



第3章 高层次综合

3.1 高层次综合概述

3.2 前端：基于C的编译优化

3.3 后端：高层次综合算法

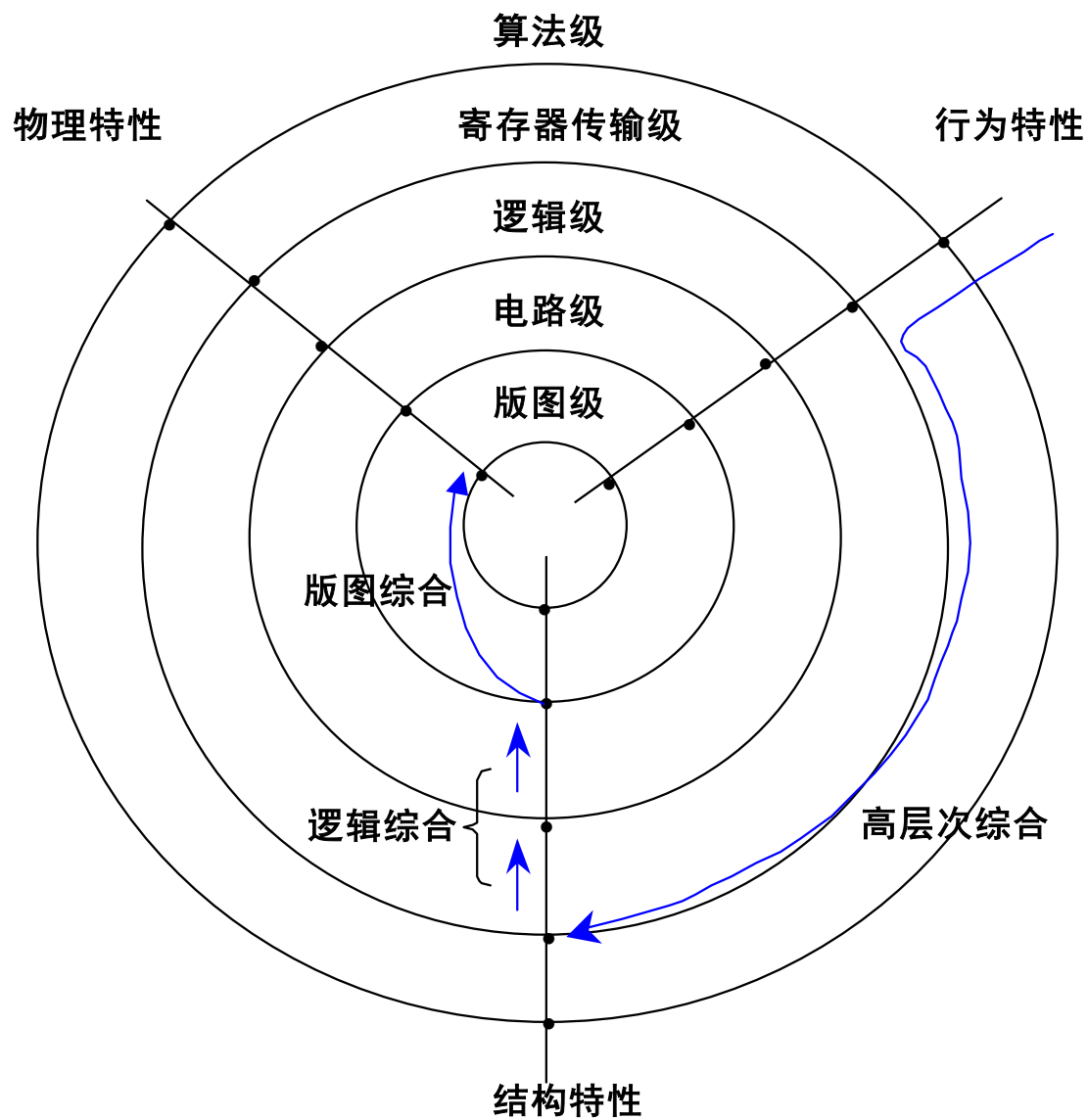
3.3.1 调度技术

3.3.2 分配技术

3.3.3 优化技术



高层次综合的层次关系





3.1 高层次综合概述

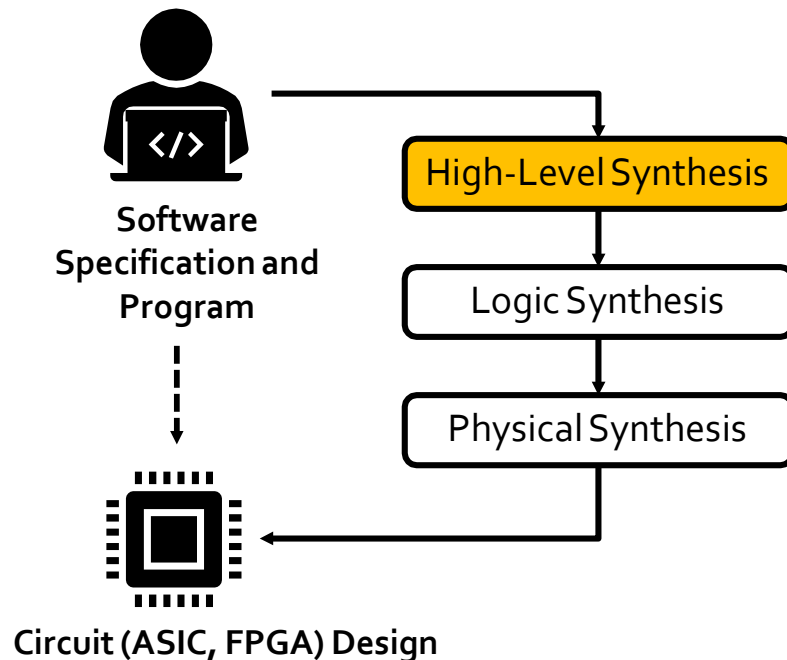
- ◆ 综合的层次概念：
 - 高层次综合（C/C++ \rightarrow RTL）；
 - 逻辑综合（RTL \rightarrow Netlist）；
 - 版图综合（Netlist \rightarrow GDS-II）。
- ◆ 高层次综合：
 - 算法级描述（行为信息）
 - \rightarrow RTL描述（行为信息 + 结构信息）



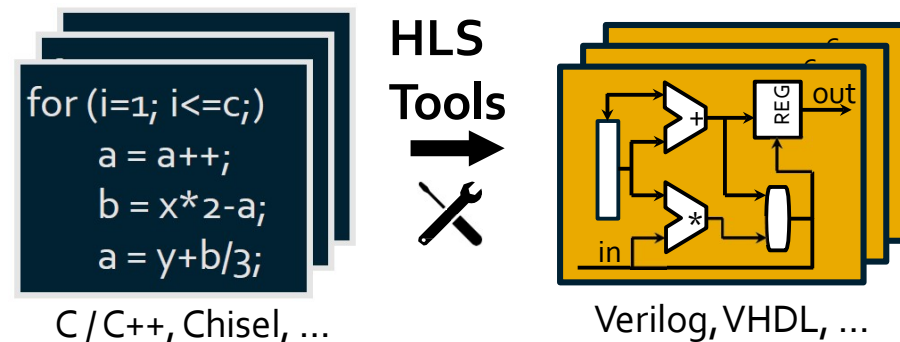
什么是HLS

软→硬

串→并



- An **automated** design process that transforms a **high-level functional specification** to optimized **register-transfer level (RTL)** descriptions for efficient hardware implementation





