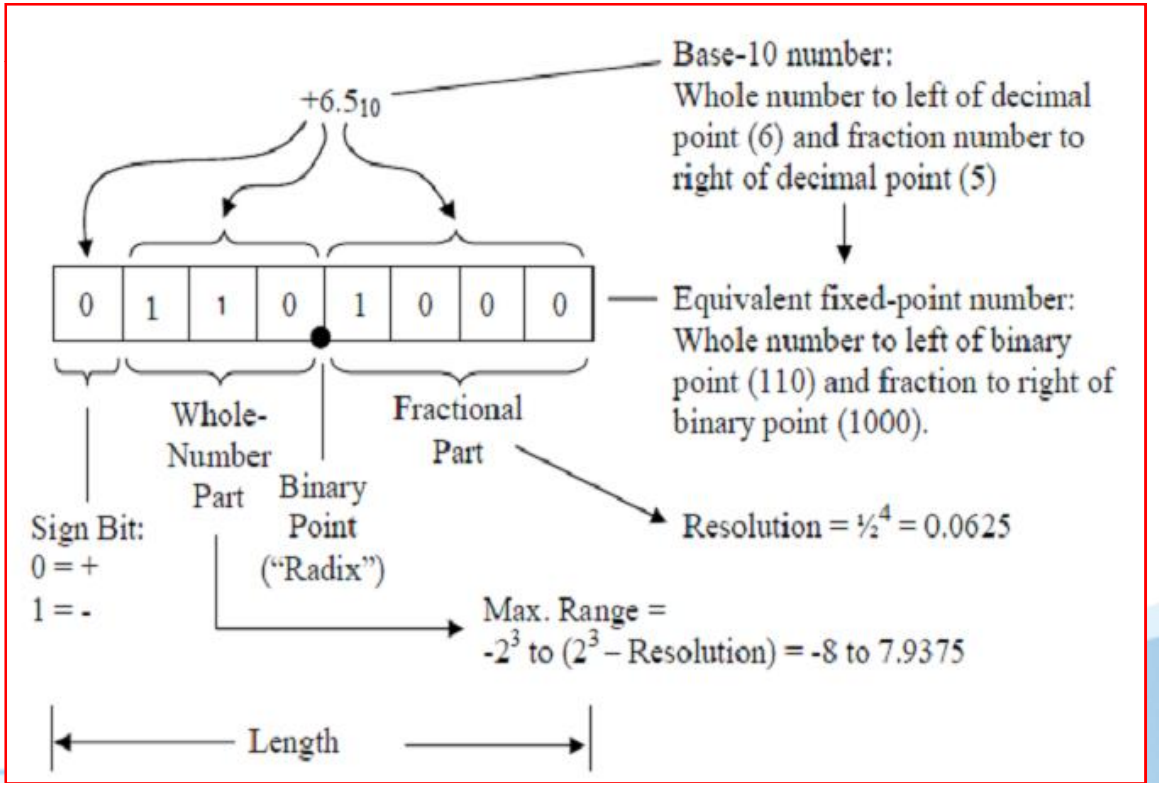
- 数据类型
  - 定点、浮点?
- 运算资源
  - 加法器，乘法器的个数
  - 寄存器的数量
- 性能估算
  - 数据量
  - 吞吐率
  - 频率

What is fix point data

- Float point



- Fix point



fix point VS. float point

Consideration	Floating Point	Fixed Point
RAM and ROM Consumption	↑	↓
Execution Speed	↓	↑
Hardware power consumption	↑	↓
Embedded Hardware Cost	↑	↓
Development time	↓	↑
Implementation complexity	↓	↑
Error Prone	↓	↑

浮点转定点

❖C语言中浮点转定点方法

➢Q：量化系数

$$X_q = (\text{int}) X_f * 2^Q$$

Q 表示	精度(近似)	十进制数表示范围
Q15	0.00002	-1≤X≤0.9999695
Q14	0.00005	-2≤X≤1.9999390
Q13	0.0001	-4≤X≤3.9998779
Q12	0.0002	-8≤X≤7.9997559
Q11	0.0005	-16≤X≤15.9995117
Q10	0.001	-32≤X≤31.9990234
Q9	0.002	-64≤X≤63.9980469
Q8	0.005	-128≤X≤127.9960938
Q7	0.01	-256≤X≤255.9921875
Q6	0.02	-512≤X≤511.9804375
Q5	0.04	-1024≤X≤1023.96875
Q4	0.08	-2048≤X≤2047.9375
Q3	0.1	-4096≤X≤4095.875
Q2	0.25	-8192≤X≤8191.75
Q1	0.5	-16384≤X≤16383.5
Q0	1	-32768≤X≤32767

高层次综合工具

- Cadence：Stratus
  - 能够从抽象的SystemC、C或C++模型中快速地设计和验证RTL实现
- Mentor Graphics：Catapult
  - 能够在 C++/SystemC 级验证收敛（实现 C++/SystemC signoff 的重大步骤）的基础上实现快速且可预测 RTL 验证收敛
- Synopsys：Synphony
  - 集成了matlab语言和基于模型的综合法，提供从m语言到优化rtl的自动流程

目录

- 高层次综合基本概念
- 数据通道设计
  - 算子调度 (scheduling)
  - 资源分配 (allocation)
  - 控制码及连线网络生成
- 控制通道设计
- Loop的处理
- 性能评估

数据通道设计需要考虑的问题

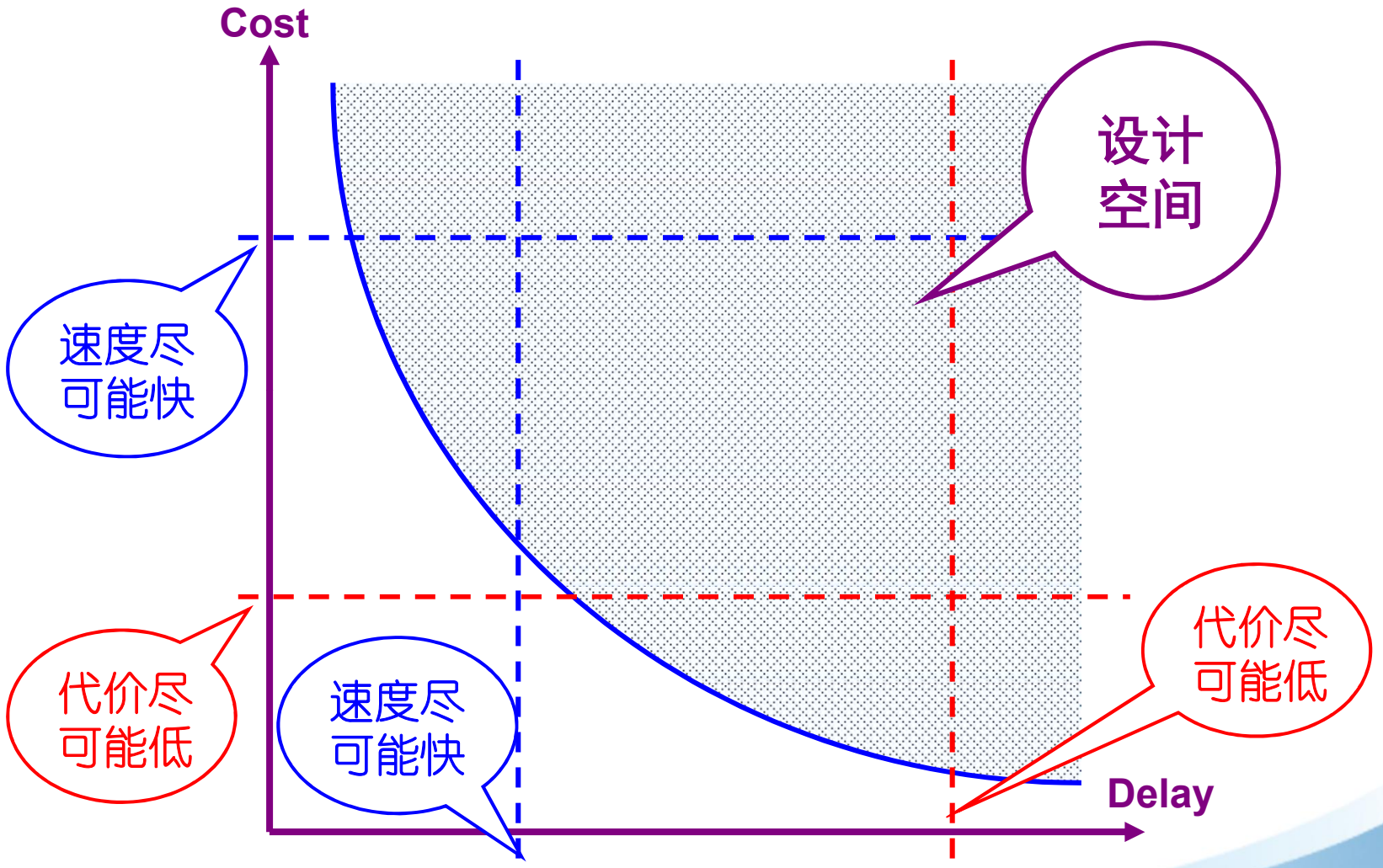
- **Scheduling**
  - 把每个操作映射到一个合适的时间周期中
- **Resource Allocation**
  - 选取合适数量的硬件资源以及硬件功能模块进行最终实现
- **Module Binding**
  - 把相应的操作和实际功能模块对应起来
  - 通过连线网络和控制码实现互联

算子调度

- 算子调度确定每个算子执行的时间位置
- 给每个算子的执行分配一个控制步
- 优化控制单元结构
- 最简连线网络结构
- 最少的存储单元
- 最少的资源

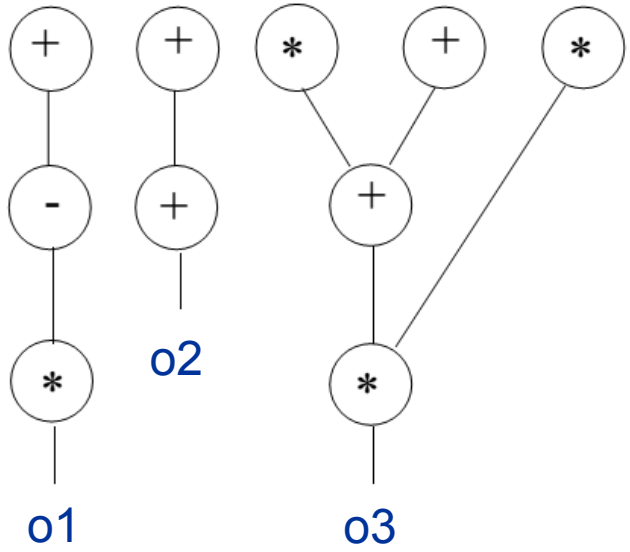
Scheduling

算子调度



算子调度

- **Resource-constrained (RC) scheduling**
  - 在资源受限的情况下找到一个最优的且符合约束的方案
- **Time-constrained (TC) scheduling**
  - 在延时最小情况下寻找最优方案

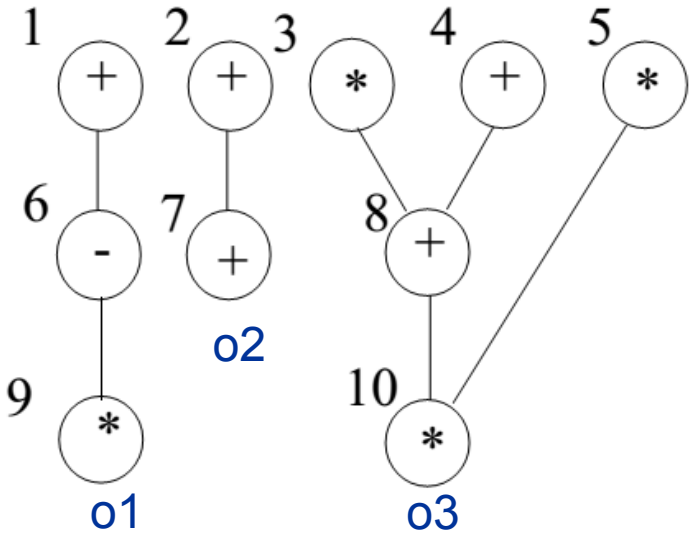



```
a = i1 + i2;
o1 = (a - i3) * 3;
o2 = i4 + i5 + i6;
d = i7 * i8;
g = d + i9 + i10;
o3 = i11 * 7 * g;
```

RC Scheduling

❖ ASAP: As soon as possible

```
a = i1 + i2;
o1 = (a - i3) * 3;
o2 = i4 + i5 + i6;
d = i7 * i8;
g = d + i9 + i10;
o3 = i11 * 7 * g;
```



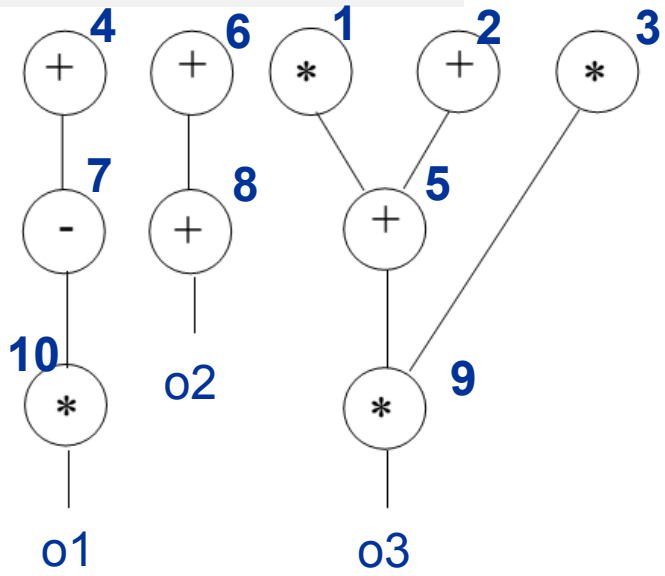
控制步	调度方案		
1	+(1)		*(3)
2		+(2)	*(5)
3			+(4)
4	-(6)		
5	*(9)	+(7)	
6			+(8)
7			*(10)

1 adder, 1 multiplier

RC Scheduling

❖ ASAP: As soon as possible

```
a = i1 + i2;
o1 = (a - i3) * 3;
o2 = i4 + i5 + i6;
d = i7 * i8;
g = d + i9 + i10;
o3 = i11 * 7 * g;
```



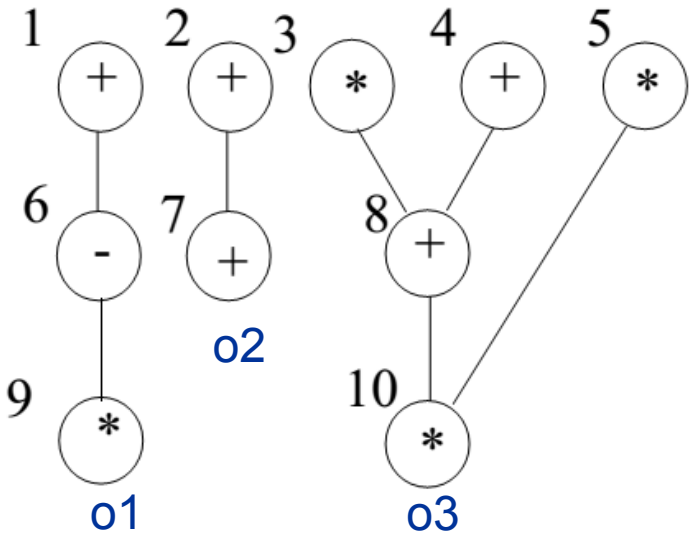
控制步	调度方案		
1			*(1) +(2)
2	+(4)		*(3)
3			+(5)
4		+(6)	*(9)
5	-(7)		
6	*(10)	+(8)	

1 adder, 1 multiplier

RC Scheduling

❖ ALAP: As late as possible

```
a = i1 + i2;
o1 = (a - i3) * 3;
o2 = i4 + i5 + i6;
d = i7 * i8;
g = d + i9 + i10;
o3 = i11 * 7 * g;
```



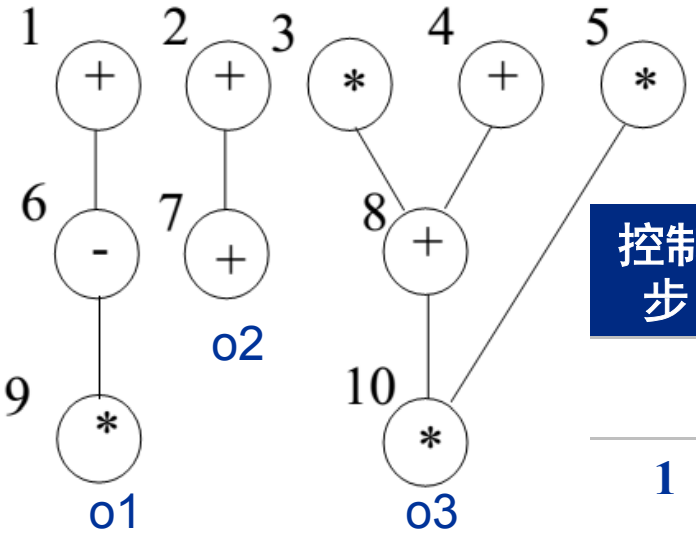
控制步	调度方案		
1	+(1)		
2		+(2)	
3			+(4) *(3)
4	-(6)		*(5)
5	*(9)		+(8)
6		+(7)	*(10)

1 adder, 1 multiplier

ASAP、ALAP特点

- 易于实现
- 算法的时间特性接近于线性函数
- 调度方案的优化程度取决于算子的排列顺序
- 要取得最优化的结果所需的时间为 t = (N0！)，这里N0!表示N0的全排列