什么是HLS

```
for(int h = 0; h < H; h++ )
  for(int w = 0; w < W; w++)
    for(int m = 0; m < K; m++)
      for(int n = 0; n < K; n++)
        ...
```

行为级描述

人能看懂的、传神的

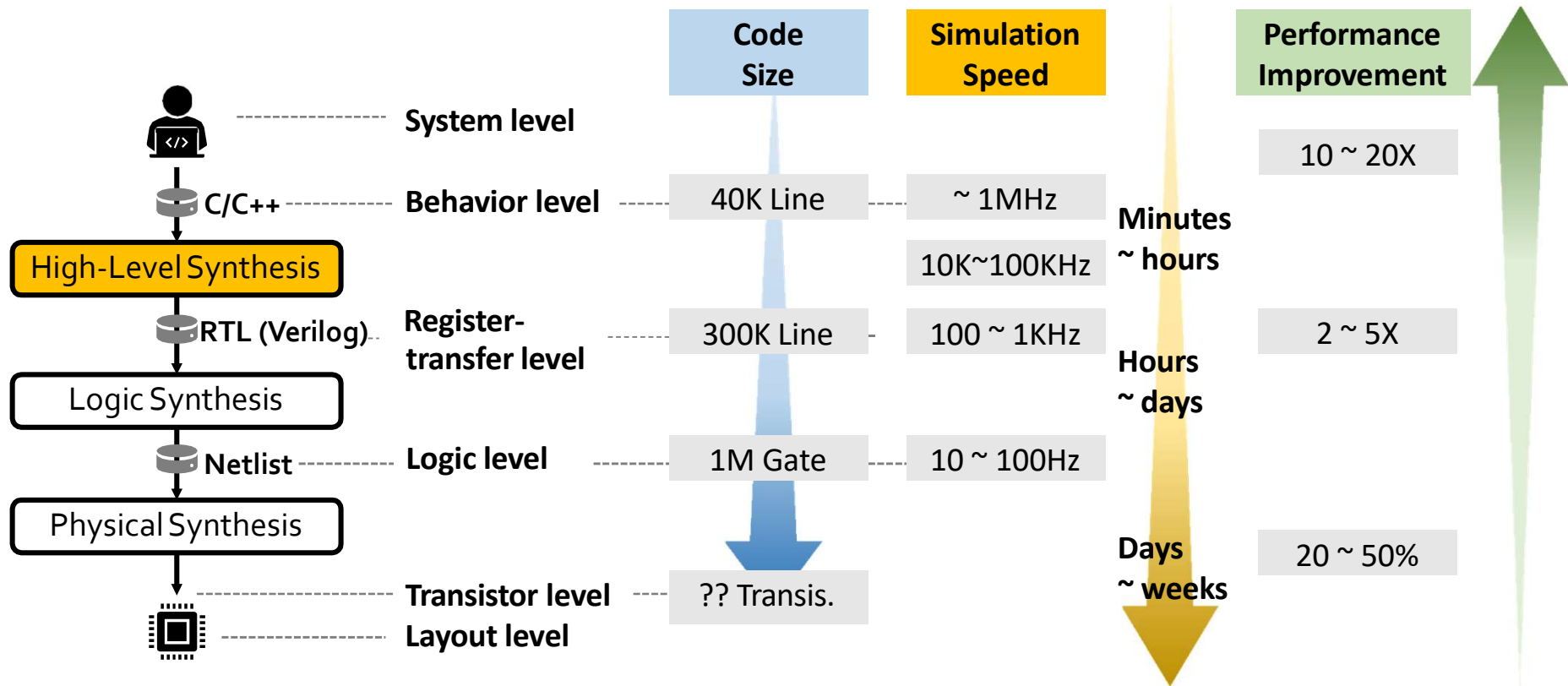
HLS
Tools
→
✂

```
44 req 32 repeat (1) 0
45 rep (posedg
46 #1 33 req 15 //
47 end 34 req 16 alw
48 35rep ~clk;
49 // (posedg 17
50 arb 36 req 18 ini
51 clk 37 req 19 $du
52 rst 38rep ("arbit
... (posedg 20 $du
39 req 21 clk
22 rst
...
1 `include "xxx.v"
2 module top ();
3
4 reg clk;
5 reg rst;
6 reg req3;
7 reg req2;
8 reg req1;
9 reg req0;
10 wire gnt3;
11 wire gnt2;
12 wire gnt1;
13 wire gnt0;
```

Register-Transfer-Level (RTL):
Verilog/VHDL



为什么用HLS ?





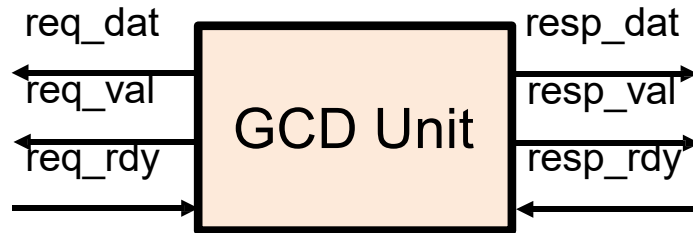
为什么用HLS?

- **Productivity (高产性)**
 - 设计复杂度低, 同时更快的仿真速度
 - Ease-of-use: C/C++/Python v.s. Verilog
- **Portability (便捷性)**
 - 一套C代码 → 多套硬件实现(不同devices)
- **Permutability (可调整)**
 - 高层更多优化的选择
 - 更快的设计空间探索 → 高质量的硬件 (QoR)
- **小结:**
 - 硬件加速器设计门槛降低
 - 硬件设计迭代周期缩短
 - 快速仿真、迭代验证
 - 早期发现设计问题



示例：C vs Verilog

- ▶ A GCD unit with handshake



HLS C Code

```
void GCD ( msg& req,
           msg& resp ) {
    short a = req.msg_a;
    short b = req.msg_b;
    while ( a != b ) {
        if ( a > b )
            a = a - b;
        else
            b = b - a;
    }
    resp.msg = a;
}
```

Manual RTL (partial)

```
module GcdUnitRTL
(
    input wire [ 0:0] clk,
    input wire [ 31:0] req_dat,
    output wire [ 0:0] req_rdy,
    input wire [ 0:0] req_val,
    input wire [ 0:0] reset,
    output wire [ 15:0] resp_dat,
    input wire [ 0:0] resp_rdy,
    output wire [ 0:0] resp_val
);
```

```
always @(*) begin
    if ((curr_state__0 == STATE_IDLE))
        if (req_val)
            next_state__0 = STATE_CALC;

    if ((curr_state__0 == STATE_CALC))
        if (!(is_a_lt_b && is_b_zero))
            next_state__0 = STATE_DONE;

    if ((curr_state__0 == STATE_DONE))
        if ((resp_val && resp_rdy))
            next_state__0 = STATE_IDLE;
end
```

Module declaration

State transition

```
always @(*) begin
    if ((current_state__1 == STATE_IDLE))
        req_rdy = 1; resp_val = 0;
        a_mux_sel = A_MUX_SEL_IN; b_mux_sel = B_MUX_SEL_IN;
        a_reg_en = 1; b_reg_en = 1;

    if ((current_state__1 == STATE_CALC))
        do_swap = is_a_lt_b; do_sub = ~is_b_zero;
        req_rdy = 0; resp_val = 0;
        a_mux_sel = do_swap ? A_MUX_SEL_B : A_MUX_SEL_SUB;
        a_reg_en = 1; b_reg_en = do_swap;
        b_mux_sel = B_MUX_SEL_A;

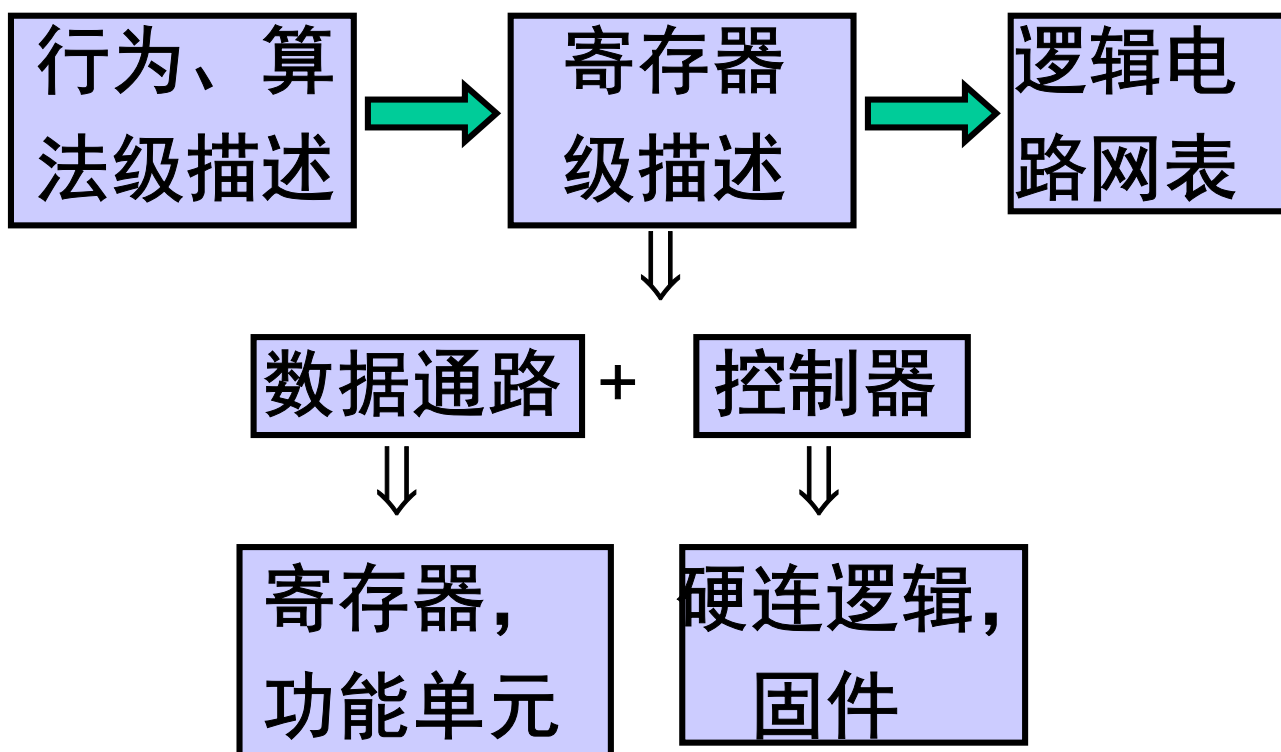
    else
        if ((current_state__1 == STATE_DONE))
            req_rdy = 0; resp_val = 1;
            a_mux_sel = A_MUX_SEL_X; b_mux_sel = B_MUX_SEL_X;
            a_reg_en = 0; b_reg_en = 0;
end
```

Output logic



HLS的基本概念

◆ 高层次综合的概念

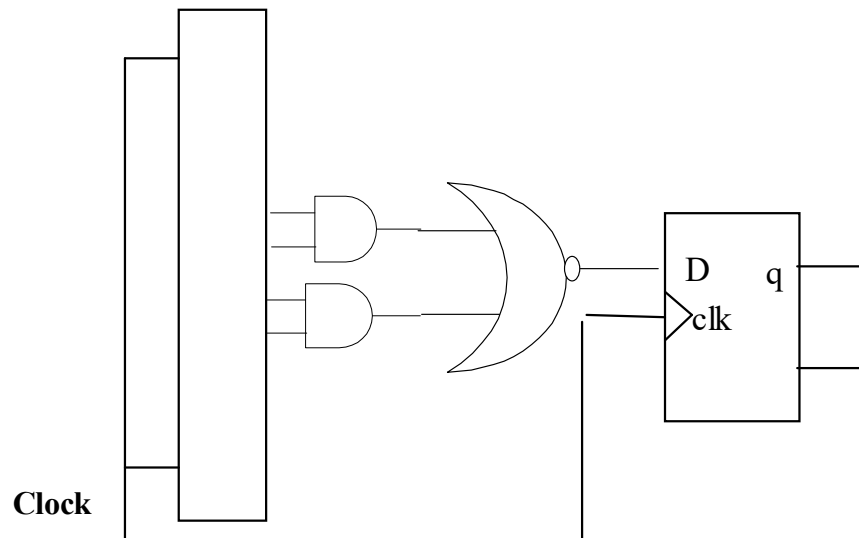




设计级别比较

◆ 门级设计：

- 固定时钟周期和延迟
- 改变时钟周期：
 - 重新设计，重新输入逻辑图，重新模拟
- 改变延迟：
 - 重新确定结构，重新设计，重新输入、模拟
- 与工艺紧密相关