关键路径调度算法



控制步	调度方案				
	步骤1	步骤2	步骤3	步骤4	步骤5
1	+(1)	+(1)*(3)	+(1)*(3)	+(1)*(3)	+(1)*(3)
2	+(6)	+(6) *(5)	+(6) *(5)	+(6) *(5)	+(6) *(5)
3	*(9)	*(9)+(4)	*(9)+(4)	*(9) +(4)	*(9)+(4)
4			+(8)	+(8)	+(8)
5				*(10) +(2)	*(10) +(2)
6					+(7)

关键路径调度算法

- 易于实现
- 与调度过程的算子顺序无关
- 算法的时间特性为O(N0Nc)，N0为算子数量，Nc为约束的数量
- 寻找关键路径需要额外的时间
- 并不能保证调度结果的最优性

算子的自由度

- 关键路径上的算子的自由度为“0”，算子自由度为“0”的算子位置已经确定。
- 非关键路径上的算子自由度给出有可能的算子调度方案数量的上界
- 一个算子的自由度是在该算子ALAP调度方案中所处的位置与其在ASAP调度方案中所处的位置之差

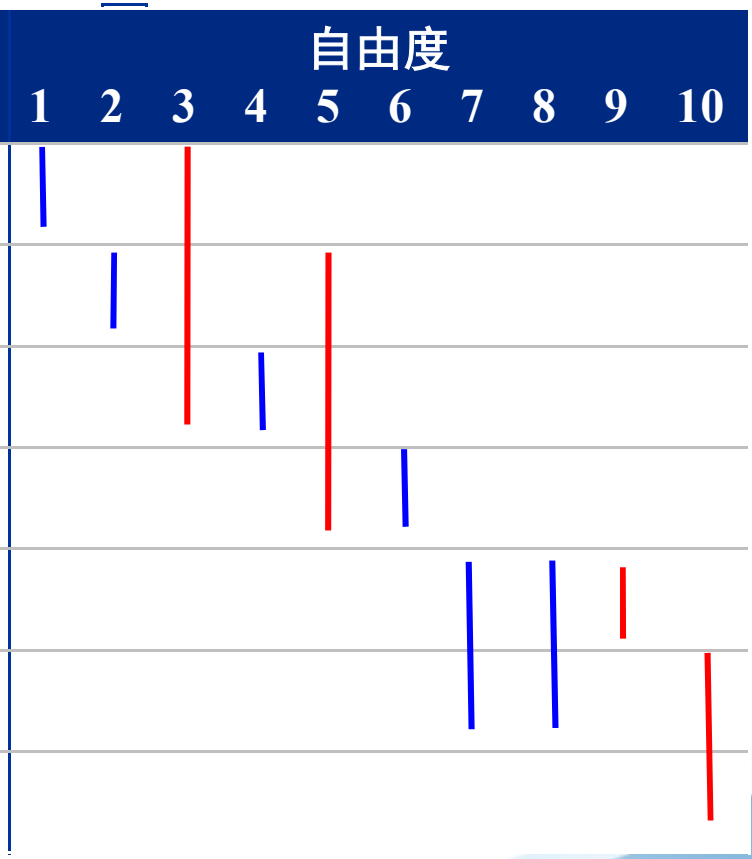
算子的自由度

控制步	ASAP			ALAP			自由度 1 2 3 4 5 6 7 8 9 10
1	+(1)		*(3)	+(1)			
2		+(2)	*(5)		+(2)		
3			+(4)			+(4) *(3)	
4	−(6)			−(6)		*(5)	
5	*(9)	+(7)		*(9)		+(8)	
6			+(8)		+(7)	*(10)	
7			*(10)				

基于自由度的的调度算法

自由度小的算子被优先安排

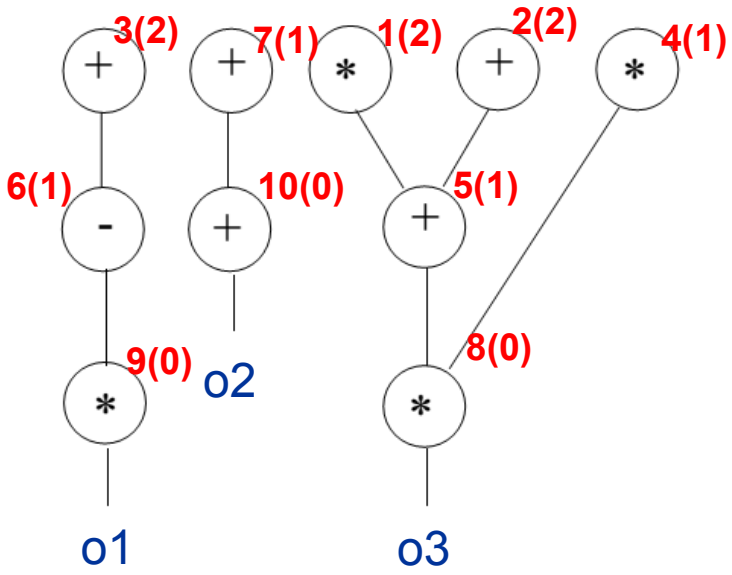
控制步	第一步	第二步	第三步
1	+(1)	+(1)	+(1) *(3)
2	+(2)	+(2)	+(2) *(5)
3	+(4)	+(4)	+(4)
4	−(6)	−(6)	−(6)
5	*(9)	*(9) +(8)	*(9) +(8)
6		+(7) *(10)	+(7) *(10)
7			



表格调度算法（list scheduling）

将一个描述中的所有操作按某种顺序进行排列并在算子调度过程中按照这种顺序选择算子进行安排。算子调度过程可以考虑各种约束的限制

设：算子的顺序按它们到描述末端的距离从大到小排列，且关键路劲优先



控制步	调度方案
1	+(2) *(1)
2	+(3) *(4)
3	+(5)
4	−(6) *(8)
5	*(9) +(7)
6	+(10)

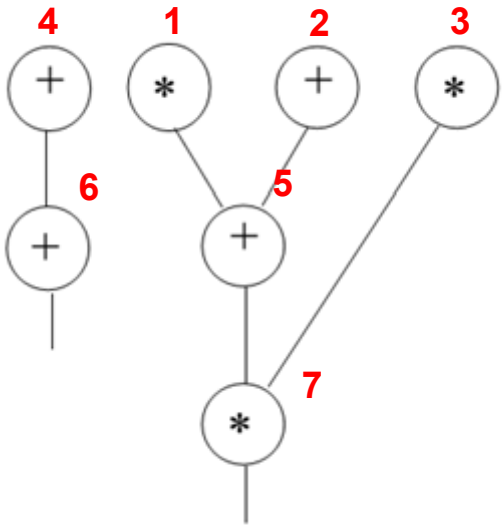
表格调度算法（list scheduling）

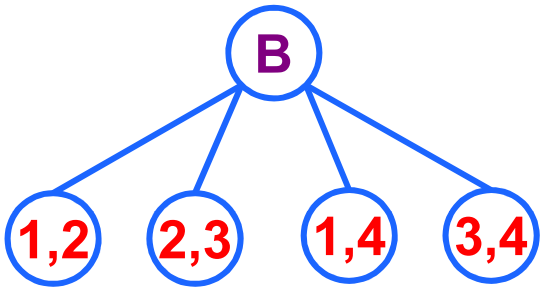
- 易于实现
- 优化程度取决于算子的排列顺序
- 通过选择算子的优先级别在大多数情况下可以获得优化的算子排列顺序
- 算法的时间特性近似于线性函数
- 易于引入资源约束