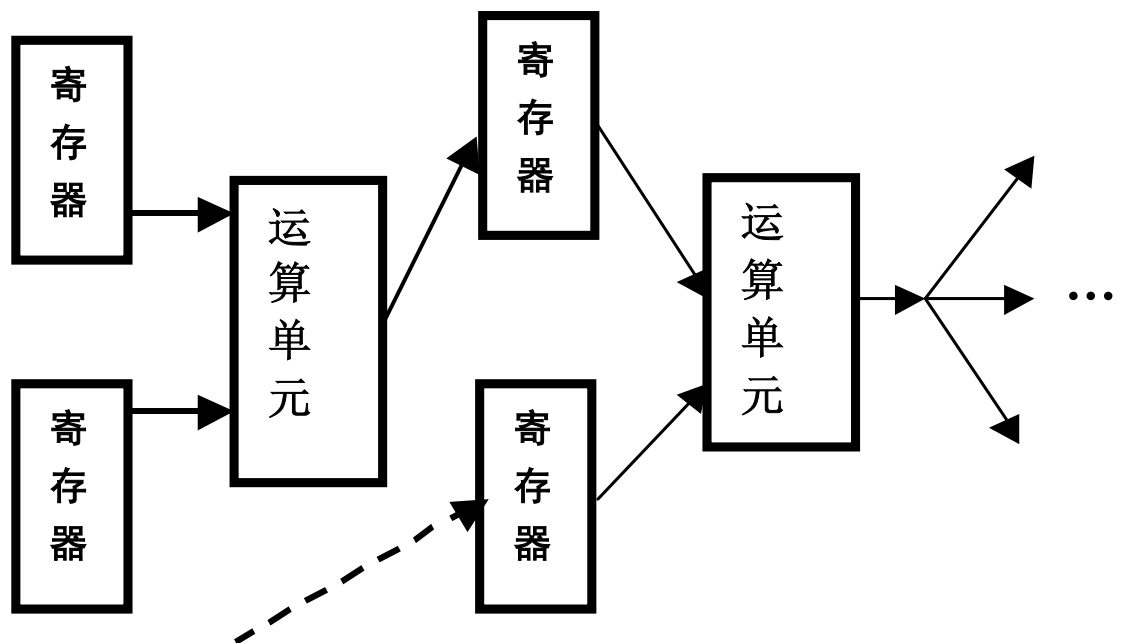




设计级别比较

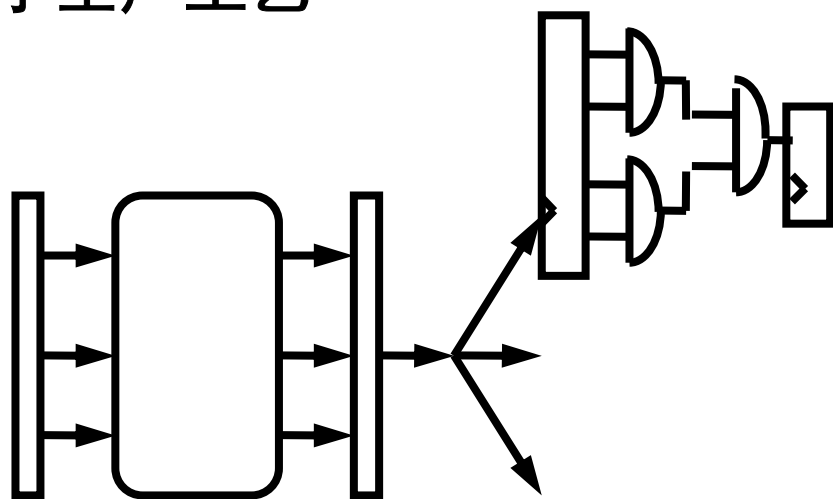
◆ RTL级设计：

- 与工艺无关；
- 设计中对控制步、寄存器、操作流程已明确规定；
- 最终电路实现与综合工具有关。





- 可变时钟周期
- 固定延迟
- 改变时钟周期重新综合
- 改变延迟需要重新确定结构，重新设计VHDL编码，重新综合
- 不受限于生产工艺





设计级别比较

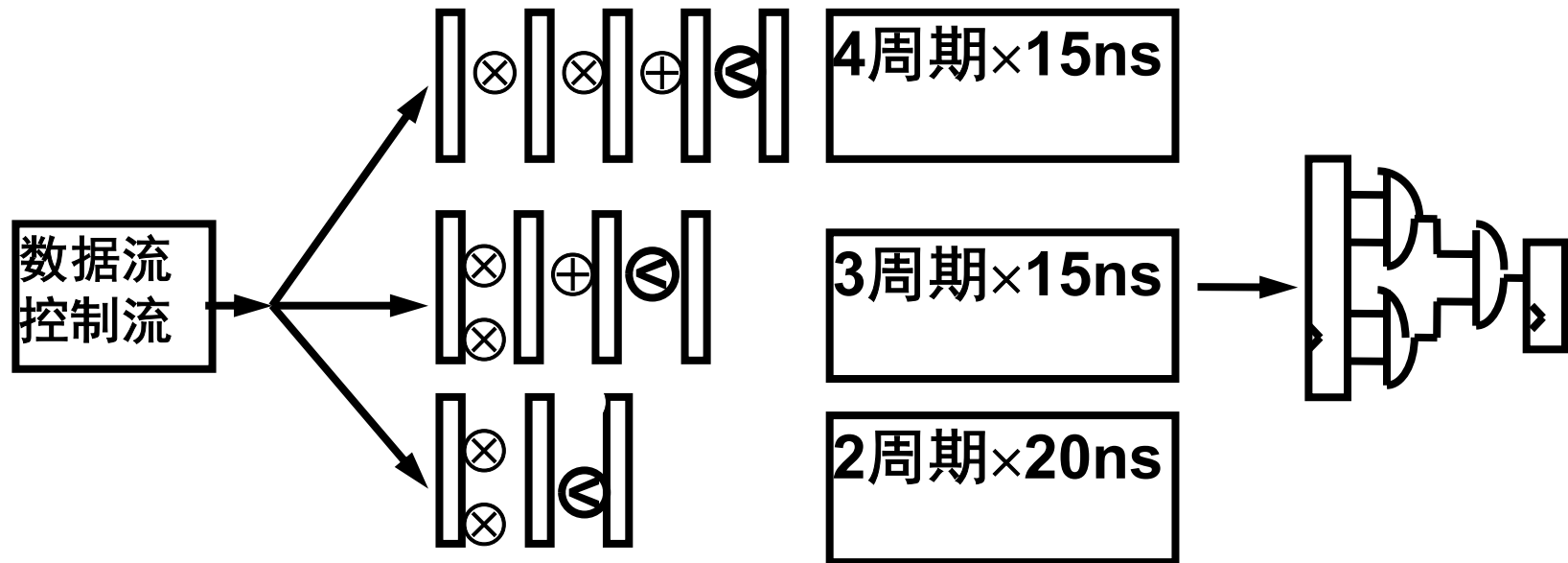
◆ 行为级设计：

- 与工艺无关；
- 设计中对电路的行为已明确规定；
- 电路的实现方案由综合工具确定，包括：
 - 控制步数；
 - 时钟周期；
 - 寄存器总数；
 - 运算单元的品种和数量.....



◆ 行为级设计

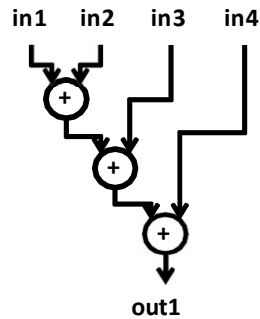
- 可变时钟周期
- 可变延迟
- 改变周期或延迟，只要重新综合





示例：Single Untimed Source to Multiple Implementations

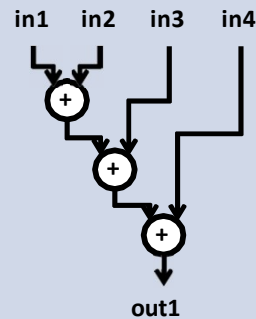
Untimed



Control-Data
Flow Graph
(CDFG)