- 不能总保证调度结果的最优性
- 确定好的算子优先级别函数比较困难

分支调度算法

对节点进行分组，通过分支枚举方式获得最优解



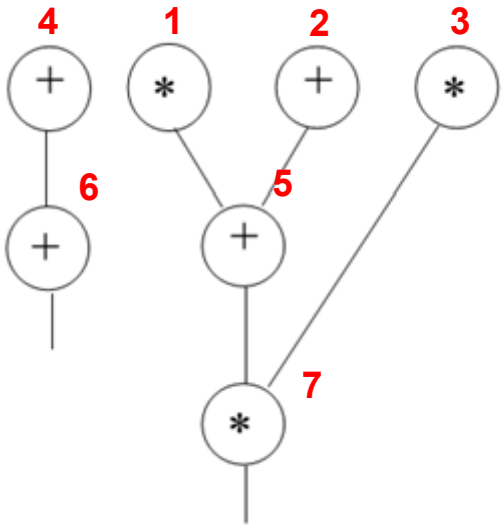


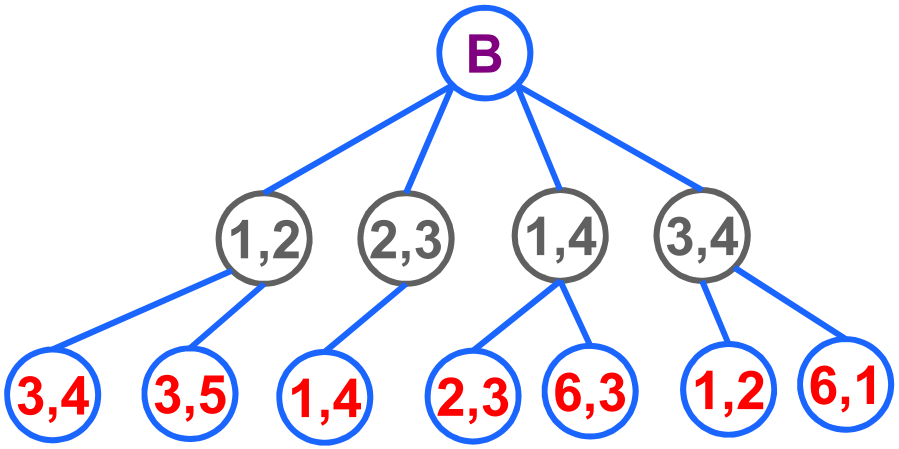
一个乘法器

一个加法器

分支调度算法

对节点进行分组，通过分支枚举方式获得最优解



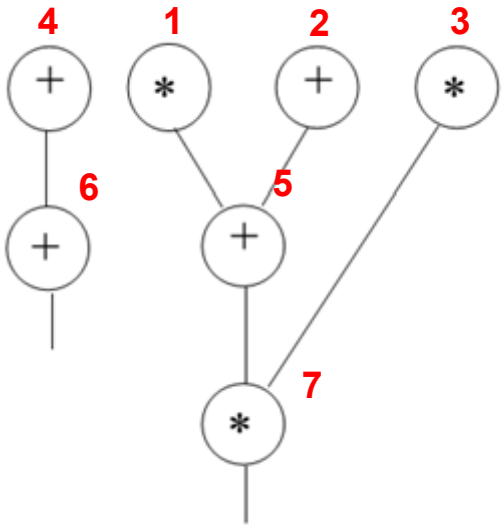


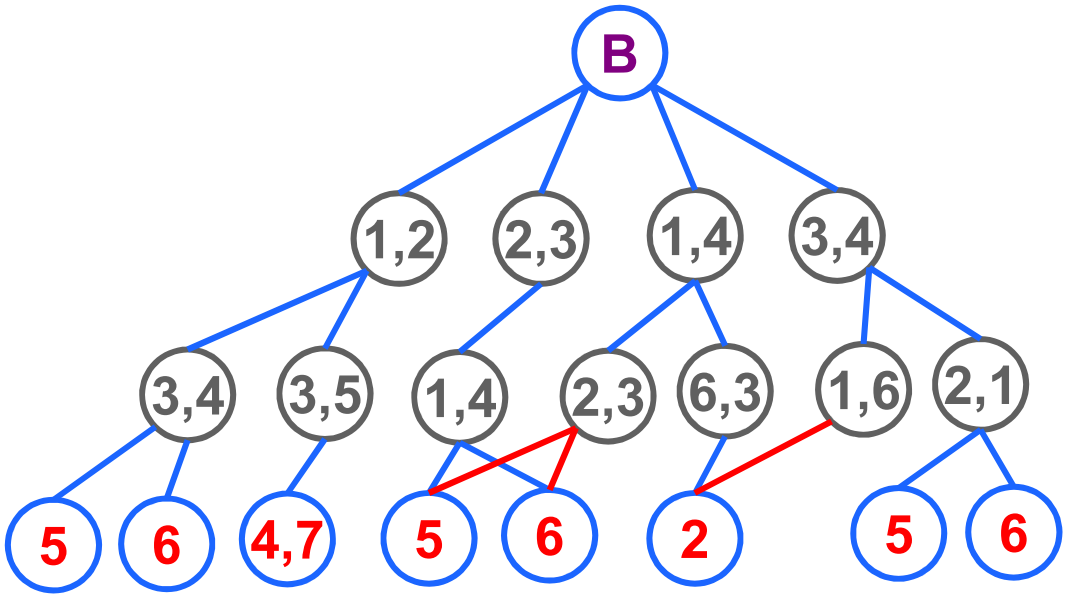
一个乘法器

一个加法器

分支调度算法

对节点进行分组，通过分支枚举方式获得最优解



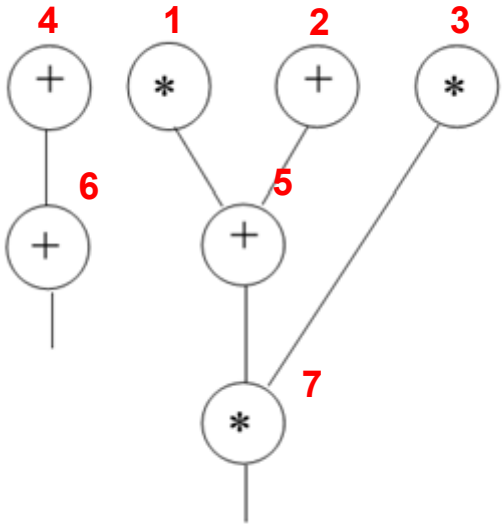


一个乘法器

一个加法器

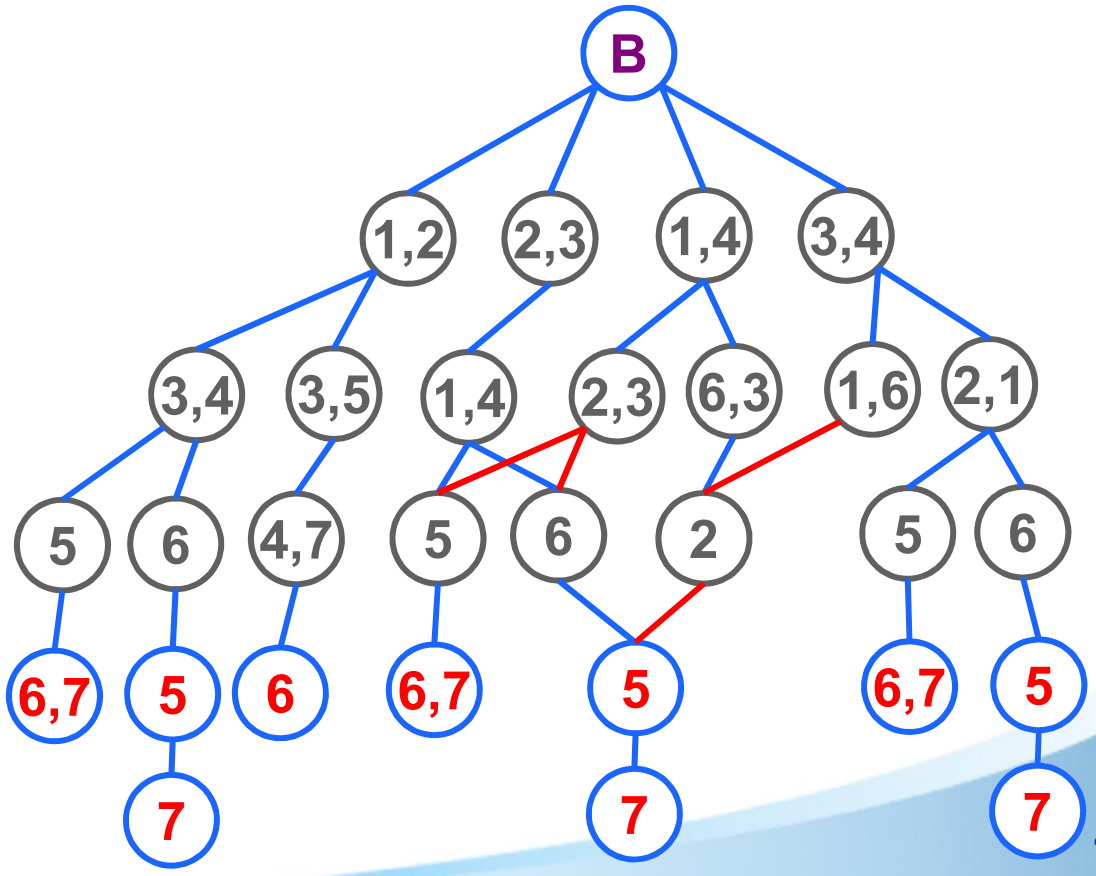
分支调度算法

对节点进行分组，通过分支枚举方式获得最优解

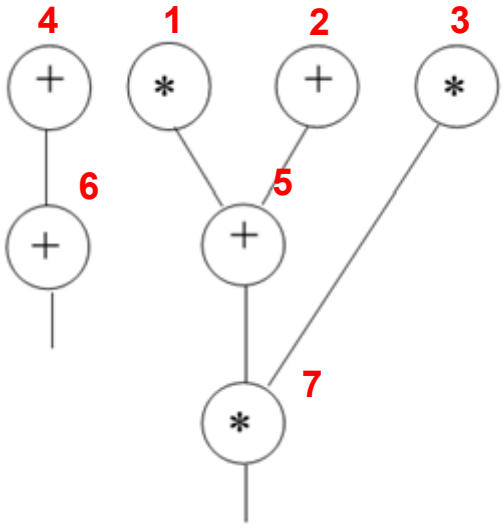


一个乘法器

一个加法器

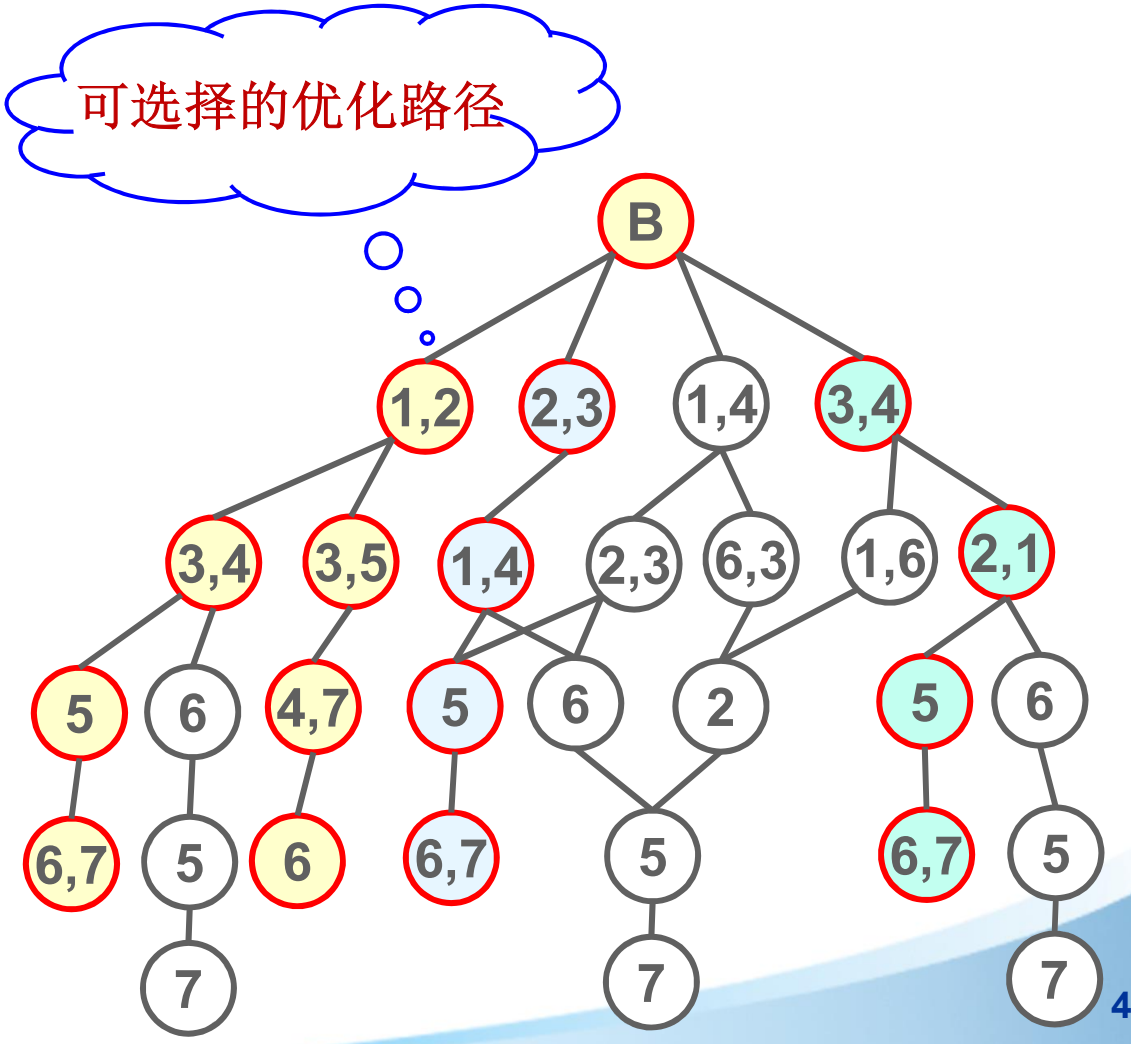


分支调度算法



一个乘法器

一个加法器



分支与边界调度算法

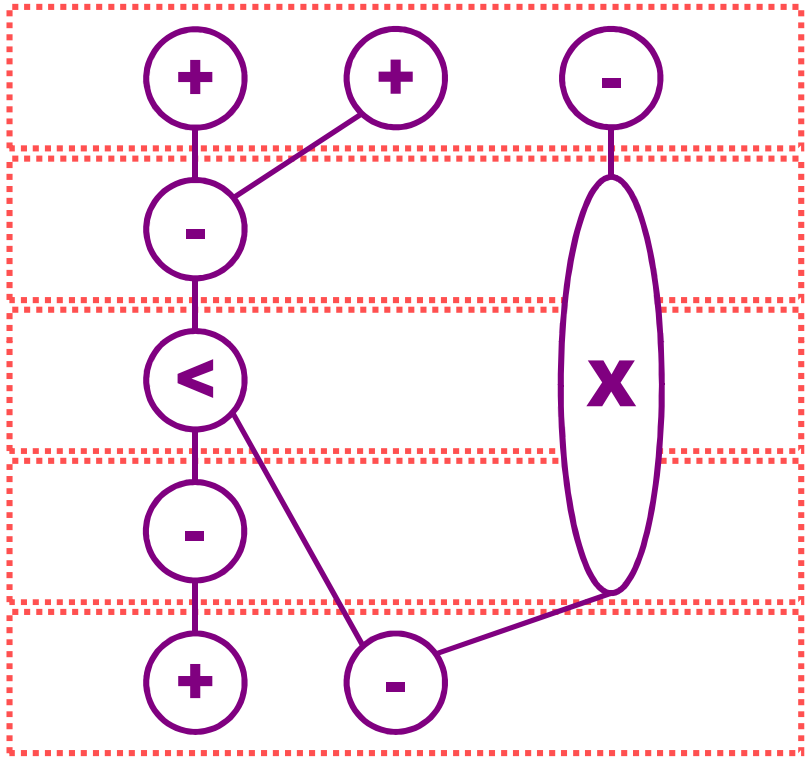
- 穷举型算法
- 在大多数情况下可以保证得到最优化的调度结果

- 算法的时间特性近似于指数函数
- 算法的实现复杂
- 需要比较大的存储空间

多周期调度

那些延时比较大的算子可以占用多个控制步

多周期调度使得控制步的步长可以不必取决于延时最大的算子



调度算法小结

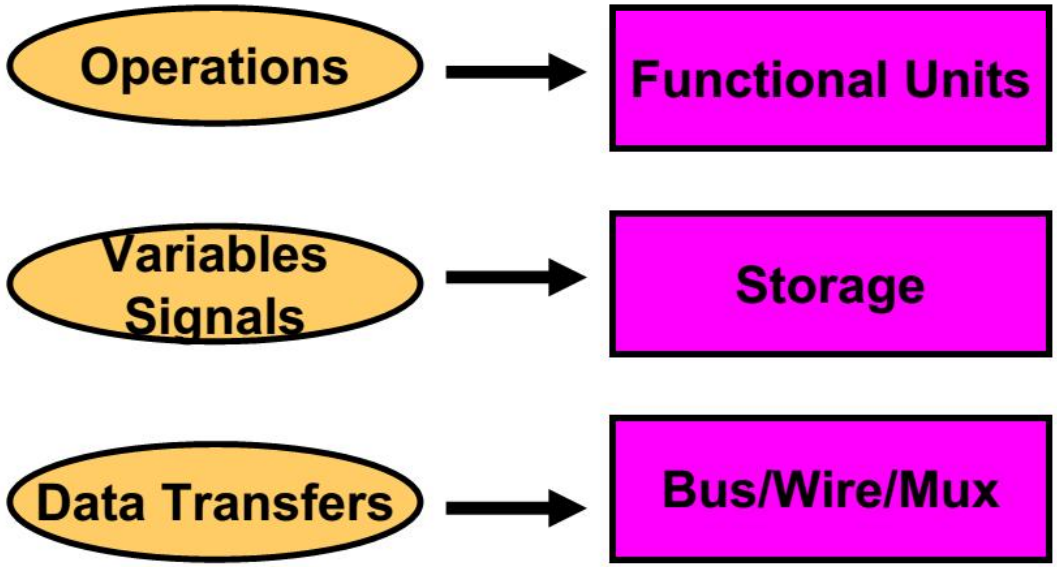
- 获得最优结果的代价是对结论空间进行全面探索
- 调度方案的优化程度取决于算子被安排的控制步
- 调度方案的优化程度取决于算子被调度的顺序
- 前一个算子的调度影响到后续算子的调度
- 每一个算子的调度只能做到局部优化

目录

- 高层次综合基本概念
- 数据通道设计
  - 算子调度 (scheduling)
  - 资源分配 (allocation)
  - 控制码及连线网络生成
- 控制通道设计
- Loop的处理
- 性能评估
- 高层次综合工具简介

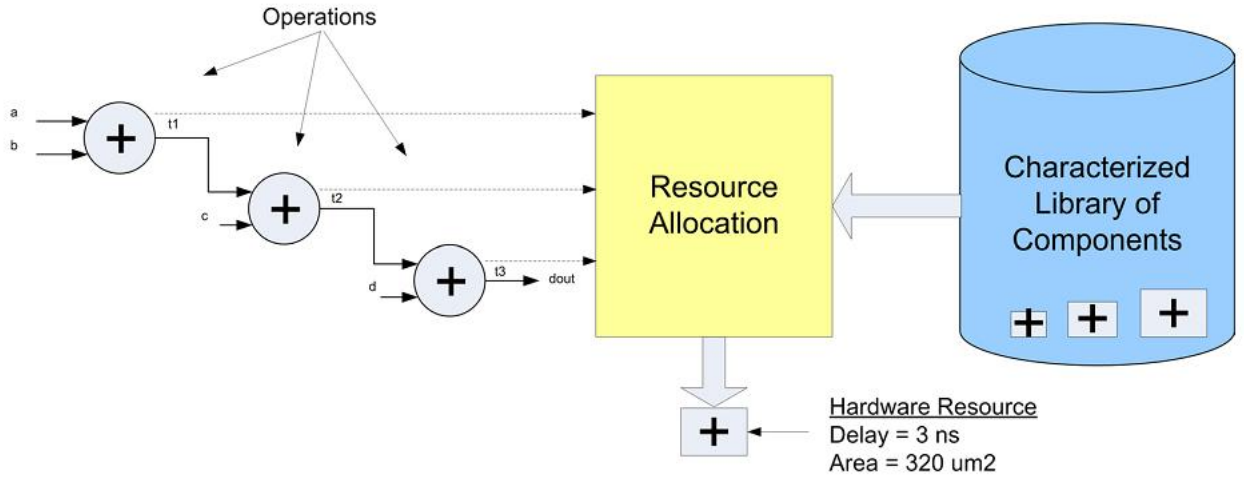
资源分配（Allocation）

- 给每个算子分配一个资源
- 给每个变量分配一个存储单元
- 提供资源与存储单元之间的数据通道

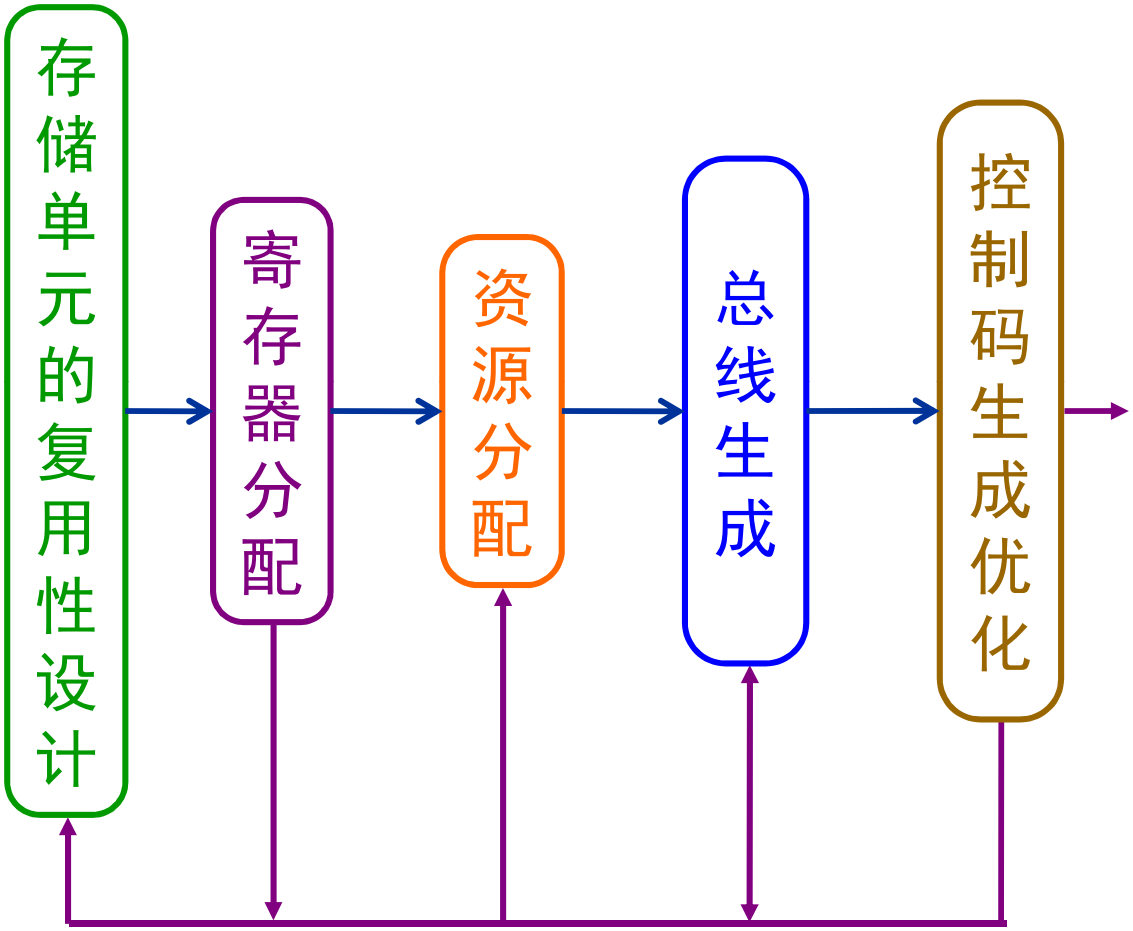


资源分配（Allocation）

- 资源分配过程中需要考虑
  - 硬件模块的速度
  - 硬件模块面积开销

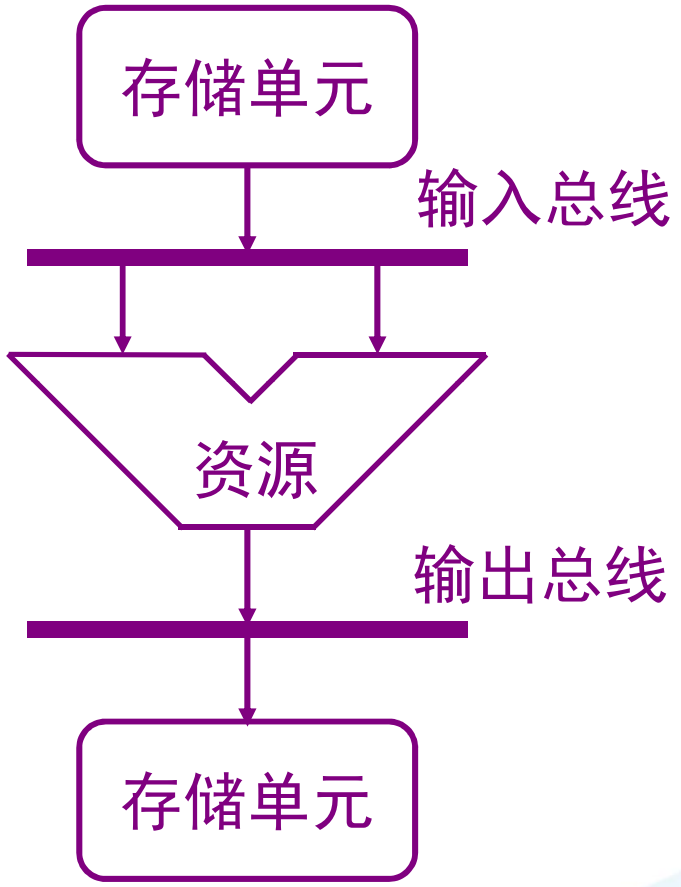


资源分配的流程



数据通道通用结构

存储单元：存放变量
资源：运算
总线：数据传输

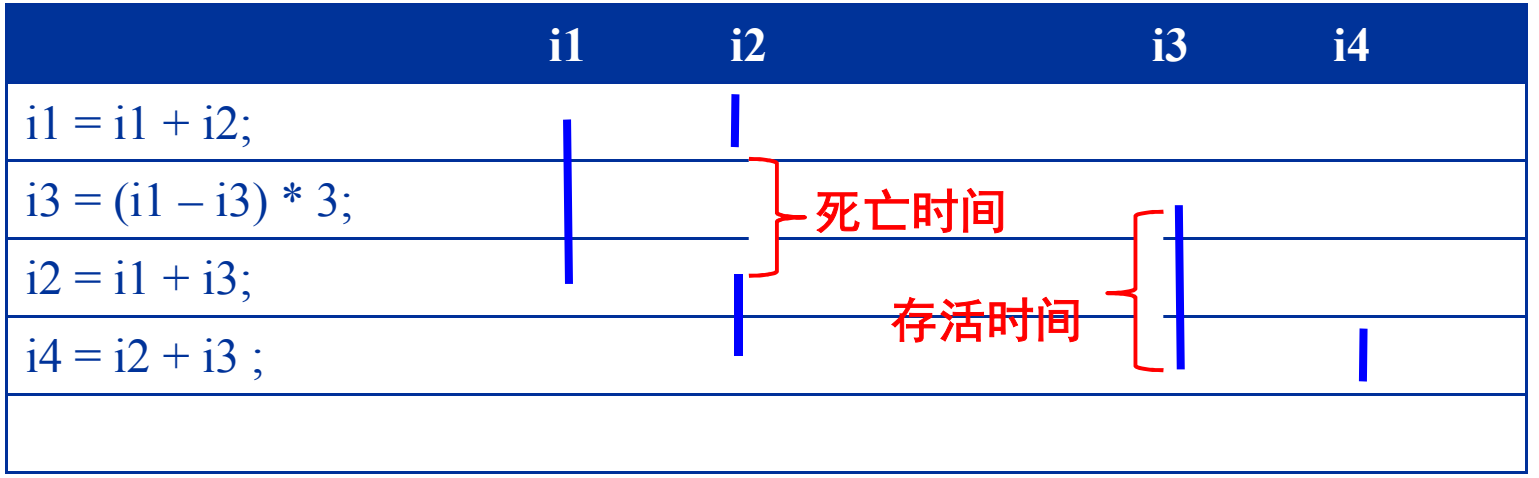


优化

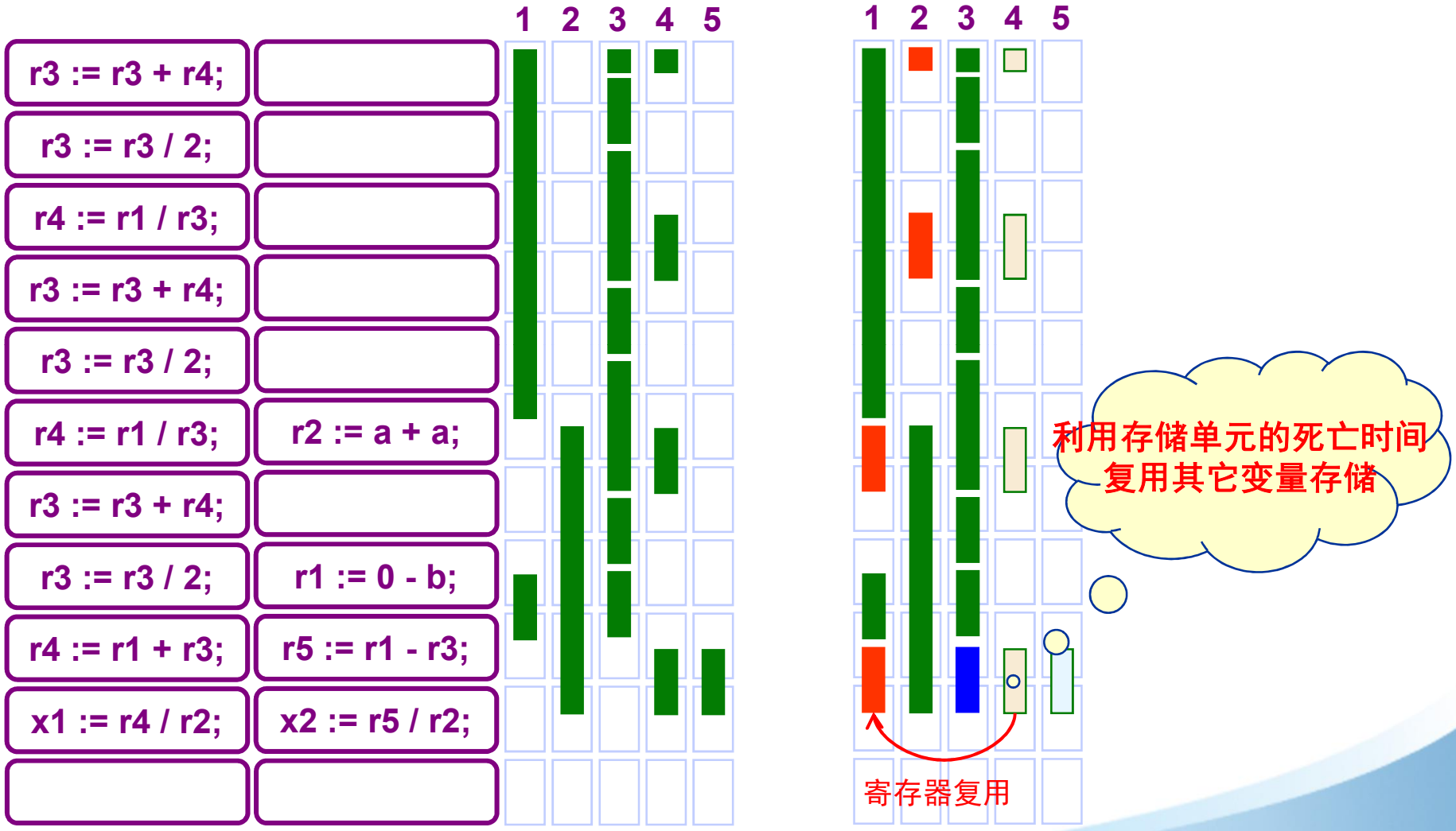
- 减小存储和资源开销
- 降低连线网络的复杂性
- 减少控制向量的数量
- 提高控制器的优化程度

存储单元分配

- **存活时间**：一个新值被写入时算起到该值被最后一次使用的时间
- **死亡时间**：所存储的数据被最后一次使用时算起到一个新值被写入时止

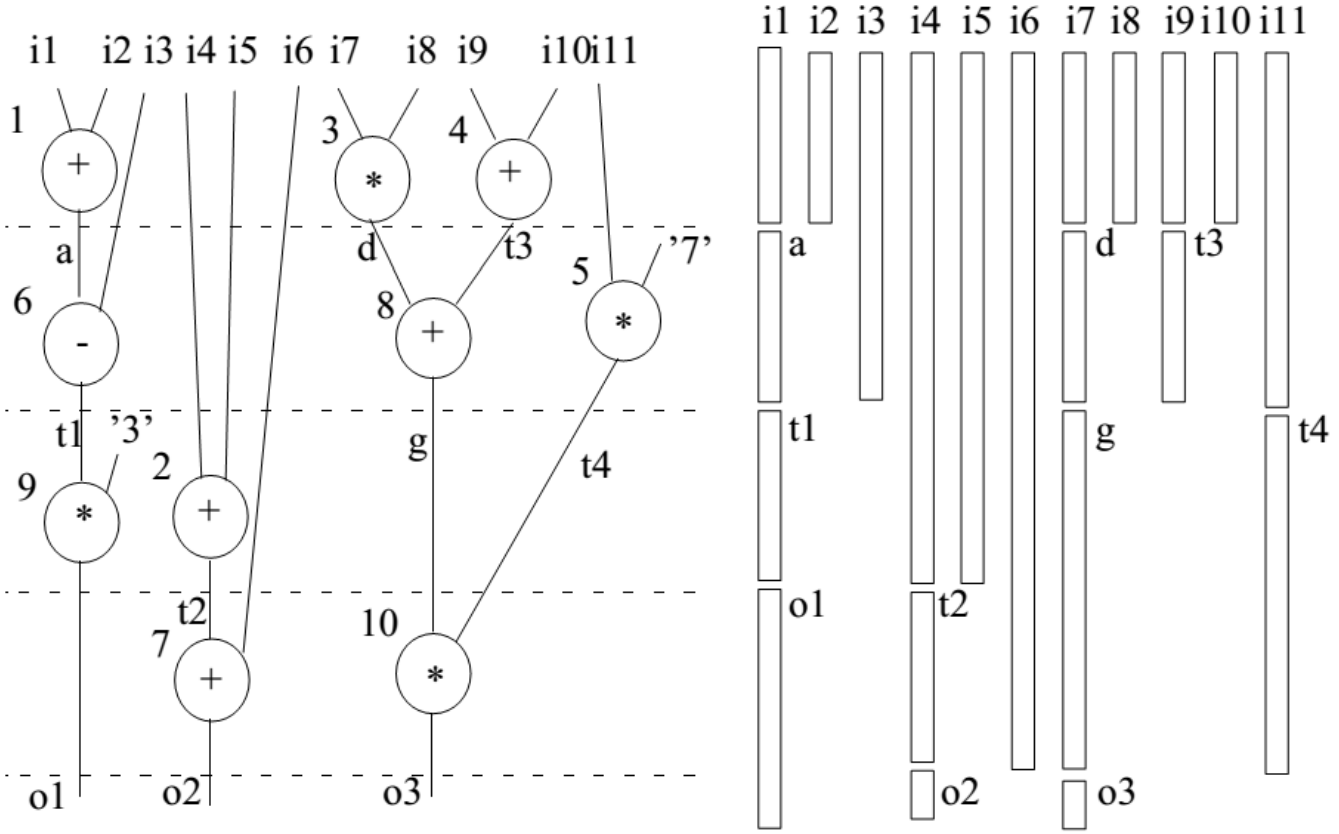


存储单元分配和复用


1 2 3 4 5
r3 := r3 + r4;
r3 := r3 / 2;
r4 := r1 / r3;
r3 := r3 + r4;
r3 := r3 / 2;
r4 := r1 / r3;
r2 := a + a;
r3 := r3 + r4;
r3 := r3 / 2;
r1 := 0 - b;
r4 := r1 + r3;
r5 := r1 - r3;
x1 := r4 / r2;
x2 := r5 / r2;
1 2 3 4 5
利用存储单元的死亡时间
复用其它变量存储
寄存器复用

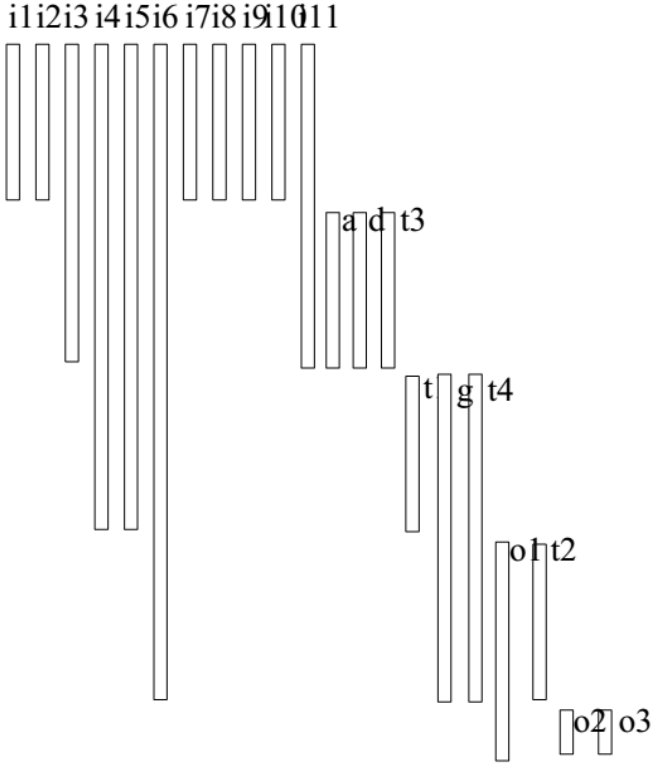
Left Edge(LE) Algorithm

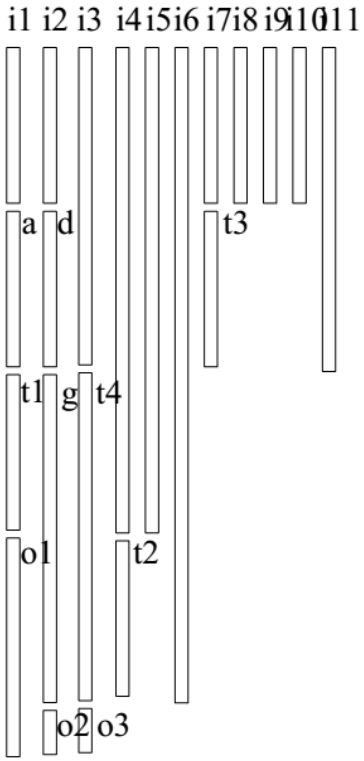
- 第一步：根据寄存器存活时间画出直方图



Left Edge(LE) Algorithm

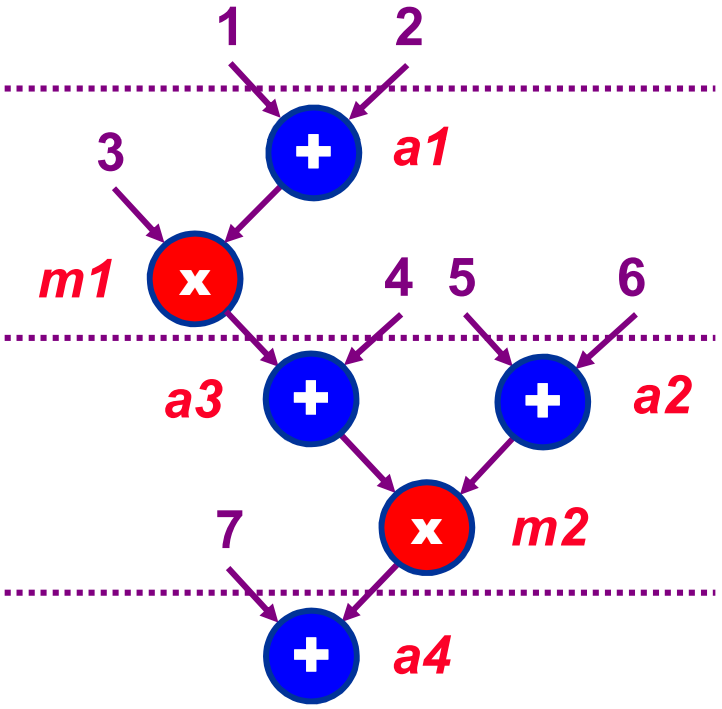
- 第二步：按照时钟节拍根据存活时间排序
- 第三步：按照时钟节拍将可以复用的寄存器值插空到已有寄存器的死亡时间空隙中



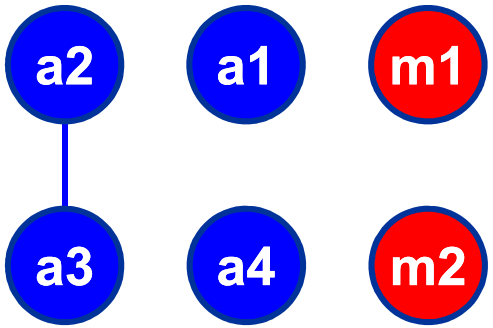


运算资源分配和复用

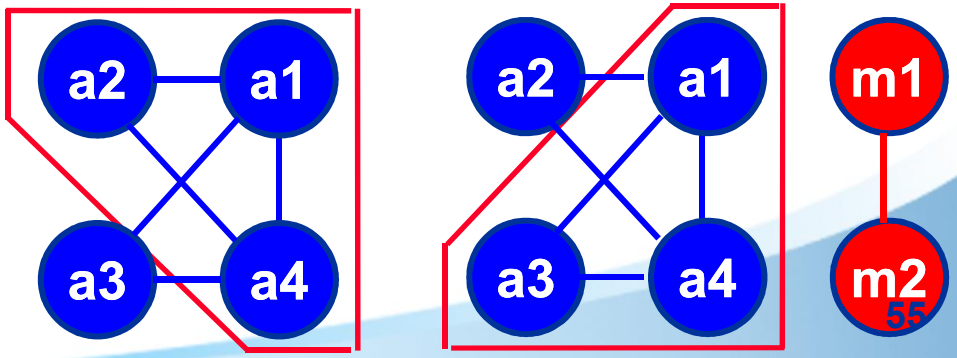
- 在算子调度方案确定后，根据不同算子所处的位置进行资源的分配和复用
  - 相容图不是唯一的