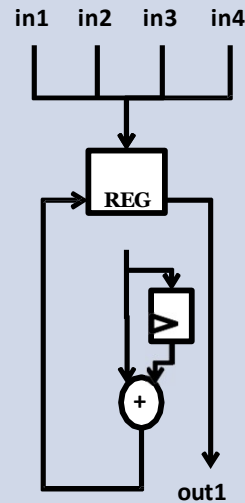
$$out1 = f(in1, in2, in3, in4)$$

Combinational
for **Latency**



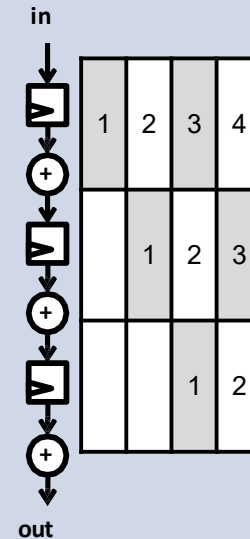
$$\begin{aligned} t_{clk} &\approx 3 * d_{add} \\ T_1 &= 1 / t_{clk} \\ A_1 &= 3 * A_{add} \end{aligned}$$

Sequential
for **Area**



$$\begin{aligned} t_{clk} &\approx d_{add} + d_{setup} \\ T_2 &= 1 / (3 * t_{clk}) \\ A_2 &= A_{add} + 2 * A_{reg} \end{aligned}$$

Pipelined
for **Throughput**



$$\begin{aligned} t_{clk} &\approx d_{add} + d_{setup} \\ T_3 &= 1 / t_{clk} \\ A_3 &= 3 * A_{add} + 6 * A_{reg} \end{aligned}$$



设计举例

◆ 复数乘法:

$$(a + bj) * (c + dj)$$

- 实部:

$$RE = (a * c) - (b * d);$$

- 虚部:

$$IM = (a * d) + (b * c);$$

- 假定:

➢ a, b, c, d的字长为 5 bit;

➢ 时钟周期: 40 ns;

延迟时间: $40\text{ns} * 6 = 240\text{ ns}$;

面积: 2438门;

实现方案之一 (ASAP) :

	得到	相应操作
Step1:	A	Read
Step2:	B	Read
Step3:	C	Read
Step4:	D	Read
Step5:	AC, BD, AD, BC,	* (4个)
Step6:	RE, IM,	+/- (2个)

5 bit 寄存器 4 个;
10 bit 寄存器 6 个;
乘法器 4个;
加法器 2个;



设计举例

◆ 实现方案之二（面积小）：

	得到	相应操作
Step1:	A	Read
Step2:	B	Read
Step3:	C	Read
Step4:	D	Read
Step5:	AC,	* (1个)
Step6:	AD,	* (1个)
Step7:	BD,	* (1个)
Step8:	BC,	* (1个)
Step9:	RE,	+/- (1个)
Step10:	IM,	+/- (1个)



- 5 bit 寄存器 4 个;
- 10 bit 寄存器 6 个;
- 乘法器 1个;
- 加（减）法器 1个;

延迟时间: $40\text{ns} * 10 = 400 \text{ ns};$

面积: 1482门;



设计改进1

Step 操作	1	2	3	4	5
READ	A	B	C	D	
$(*)_1$			$A * C$	$A * D$	
$(*)_2$			$B * C$	$B * D$	
$(+/-)_1$					$A * D + B * C$
$(+/-)_2$					$A * C - B * D$

所需部件				结果统计	
乘法器	加法器	5 bit 寄存器	10 bit 寄存器	运算时间	面积
2	2	2	6	$40\text{ns} * 5 = 200\text{ns}$	1674



设计改进2

Step 操作	1	2	3	4	5	6	7
READ	A	B	C	D			
*			A * C	A * D	B * C	B * D	
+/-						A*D + B*C	A*C - B*D

所需部件				结果统计	
乘法器	加法器	5 bit 寄存器	10 bit 寄存器	运算时间	面积
1	1	3	5	40ns * 7 = 280ns	1264

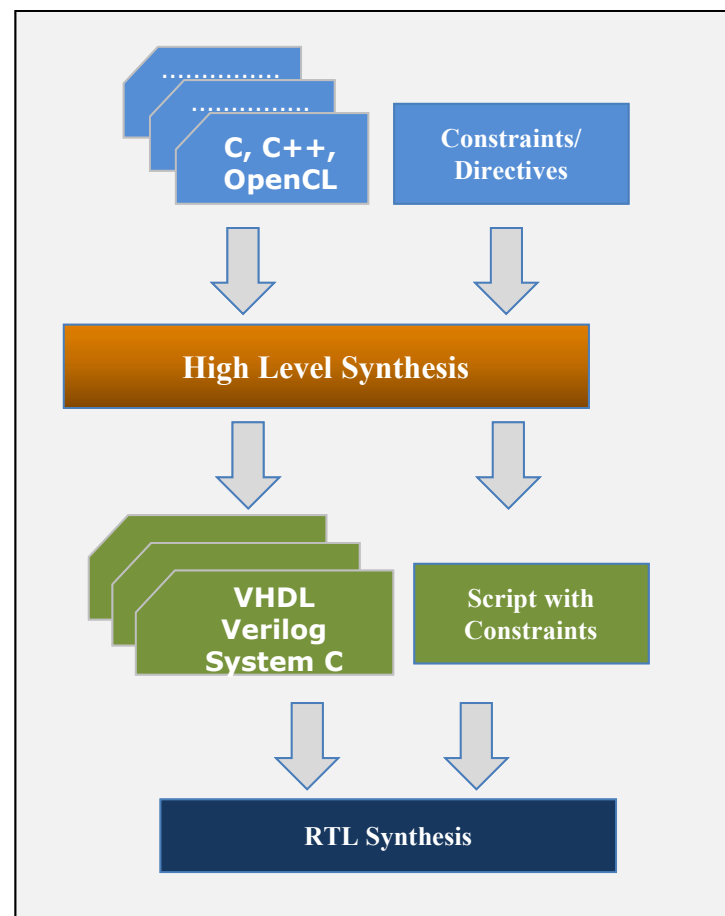
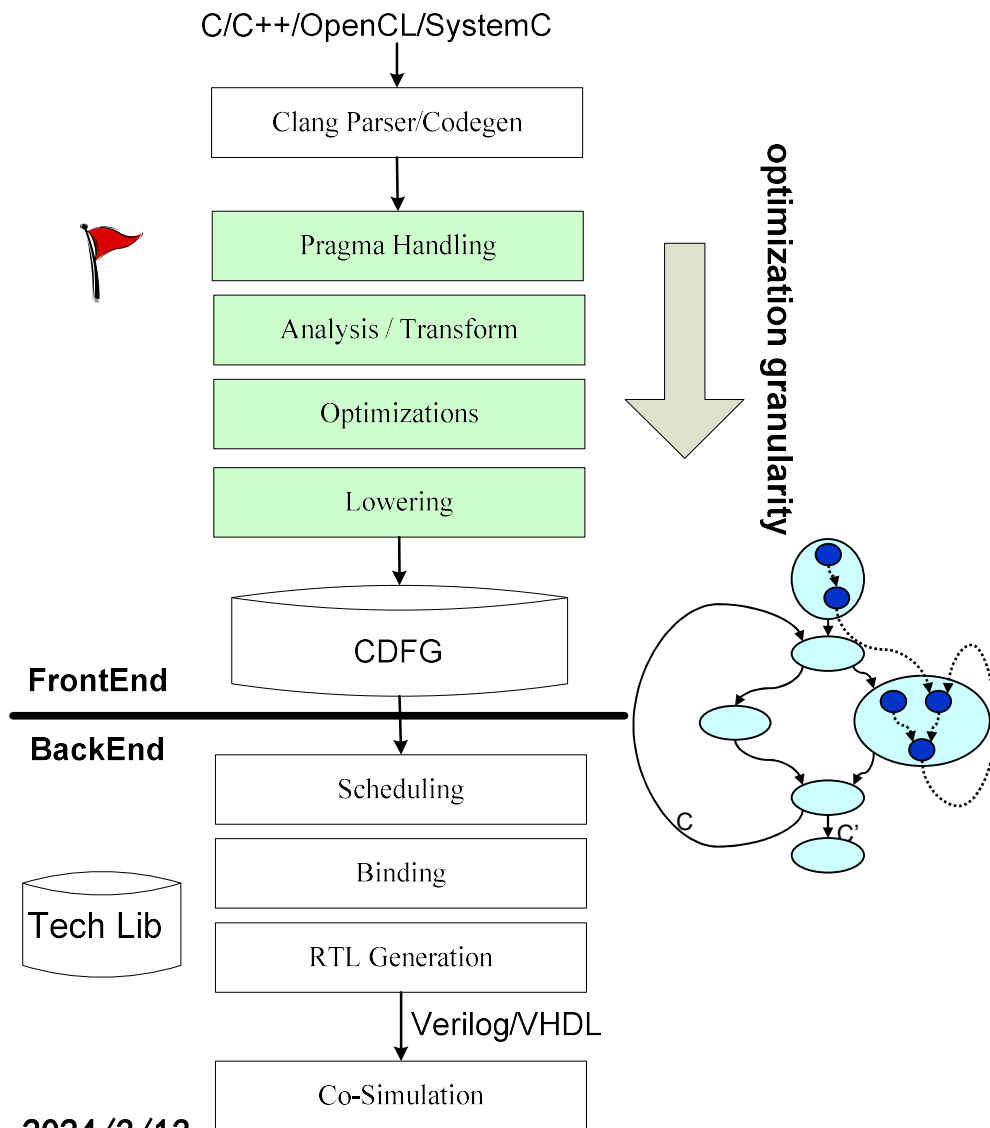