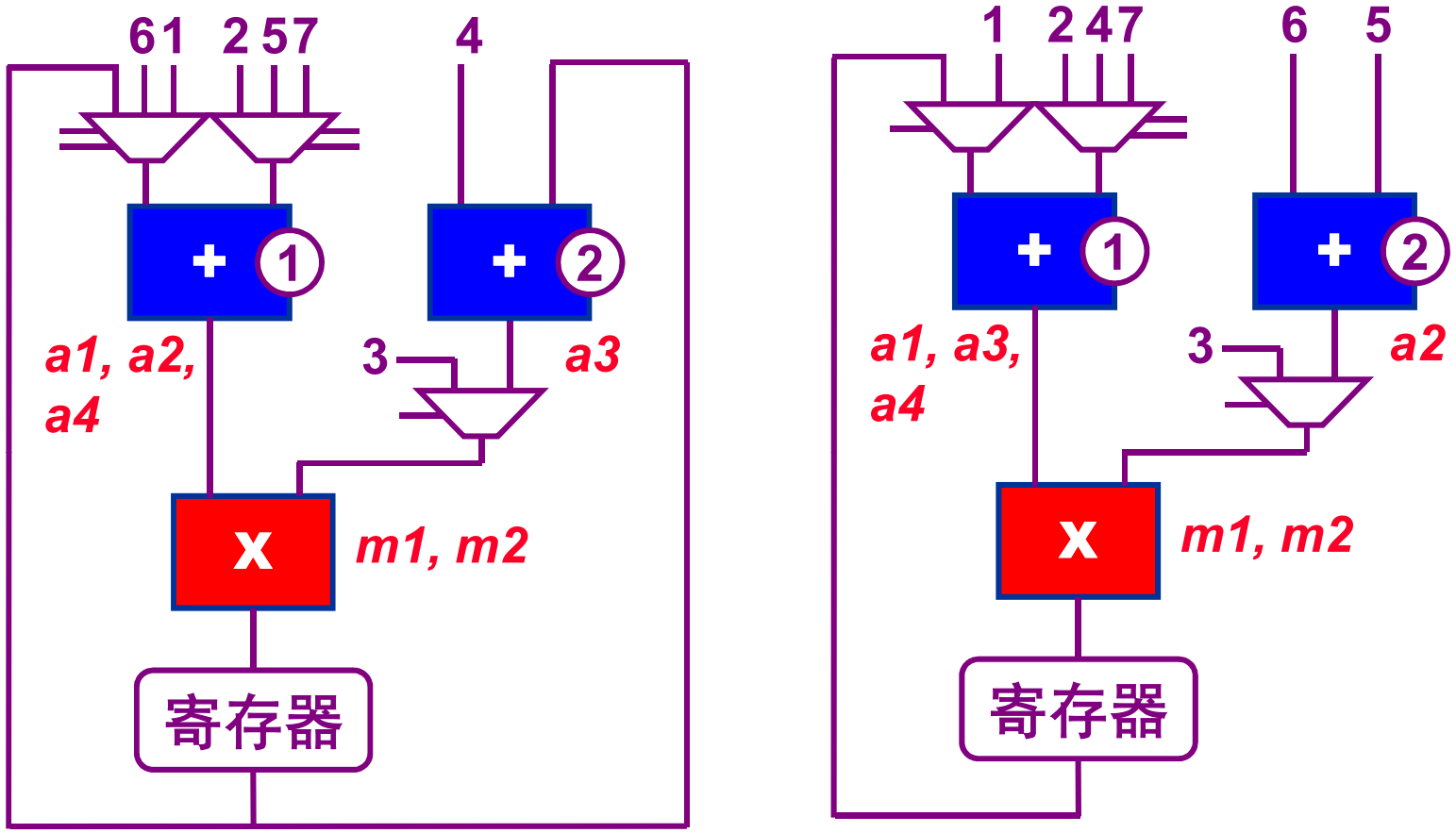
冲突图



相容图



运算资源分配和复用



以多路选择器的复杂度(输出/输入比，控制端数量)来衡量互连的复杂性。

运算资源分配和复用

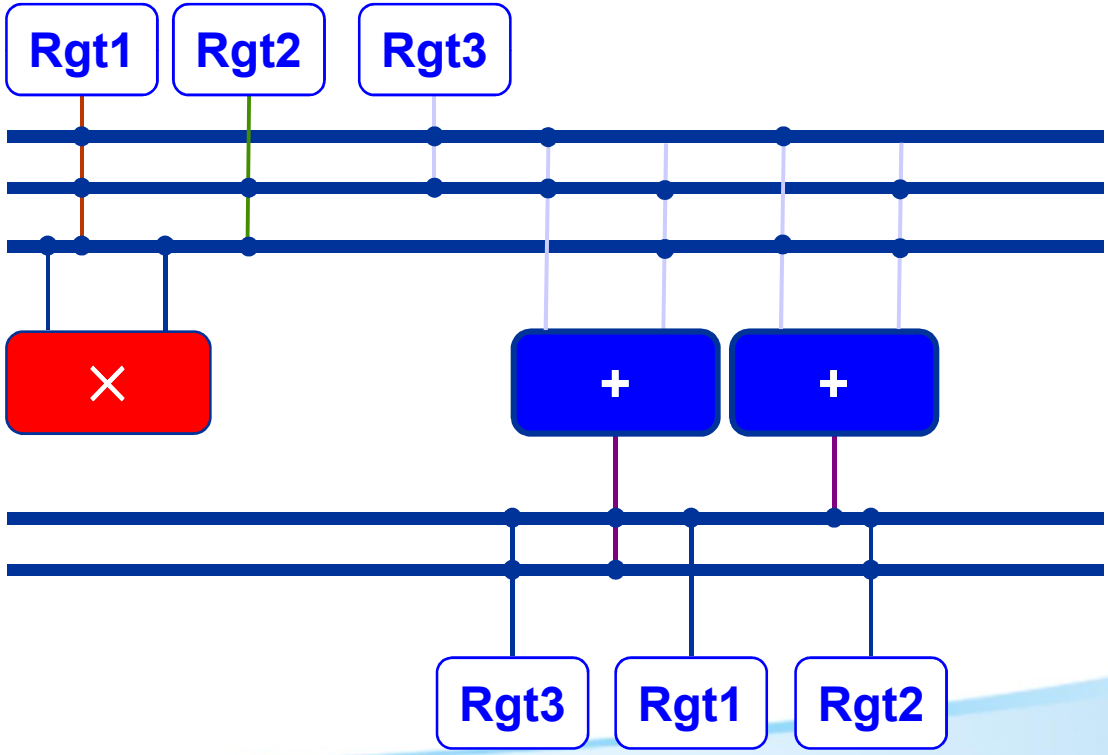
- 将资源分配问题变成寻找最大“全通图”的问题
- 相容图是冲突图的对偶图
- 优化程度取决于对图的划分
- 寻找全通图是件困难的事情

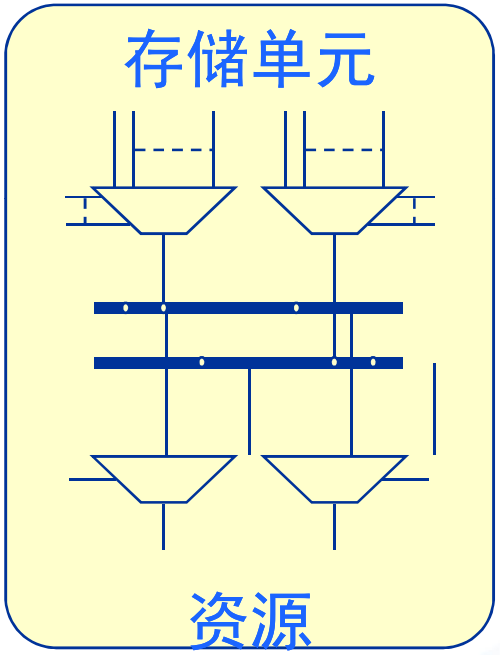
目录

- 高层次综合基本概念
- 数据通道设计
  - 算子调度 (scheduling)
  - 资源分配 (allocation)
  - 控制码及连线网络生成
- 控制通道设计
- Loop的处理
- 性能评估

连线网络生成

- 连线网络提供存储单元与资源之间的信号通路
  - 通常为总线形式
  - 输入总线端多路选择器网络
  - 输出总线端多路选择器网络





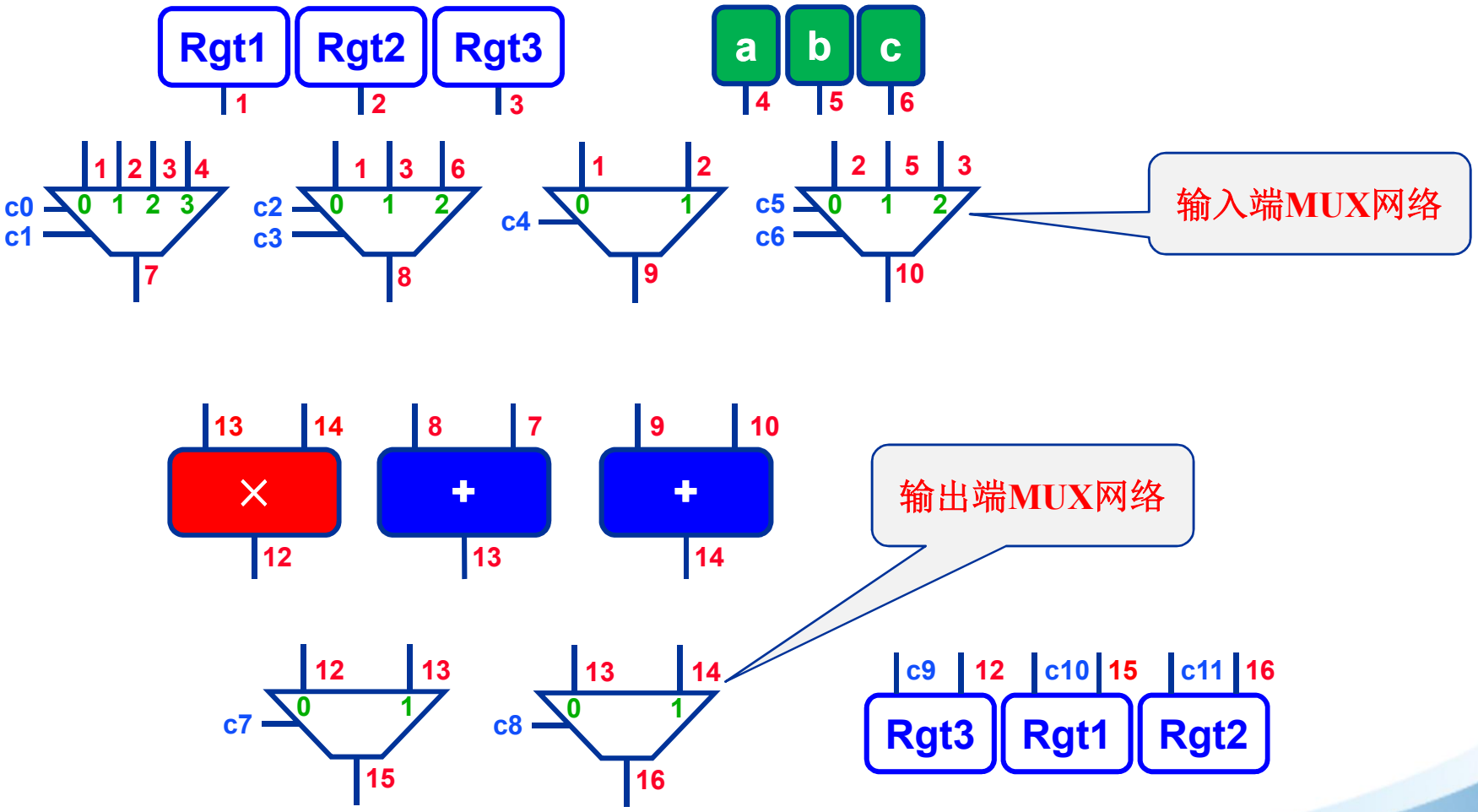
控制码生成

- 控制码向数据通道提供控制信号
- 控制码控制数据通道的运行
- 控制码决定控制单元的代价
- 控制码由数据通道的结构决定
  - 连线网络的控制码
  - 资源的控制码
  - 存储单元的控制码

控制码优化

- 控制码组的优化包括控制码数量的减少和控制码码长的减少；
- 如果NCS表示控制步的步数，NCD表示控制码数量，则有NCD ≤ NCS。或者说NCS给出了控制码数量的严格上界；
- 如果LCD表示控制码的码长，则减少LCD成为控制码优化的重要内容。

控制码生成和优化



控制码CS= {c11,c10, c9,…c1, c0}

控制码生成和优化

	c11	c10	c9	c8	c7	c6	c5	c4	c3	c2	c1	c0
Cs0	0	0	0	x	x	1	0	0	0	1	0	0
Cs1	1	1	1	1	1	0	0	0	0	0	0	0
Cs2	0	0	1	0	0	1	1	1	1	1	1	1
Cs3	0	1	0	1	0	0	1	1	1	0	0	1
Cs4	0	0	0	x	x	1	1	1	1	1	1	1
Cs5	1	0	1	0	0	0	0	0	0	0	0	0
Cs6	0	1	1	0	1	0	0	0	1	0	1	1

控制码不经过任何优化，则所需存储空间7*12bit

控制码生成和优化

	c11	c10	c9	c8	c7	c6	c5	c4	c3	c2	c1	c0
Cs0	0	0	0	x	x	1	0	0	0	1	0	0
Cs1	1	1	1	1	1	0	0	0	0	0	0	0
Cs2	0	0	1	0	0	1	1	1	1	1	1	1
Cs3	0	1	0	1	0	0	1	1	1	0	0	1
Cs4	0	0	0	x	x	1	1	1	1	1	1	1
Cs5	1	0	1	0	0	0	0	0	0	0	0	0
Cs6	0	1	1	0	1	0	0	0	1	0	1	1

c5=c4, c3=c0,

控制码生成和优化

	c11	c10	c9	c8	c7	c6	c5	c4	c3	c2	c1	c0
Cs0	0	0	0	x	x	1	0	0	0	1	0	0
Cs1	1	1	1	1	1	0	0	0	0	0	0	0
Cs2	0	0	1	0	0	1	1	1	1	1	1	1
Cs3	0	1	0	1	0	0	1	1	1	0	0	1
Cs4	0	0	0	x	x	1	1	1	1	1	1	1
Cs5	1	0	1	0	0	0	0	0	0	0	0	0
Cs6	0	1	1	0	1	0	0	0	1	0	1	1

c5=c4, c3=c0,

Cs4=~Cs1,

控制码组的优化

- 如果两组控制码相同则它们可以合并
- 如果两组控制码之间存在某种逻辑关系则它们可以合并

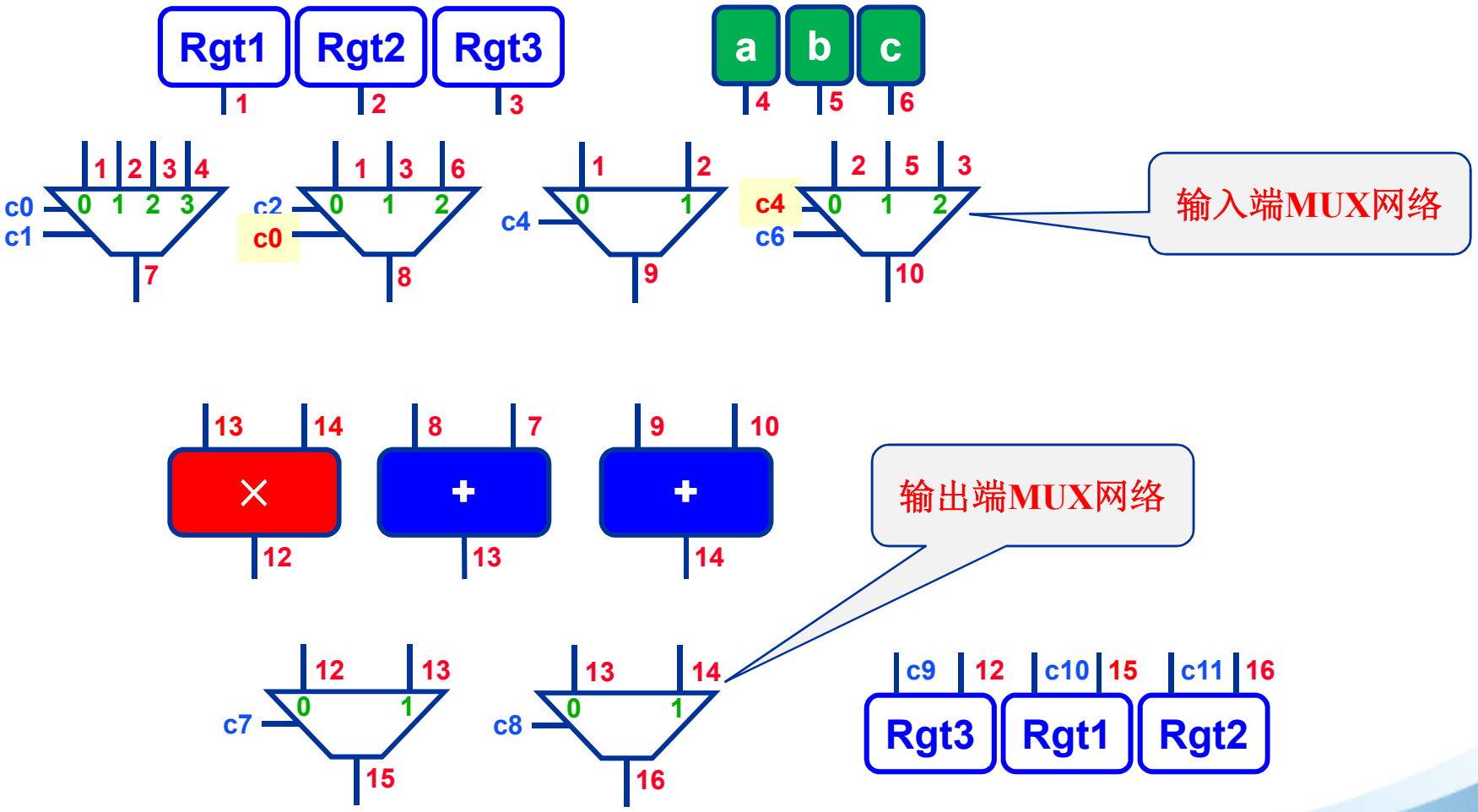
控制码生成和优化

	c11	c10	c9	c8	c7	c6	c5	c4	c3	c2	c1	c0
Cs0	0	0	0	x	x	0	0	0	0	0	0	0
Cs1	0	0	0	1	1	0	0	0	0	0	0	0
Cs2	1	1	1	0	0	1	1	1	1	1	1	1
Cs3	0	1	0	1	0	0	1	1	1	0	0	1
Cs4	0	0	0	x	x	1	1	1	1	1	1	1
Cs5	1	0	1	0	0	0	0	0	0	0	0	0
Cs6	0	1	1	0	1	1	0	0	1	0	1	1

优化后，所需存储空间为6*10bit

控制码数量的优化只会减少控制码的存储量，
但是在系统运行过程中所需的控制码数量不会变

控制码生成和优化



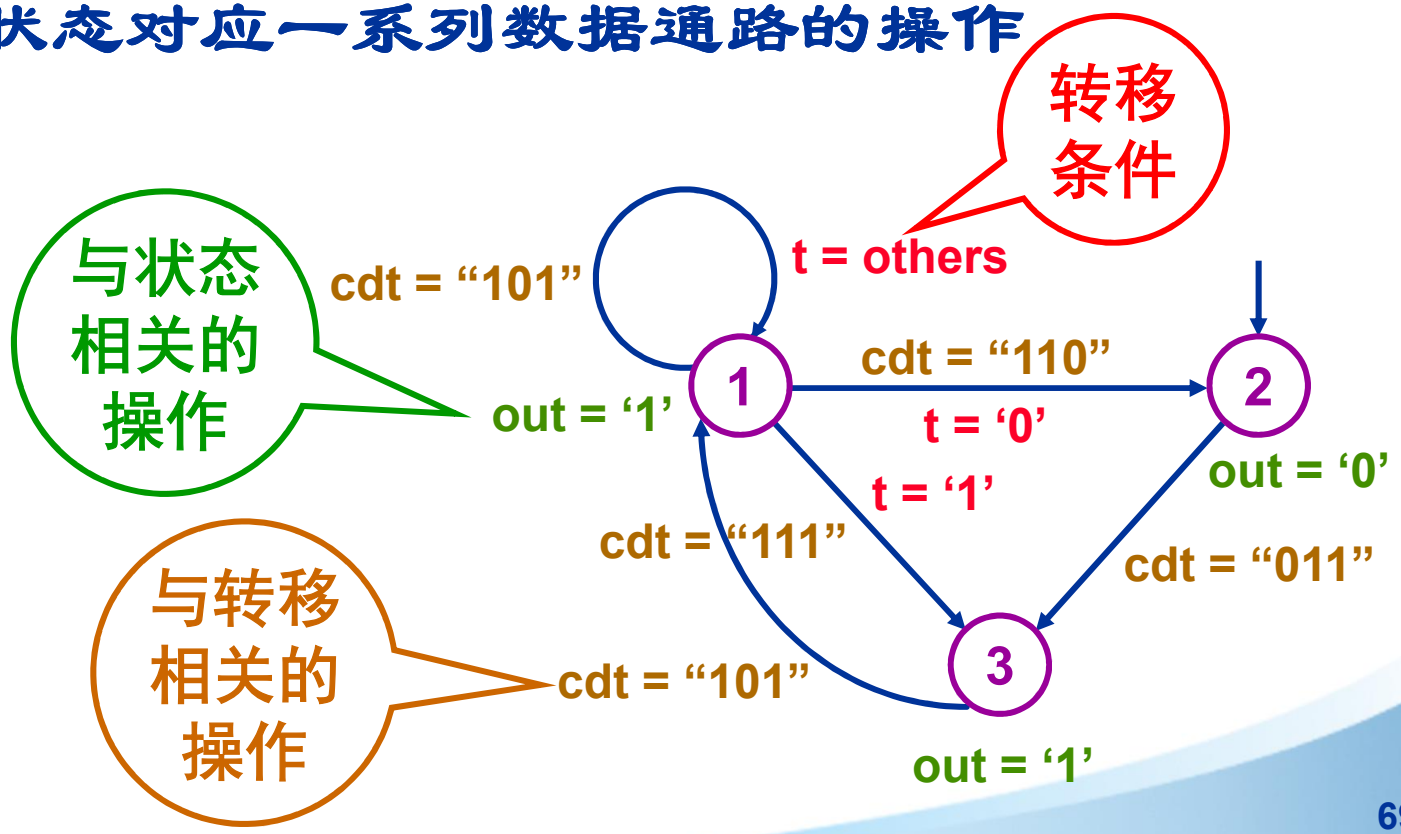
控制码CS= {c11,c10, c9,…c1, c0}

目录

- 高层次综合基本概念
- 数据通道设计
  - 算子调度 (scheduling)
  - 资源分配 (allocation)
  - 控制码及连线网络生成
- 控制单元设计
- Loop的处理
- 性能评估

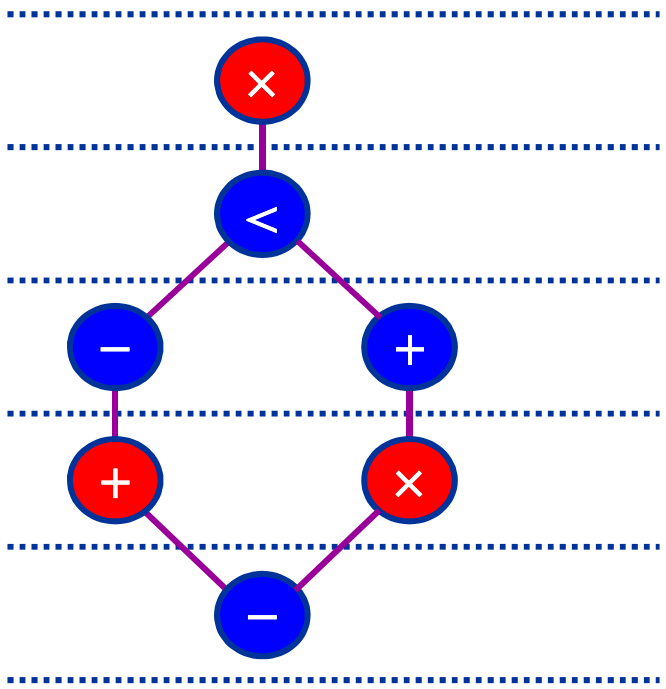
控制单元设计

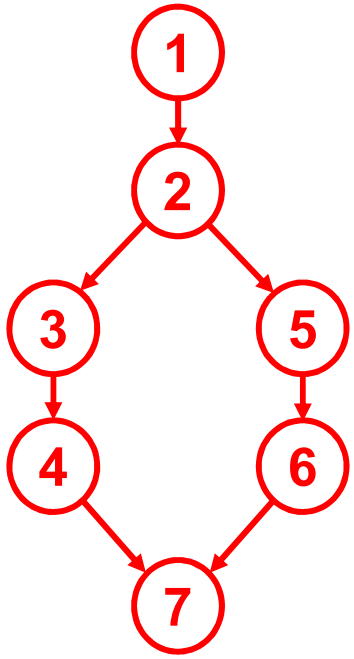
- 控制单元向数据通道提供所需的控制码
- 控制单元是有限状态机器(FSM)
- 每一个状态对应一系列数据通路的操作



控制单元设计

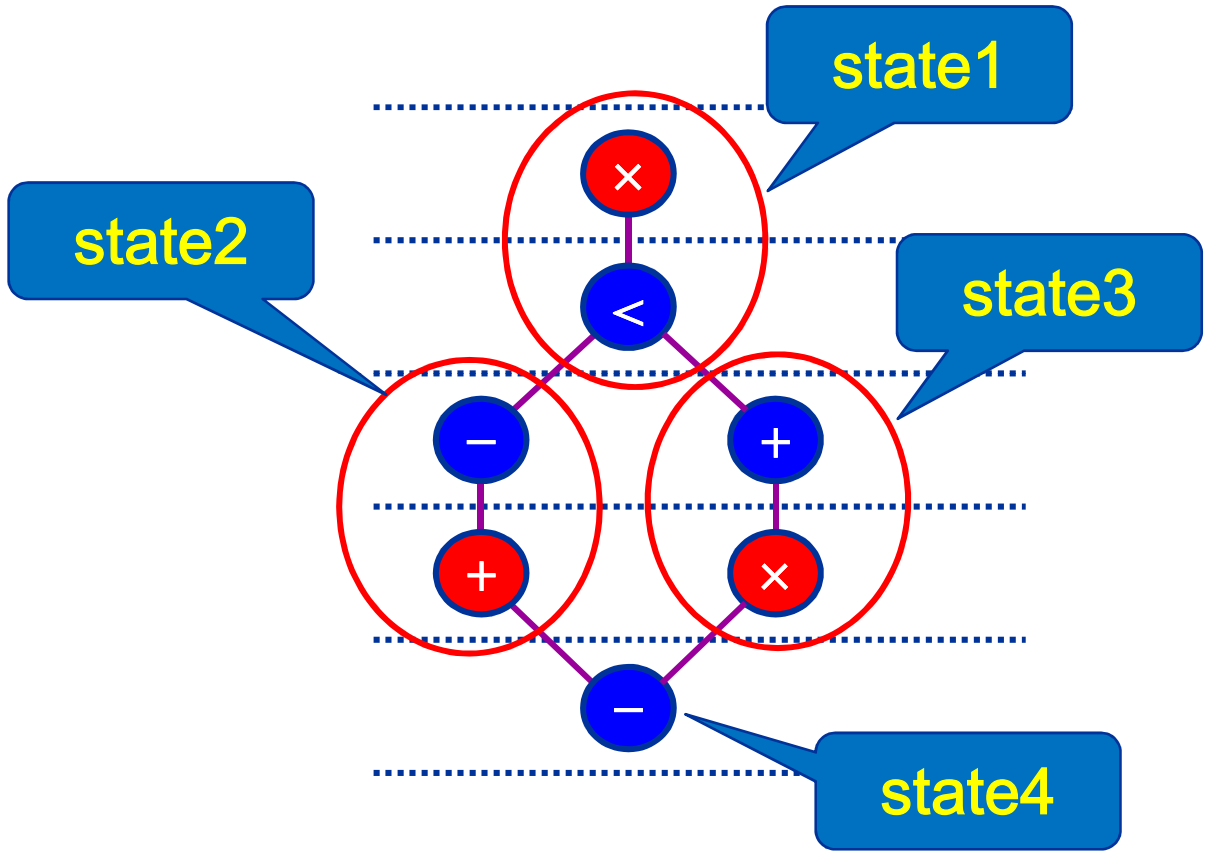
例一：状态=控制步





- 分支节点：单个输入，多个输出
- 汇合节点：多个输入，单个输出
- 起始节点：仅具有输出的节点是
- 结束节点：仅具有输入的节点是

例二：状态=支路



在支路状态上，控制码的执行按照控制步给出的顺序进行
实质是将一级寻址转化为两级寻址

目录

- 高层次综合基本概念
- 数据通道设计
  - 算子调度 (scheduling)
  - 资源分配 (allocation)
  - 控制码及连线网络生成
- 控制通道设计
- Loop的处理
- 性能评估
- 高层次综合工具简介

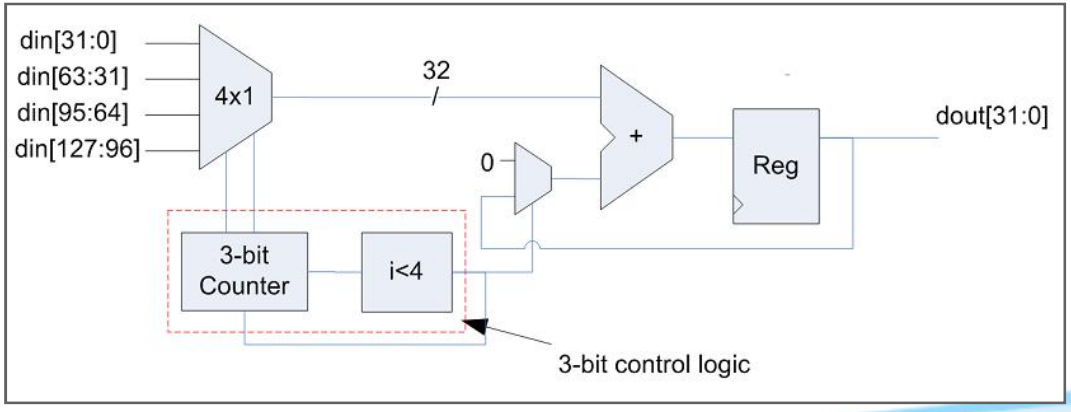
Loop

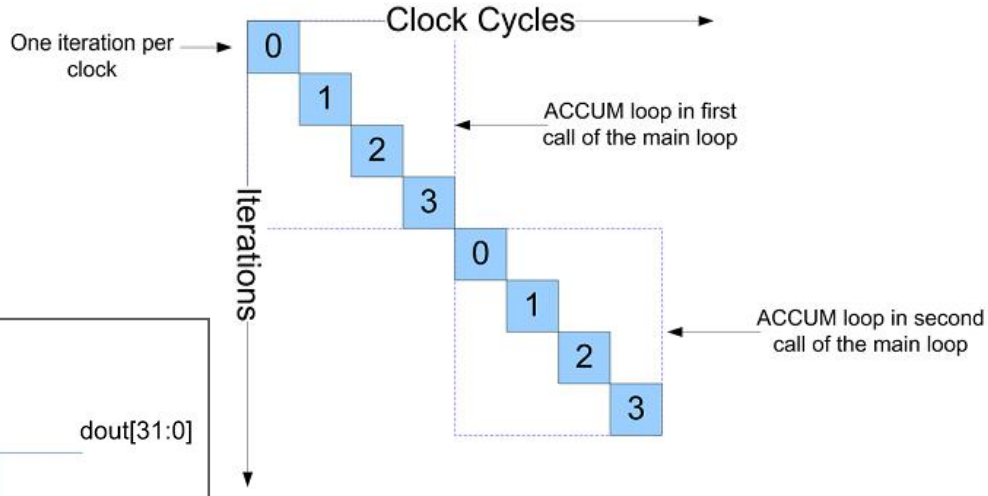
- Loop是程序中非常重要的一种描述方式，在高层次综合中对loop的合理处理直接影响映射出的硬件结构的性能
- Loop的常用概念
  - Loop iterations：loop循环的次数
  - Loop iterator：计算循环次数的变量
  - Loop unrolling：循环体被展开的次数
  - Loop pipelining：循环体的流水线级数
  - Initiation Interval (II)：定义多少个时钟周期开始一次新的循环

Loop

- 乘累加循环举例
  - Clock cycle：4
  - 1 adder

```
void accumulate4(int din[4], int &dout){
  int acc=0;
 ACCUM:for(int i=0;i<4;i++){
    acc += din[i];
  }
  dout = acc;
}
```

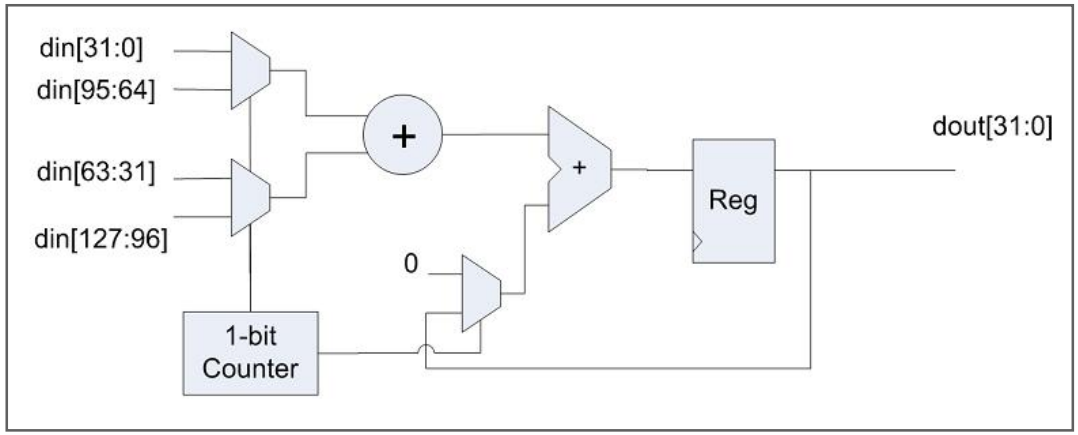


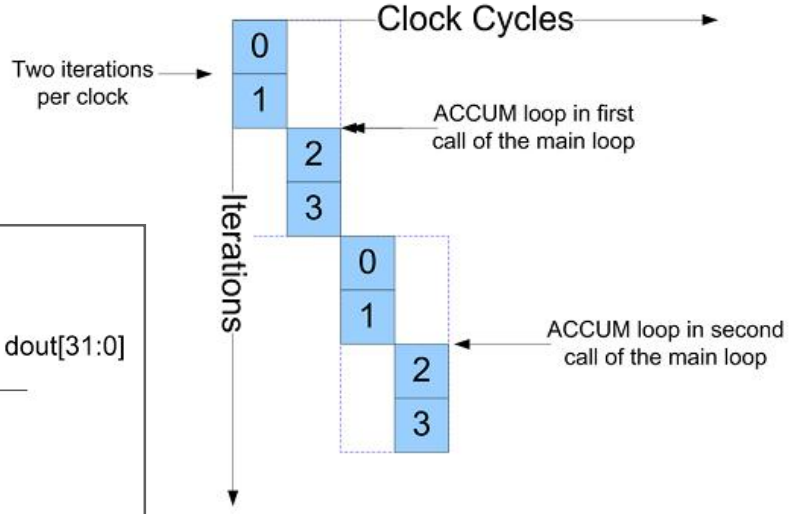


Loop

- **Partial loop unrolling**
  - **Clock cycle：2**
  - **2 adder**

```
void accumulate(int din[4], int &dout){
  int acc=0;
 ACCUM:for(int i=0;i<4;i+=2){
    acc += din[i];
    acc += din[i+1];
  }
  dout = acc;
}
```