高层次综合的优点

- ◆ 高层次综合的实质是从行为描述到电路结构描述（一般指RTL级）的转换，是从较高抽象层次的描述到较低抽象层次描述的转换。
- ◆ 硬件设计者用行为描述的方式描述设计，有如下优点：
 - 简练；
 - 概念清晰；
 - 易于修改和排除错误；
 - 缩短设计周期。
- ◆ 高层次综合和较低层次的逻辑综合相比，技术复杂性增大很多。 → 把困难交给EDA工具！



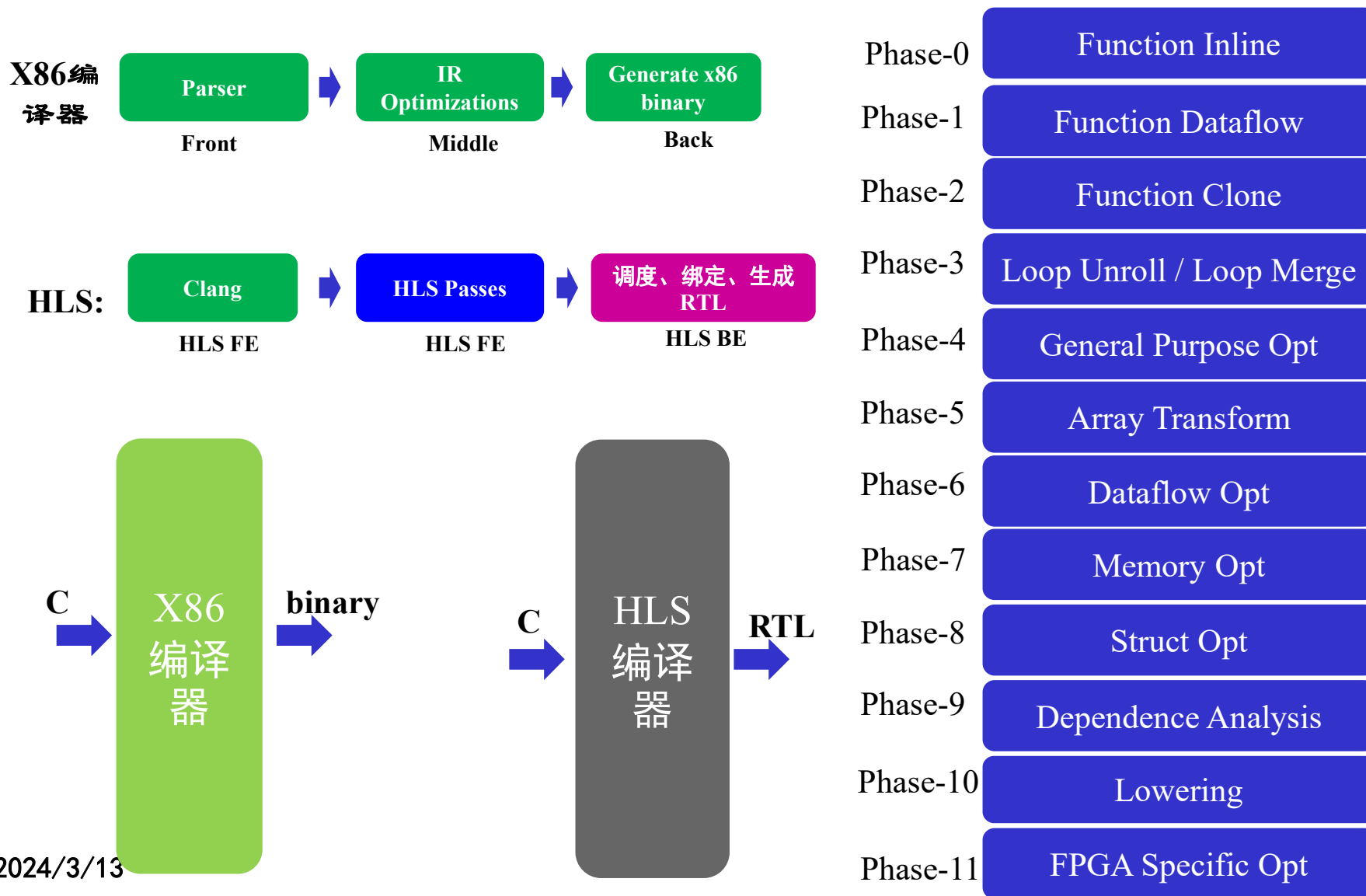
目前商业HLS工具全流程 (Xilinx)



2024/3/13



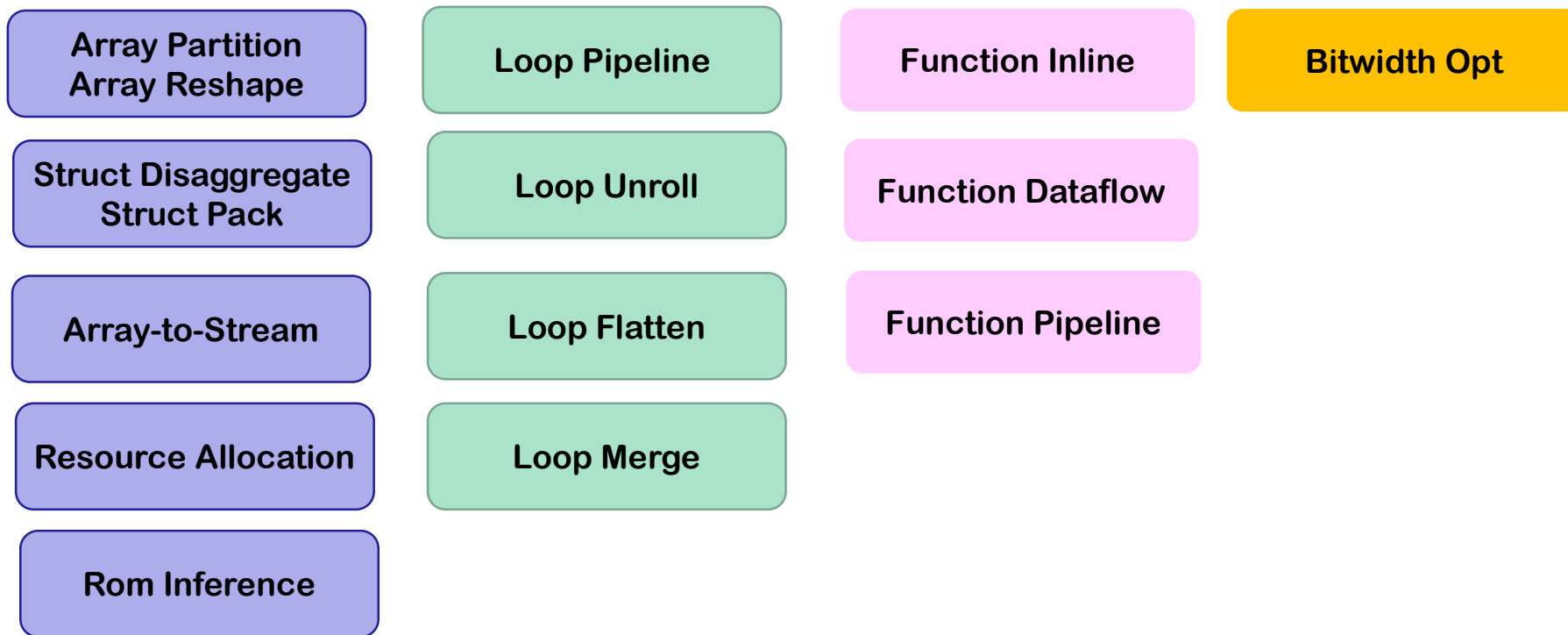
3.2 前端：基于C的编译优化





HLS通常采用的方法

- 用户控制(pragma)
 - 根据用户自己的需求对loop/branch/array/struct做特殊处理
- 自动优化





Hardware Specialization with HLS

- Where does performance gain come from? Specialization!
- **Data type specialization**
 - Arbitrary-precision fixed-point, custom floating-point
- **Interface/communication specialization**
 - Streaming, memory-mapped I/O, etc.
- **Memory specialization**
 - Array partitioning, data reuse, etc.
- **Compute specialization**
 - Unrolling, pipelining, dataflow, multithreading, etc.
- **Architecture specialization**
 - Pipelined, recursive, hybrid, etc.

1. System-level

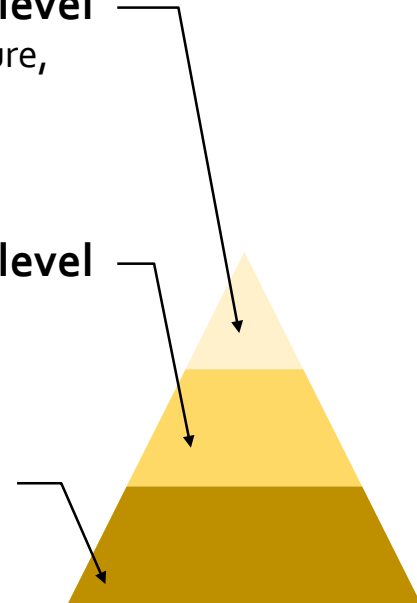
- Architecture, interface

2. Module-level

- Compute, memory

3. Bit-level

- Data type





HLS Pragmas

- All about “**pragma**”s: instructions to tell your compiler how to build the hardware
- This link has all the **pragmas** you need:

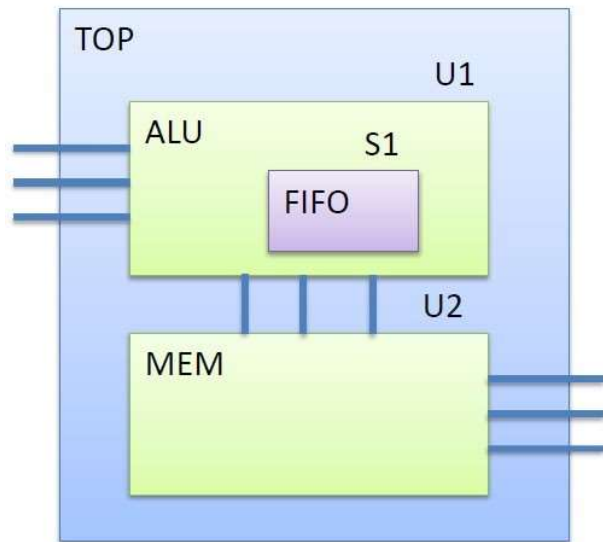
○ <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS->

Kernel Optimization	<ul style="list-style-type: none">• pragma HLS aggregate• pragma HLS bind_op• pragma HLS bind_storage• pragma HLS expression_balance• pragma HLS latency• pragma HLS reset• pragma HLS top	Loop Unrolling	<ul style="list-style-type: none">• pragma HLS unroll• pragma HLS dependence
		Loop Optimization	<ul style="list-style-type: none">• pragma HLS loop_flatten• pragma HLS loop_merge• pragma HLS loop_tripcount
Function Inlining	<ul style="list-style-type: none">• pragma HLS inline	Array Optimization	<ul style="list-style-type: none">• pragma HLS array_partition• pragma HLS array_reshape
Interface Synthesis	<ul style="list-style-type: none">• pragma HLS interface	Structure Packing	<ul style="list-style-type: none">• pragma HLS aggregate• pragma HLS dataflow
Task-level Pipeline	<ul style="list-style-type: none">• pragma HLS dataflow• pragma HLS stream	Resource Optimization	<ul style="list-style-type: none">• pragma HLS allocation• pragma HLS function_instantiate
Pipeline	<ul style="list-style-type: none">• pragma HLS pipeline• pragma HLS occurrence		



HDL Structures and Hierarchy

- Hierarchical HDL structures are achieved by defining modules (**definition**) and instantiating modules (**instance**)
 - Instantiation is the process of “calling” a module



```
module TOP ( port_list );  
    ALU U1 ( port_connection );  
    MEM U2 ( port_connection );  
endmodule
```

Instance

```
module ALU ( port_list );  
    FIFO S1 ( port_connection );  
endmodule
```