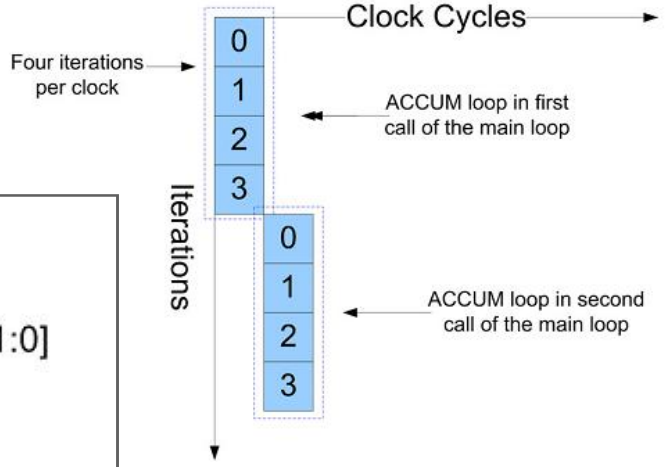


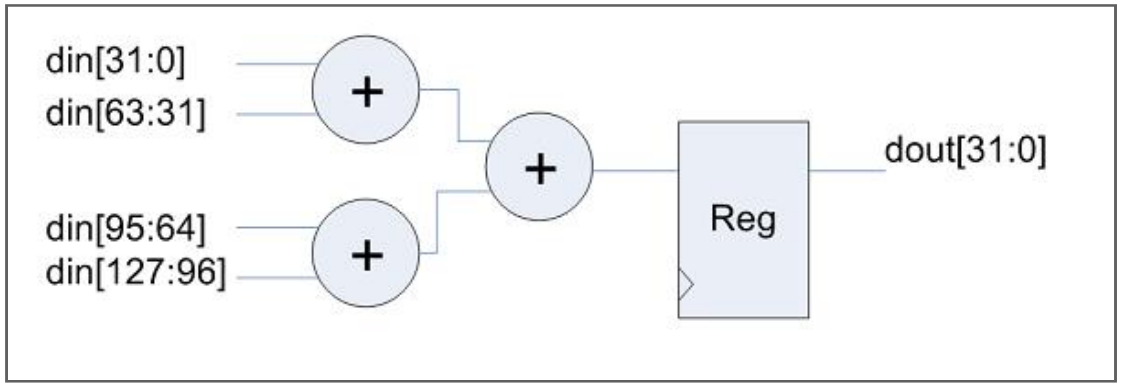
Loop

Fully Unrolled Loops

- Clock cycle：1
- 3 adder

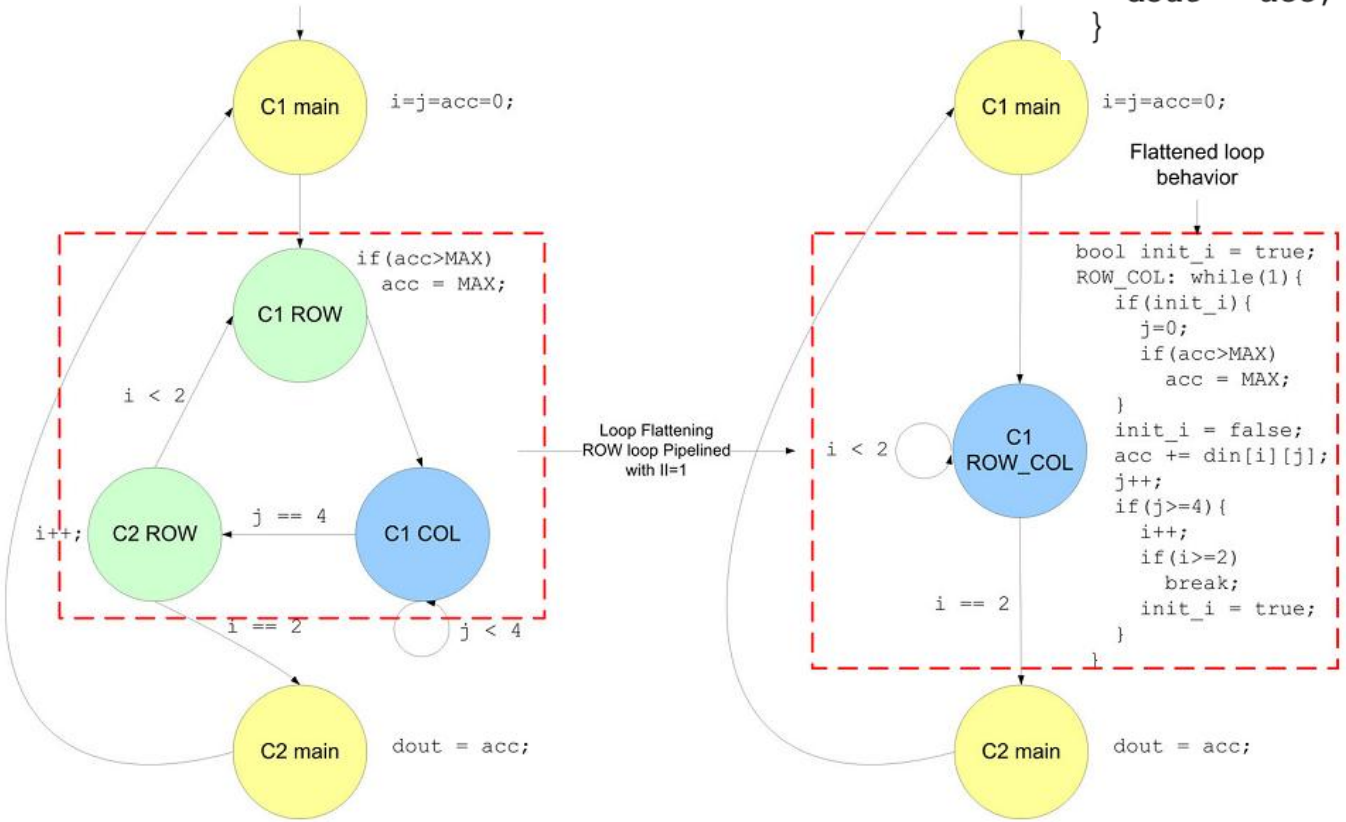
```
void accumulate(int din[4], int &dout){
    int acc=0;

    acc += din[0];
    acc += din[1];
    acc += din[2];
    acc += din[3];

    dout = acc;
}
```



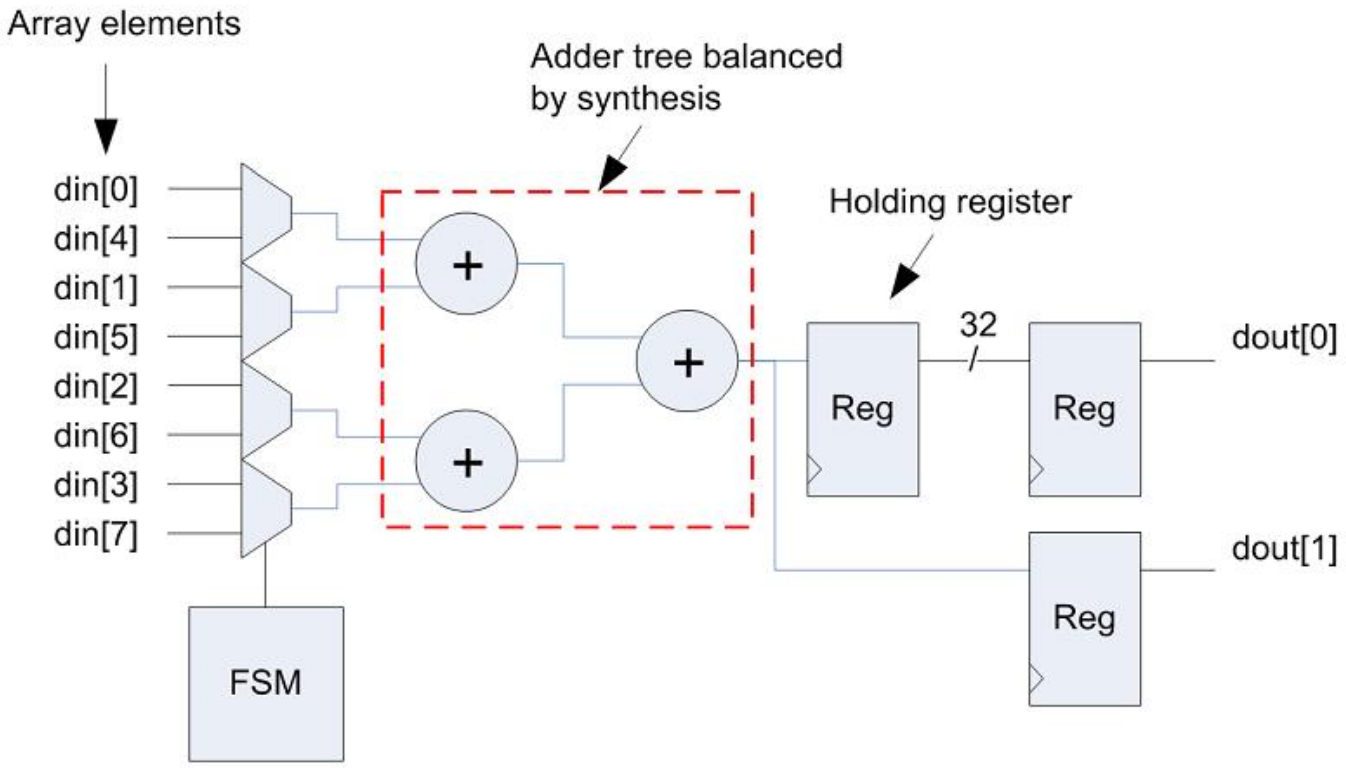


Nested loop unrolling

```
#include "accum.h"
#include <ac_int.h>
#define MAX 100000
void accumulate(int din[2][4], int &dout){
  int acc=0;
 ROW:for(int i=0;i<2;i++){
    if(acc>MAX)
      acc = MAX;
  COL:for(int j=0;j<4;j++){
      acc += din[i][j];
    }
  }
  dout = acc;
}
```

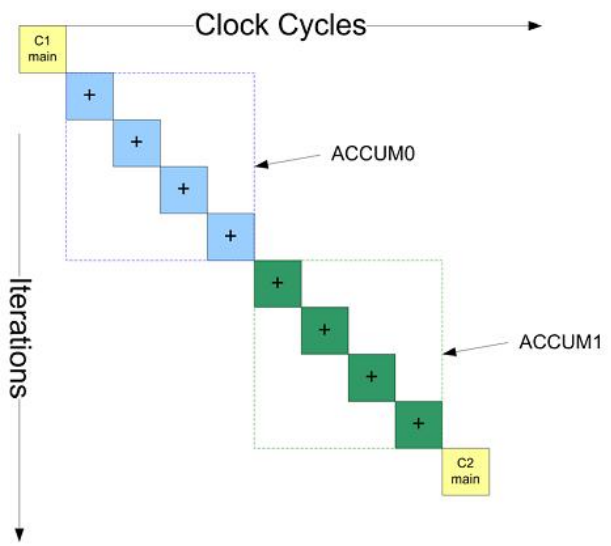


Nested loop unrolling

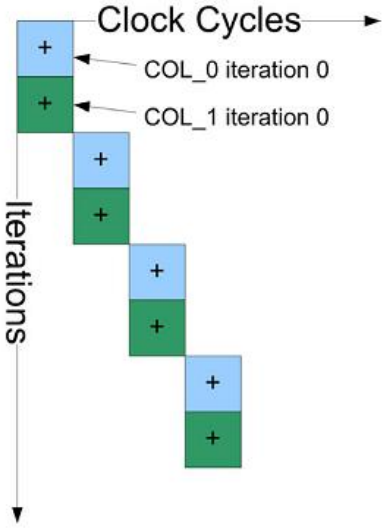


Loop merging

```
#include "accum.h"
void accumulate(int din[2][4], int dout[2]){
  int acc[2];

  acc[0] = 0;
 COL_0:for(int j=0;j<4;j++){
    acc[0] += din[0][j];
  }
  dout[0] = acc[0];
  acc[1] = 0;
 COL_1:for(int j=0;j<4;j++){
    acc[1] += din[1][j];
  }
  dout[1] = acc[1];
}
```

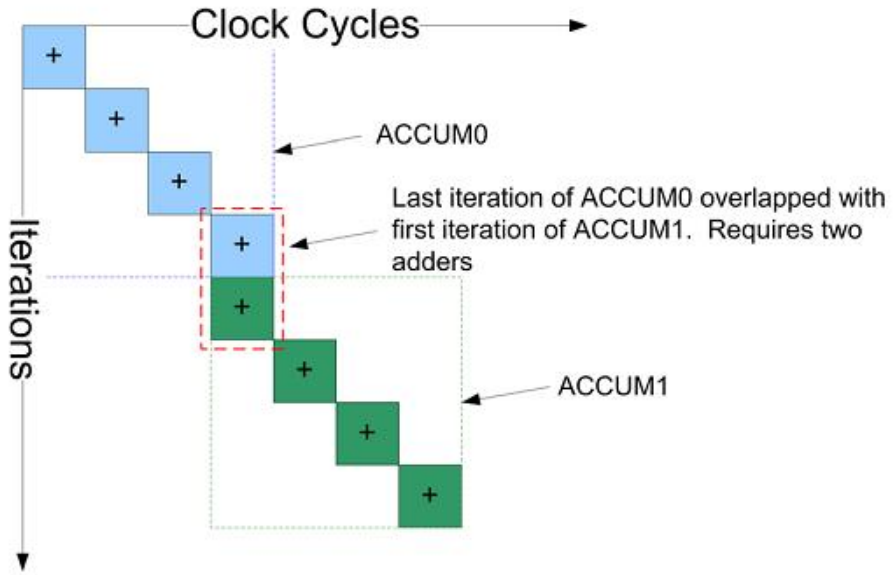



```
#include "accum.h"
void accumulate(int din[2][4], int dout[2]){
  int acc[2];

  acc[0] = 0;
  acc[1] = 0;
 COL_0_1:for(int j=0;j<4;j++){
    acc[0] += din[0][j];
    acc[1] += din[1][j];
  }
  dout[0] = acc[0];
  dout[1] = acc[1];
}
```



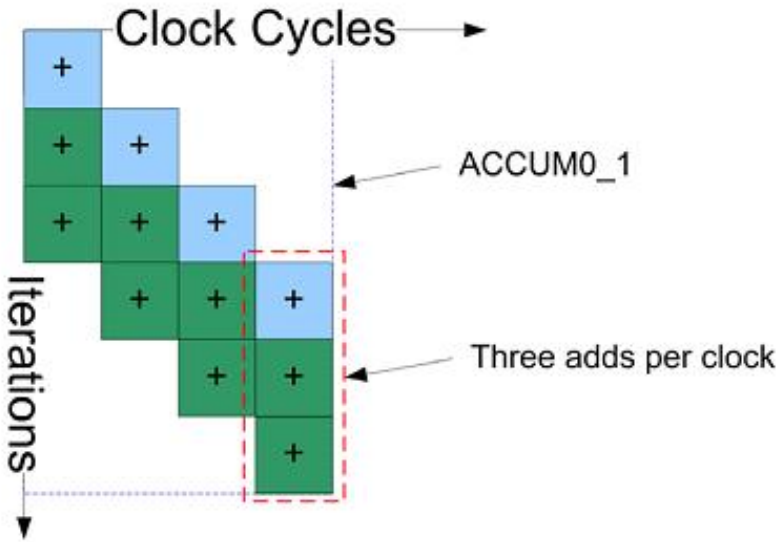
Sequential Loops

```
#include "accum.h"
void accumulate(int din0[4], int din1[4],int &dout0, int &dout1){
  int acc0=0;
  int acc1=0;

 ACCUM0:for(int i=0;i<4;i++){
    acc0 += din0[i];
  }
  acc1 = acc0;
 ACCUM1:for(int i=0;i<4;i++){
    acc1 += din1[i];
  }
  dout0 = acc0;
  dout1 = acc1;
}
```



Sequential Loops

```
#include "accum.h"
void accumulate(int din0[4], int din1[4],int &dout0, int &dout1){
  int acc0=0;
  int acc1=0;
  int tmp;
 ACCUM0_1:for(int i=0;i<4;i++){
    tmp = din0[i];
    acc0 += tmp;
    acc1 += din1[i]+tmp;
  }
  dout0 = acc0;
  dout1 = acc1;
}
```



目录

- 高层次综合基本概念
- 数据通道设计
  - 算子调度 (scheduling)
  - 资源分配 (allocation)
  - 控制码及连线网络生成
- 控制通道设计
- Loop的处理
- 性能评估

性能评估

时间特性：

- 控制步步数 NCS
- 控制步步长 LCS
- 系统全延迟 DT = NCS * LCS
- 系统最高工作频率 fmax = 1/DT
- 系统最高时钟频率 fclk = 1/LCS

资源:

- 资源种类：Trsc
- 资源数量：Nrsc
- 资源代价：Crsc

$N_{rsc}(T_{rsc})$：单个控制步中类型为T_{rsc}的资源数量的最大值

性能评估

$N_{op}(i)$：控制步 i 中被执行的操作数量

连线网络的评估

连线网络：

$$N_{busIn} = 2 * \max_i \{N_{op}(i)\}$$
$$N_{busOut} = \max_i \{N_{op}(i)\}$$

- 寄存器数量：Nrgt
- 系统输入个数：Nin
- 所用的常数个数：Nconst
- 总线数量：NbusIn + NbusOut
- 多路器：
  - 输入总线多路数数量：Nmux(in)
  - 输出总线多路器数量：Nmux(out)

性能评估

控制器与控制码：

- 描述分支数量：Nb
- 控制器状态数：Nstate
- 控制码码宽：Wcc

$N_{state} = N_b$

输入总线输入端控制码：$W_{cc}(inin)$
输入总线输出端控制码：$W_{cc}(inout)$
输出总线输入端控制码：$W_{cc}(outin)$
输出总线输出端控制码：$W_{cc}(outout)$

Things to remember

- A good design methodology
  - Can keep up with changing specs
  - Permits architectural exploration
  - Facilitates verification and debugging
  - Eases changes for timing closure
  - Eases changes for physical design
  - Promotes reuse

中国科学院大学研究生课程

高等数字集成电路分析与设计

cache一致性

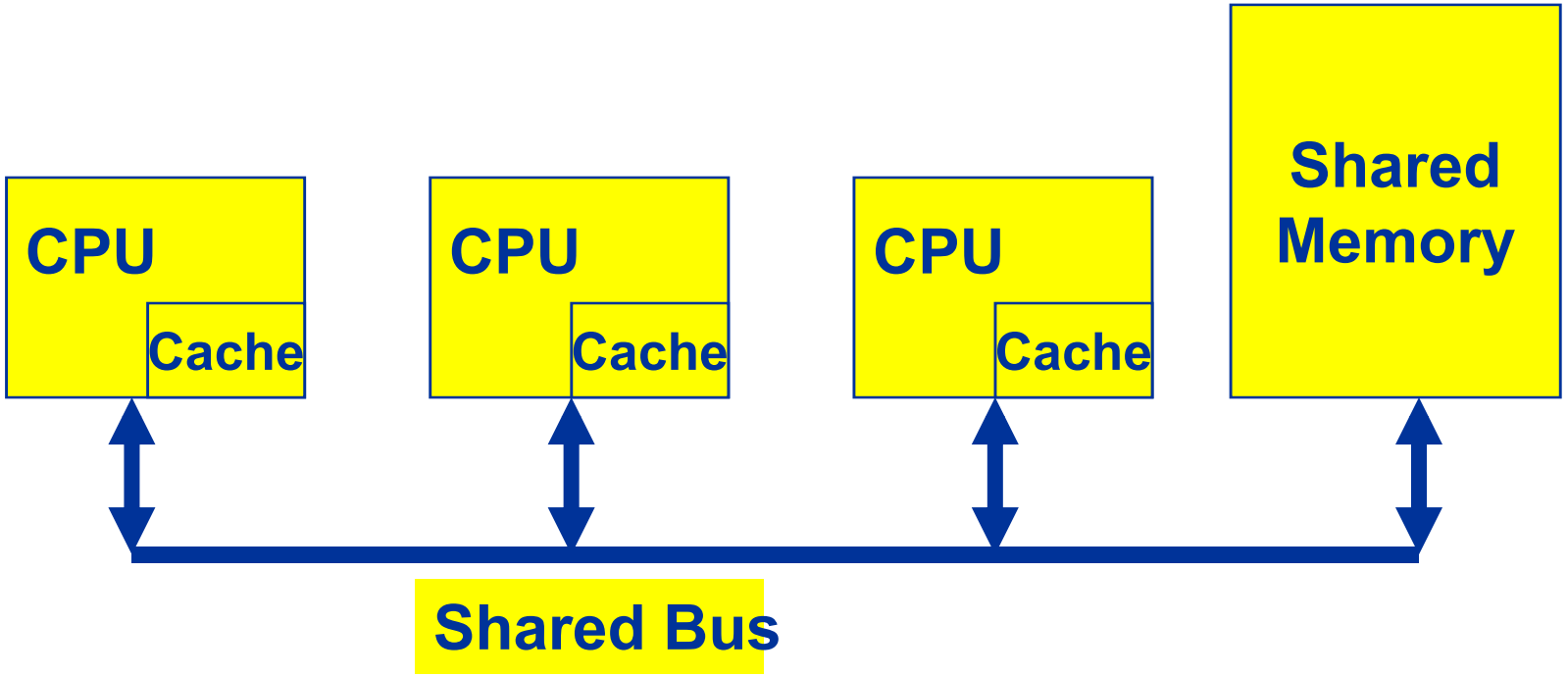
授课教师：王志君
电子邮件：wangzhijun@ime.ac.cn

中国科学院微电子研究所

概述

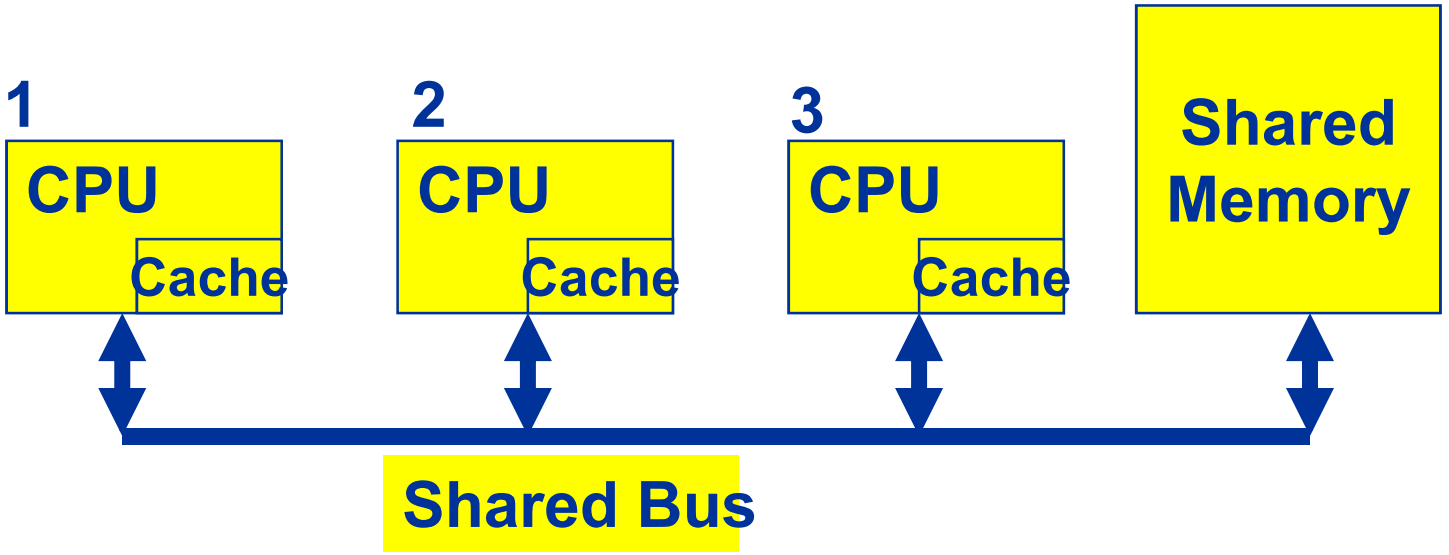
- 多核存储共享的两种基本架构:
  - 基于总线：Bus-based shared-memory (small-scale)
  - 基于目录：Directory-based shared-memory (large-scale)

Bus-based Shared Memory



- 总线带宽和速度限制了挂接CPU数量

数据一致性问题



CPU 1 读取 X地址: 读取数据20存放在本地cache中
CPU 2 读取 X地址:读取数据20存放在本地cache中
CPU 1 向X地址写数据 32: 本地cache数据被更改
CPU 3 读取 X地址: 它会读到什么数据?

Snooping

- 每当时有数据操作时，监测每个CPU的cache状态
- 每个CPU有写操作时
  - Write Invalidate
    - 一个CPU更新数据后，其他CPU将本地有相同项的数据直接invalid
  - Write Update
    - 一个CPU更新数据后，通过广播的方式让其他CPU更新本地有相同项的数据

Update or Invalidate?

- **Update**
  - 优点：简单直观
  - 缺点：占用带宽大，当有多次数据更新的时候会发起多次操作
- **Invalid**
  - 优点：占用带宽小
    - 多次写操作只需要发起一次invalid
  - 缺点：下次读操作需要重新fetch数据

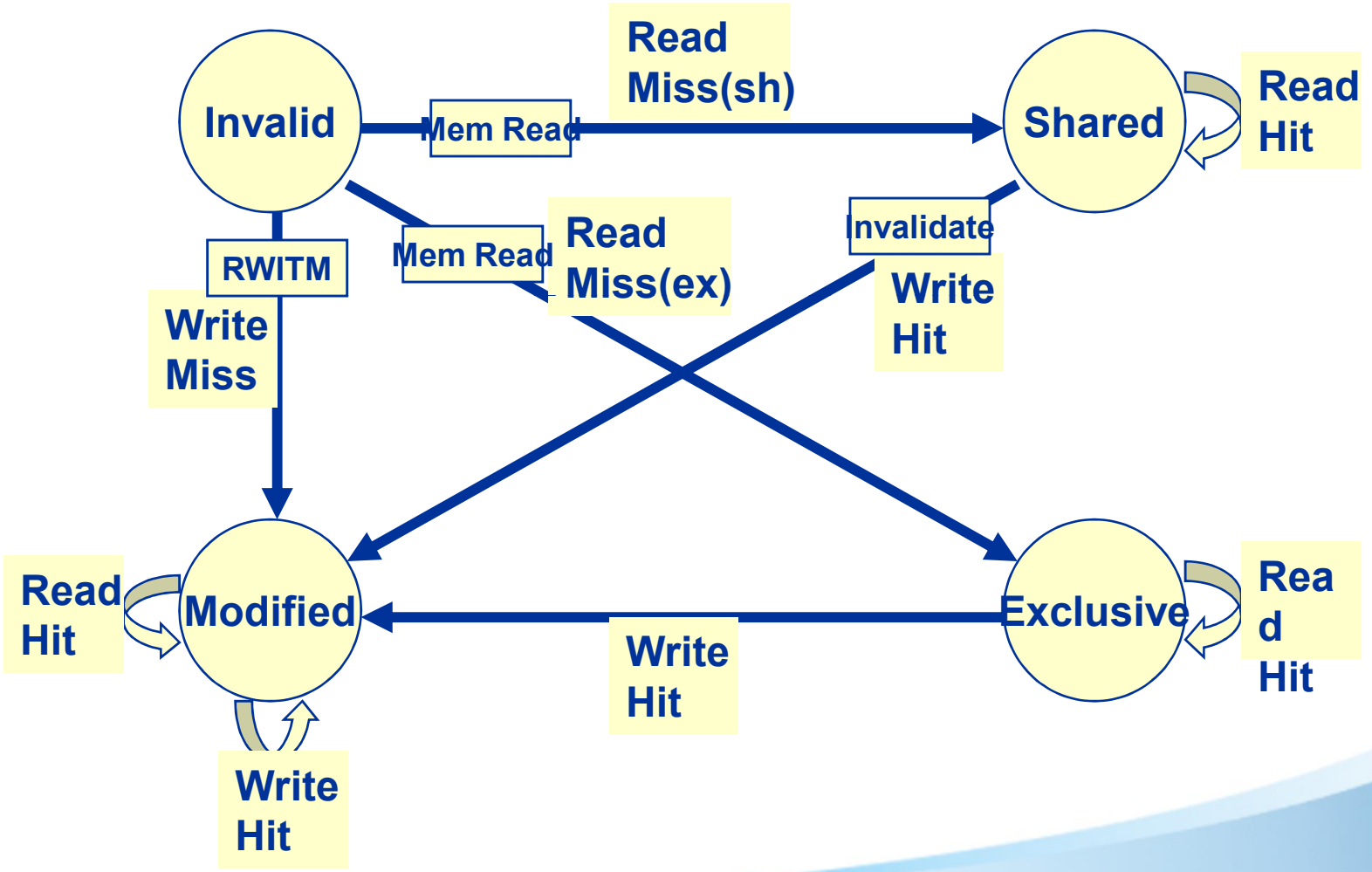
MESI Protocol

- MESI协议是一种基于write invalidate的一致性协议，旨在最小化总线带宽
- 支持 'write back' 写策略
- 每个cache line有4种状态
  - Modified - cache line has been modified, is different from main memory - is the only cached copy.
  - Exclusive - cache line is the same as main memory and is the only cached copy
  - Shared - Same as main memory but copies may exist in other caches.
  - Invalid - Line data is not valid

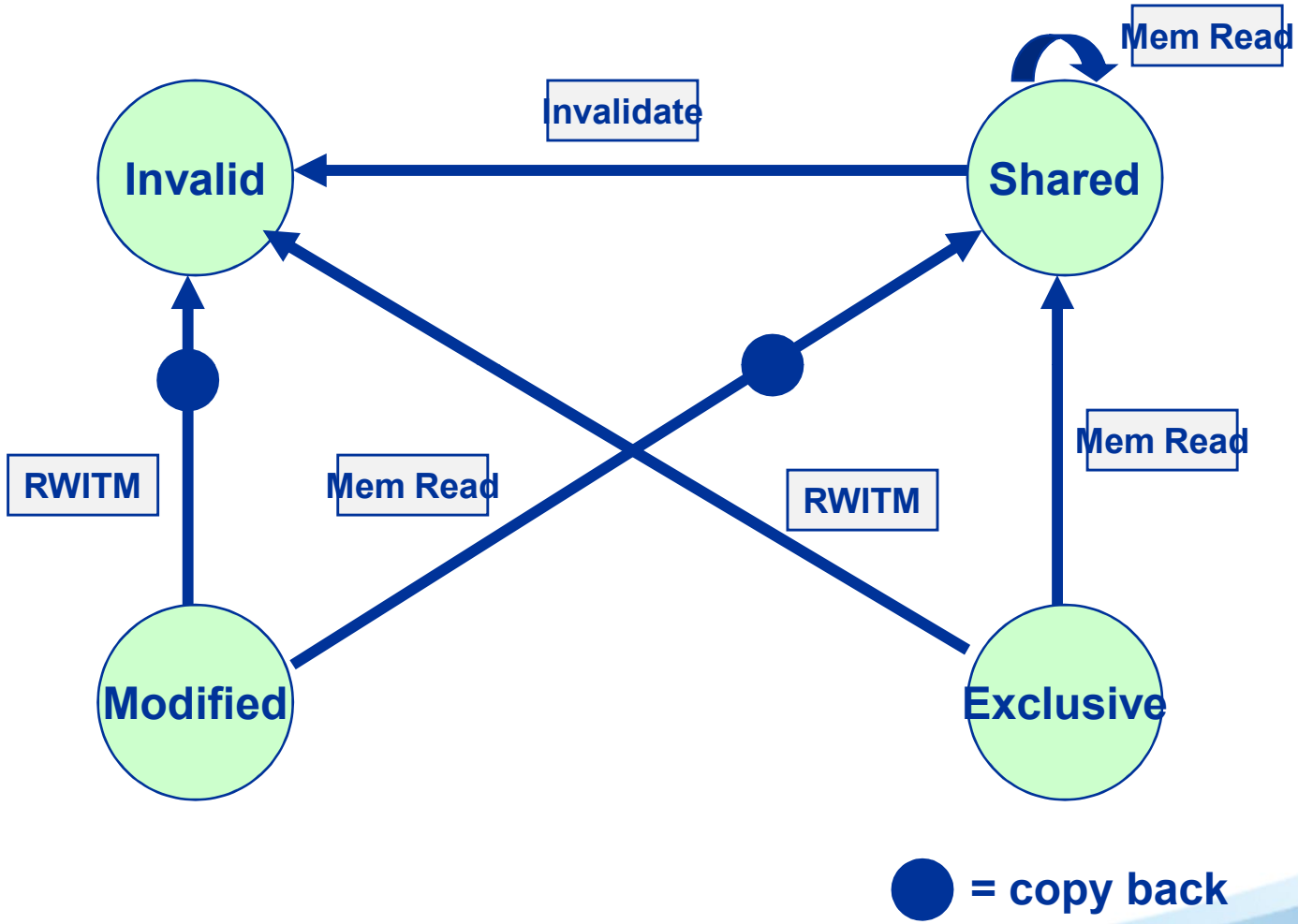
MESI Protocol

- Cache line 状态的更新发生在两种情况
  - 本地处理器自身的活动
  - 一致性总线上的活动（其他处理器引起）
- 本地处理器cache操作的分类
  - Read Hit
  - Read Miss
  - Write Hit
  - Write Miss

MESI – locally initiated accesses



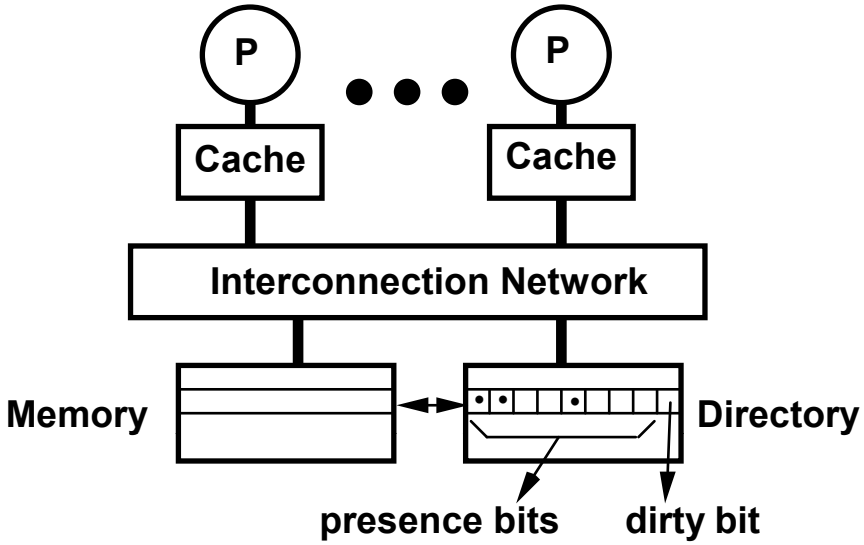
MESI – remotely initiated accesses



Directory Schemes

- Snoopy 扩展性差，只适合小规模使用（8核以内）
- Directory-based schemes allow scaling.
  - 通过一个目录记录memory block状态，然后通过点对点的方式维护存储一致性，避免了广播

Directory Basic Scheme



- 假设有 “k”个处理器
- 对每一个 cache-block: 目录中存放k比特当前状态位和 1个 dirty-bit
- cache中的每个cache-block： 1valid bit, 1 dirty bit

➢ **PE[i] Read :**

- 如果目录中 dirty-bit为0：直接读取主存数据; p[i]置1;
- 如果目录中dirty-bit 为1：令dirty PE 写回，同时cache state 置shared; 把目录中dirty-bit置0， p[i]置1; PE-I fetch数据 同时cache state shared置1; }

➢ **PE[i]Write:**

- 如果 dirty-bit 为0：把目录中记录有p[x]=1的处理器相应行invalid; 更新自己cache block并把目录中dirty-bit置1，P[i]置1;
- 如果 dirty-bit 为1：令dirty PE 写回后invalid，自己cache block 重新fetch数据后进行write操作，把目录中dirty-bit置1，P[i]置1;

谢谢!