Definition

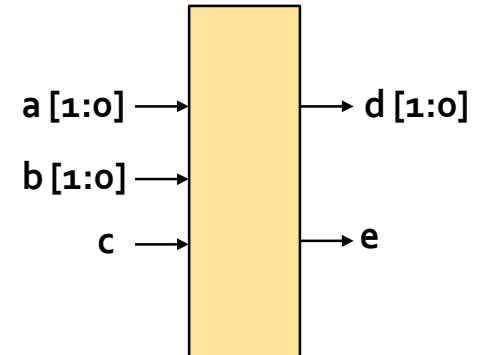


Module Ports

- **Module header starts with module keyword, contains the I/O ports**
- **Port declarations begins with output, input or inout follow by bus indices**
 - Provide the interface by which a module can communicate with the environment

```
module S1 (a, b, c, d, e);  
input [1:0] a, b;  
input c;  
output reg [1:0] d;  
output e;  
//Verilog 1995 Style  
endmodule
```

```
module S2 (input [1:0] a, b,  
          input c,  
          output reg [1:0] d,  
          output e);  
  
//ANSI C Style  
endmodule
```





Basic Mapping Rule from C/C++ to RTL

C Constructs		RTL Components
Functions	→	Modules
Arguments	→	I/O Ports
Operators (+, *)	→	Functional units (adder, multiplier)
Scalars	→	Wires or registers
Arrays	→	Memory
Control flows	→	Control logics (Finite State Machine)



Basic Mapping Rule from C/C++ to RTL

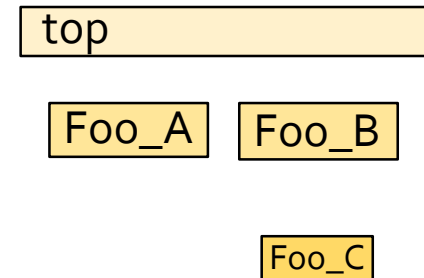
C Constructs		RTL Components
Functions	→	Modules
Arguments	→	I/O Ports
Operators (+, *)	→	Functional units (adder, multiplier)
Scalars	→	Wires or registers
Arrays	→	Memory
Control flows	→	Control logics (Finite State Machine)

C Source Code

```
void Foo_C() {...}
void Foo_A() {...}
Void Foo_B() {
    Foo_C();
}

void top() {
    Foo_A();
    Foo_B();
    ...
    Foo_B();
}
```

RTL Hierarchy



Resource sharing:
only one *instance* of
Foo_B on hardware

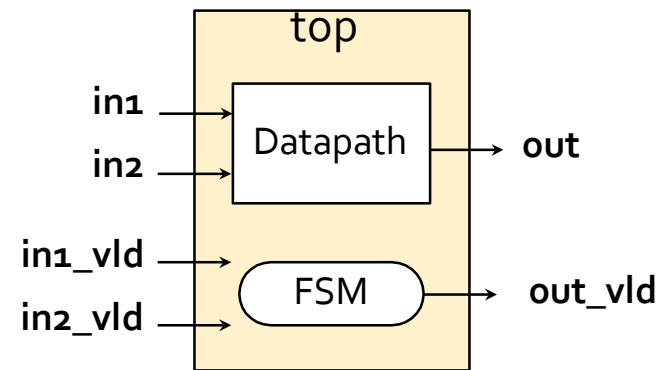


Basic Mapping Rule from C/C++ to RTL

C Constructs		RTL Components
Functions	→	Modules
Arguments	→	I/O Ports
Operators (+, *)	→	Functional units (adder, multiplier)
Scalars	→	Wires or registers
Arrays	→	Memory
Control flows	→	Control logics (Finite State Machine)

C Source Code

```
void top(int* in1, int* in2, int* out) {  
    *out = *in1 + *in2;  
}
```



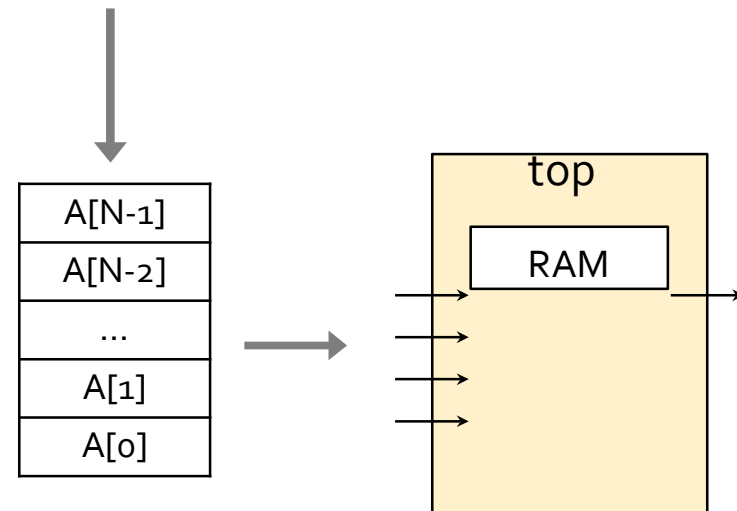


Basic Mapping Rule from C/C++ to RTL

C Constructs		RTL Components
Functions	→	Modules
Arguments	→	I/O Ports
Operators (+, *)	→	Functional units (adder, multiplier)
Scalars	→	Wires or registers
Arrays	→	Memory
Control flows	→	Control logics (Finite State Machine)

C Source Code

```
for (i = 0; i < N; i++)  
    A[i+x] = A[i] + i;
```





Deterministic at Compile Time

- On FPGA, memory maps to BRAM
- Everything must be decided at compile time – your hardware cannot be changed while running!
 - Adding one more piece of memory after the circuit is built?

```
int mem[var];
```

```
int mem* = malloc(var * sizeof(int));
```

```
reg [0:7] mem [var:0];
```



8-bit element



8-bit element



8-bit element

...

...

How
many??



Hardware Specialization with HLS

- Where does performance gain come from? [Specialization!](#)

- Data type specialization
 - Arbitrary-precision fixed-point, custom floating-point
- Interface/communication specialization
 - Streaming, memory-mapped I/O, etc.
- **Memory** specialization
 - Array partitioning, data reuse, etc.
- **Compute** specialization
 - Unrolling, pipelining, dataflow, multithreading, etc.
- Architecture specialization
 - Pipelined, recursive, hybrid, etc.

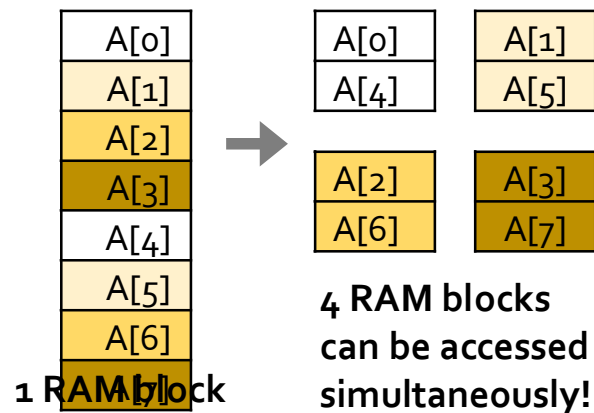


The Three Musketeers
(i) Array partition
(ii) Loop unroll
(iii) Loop pipeline



Array Partition – Memory Parallelism

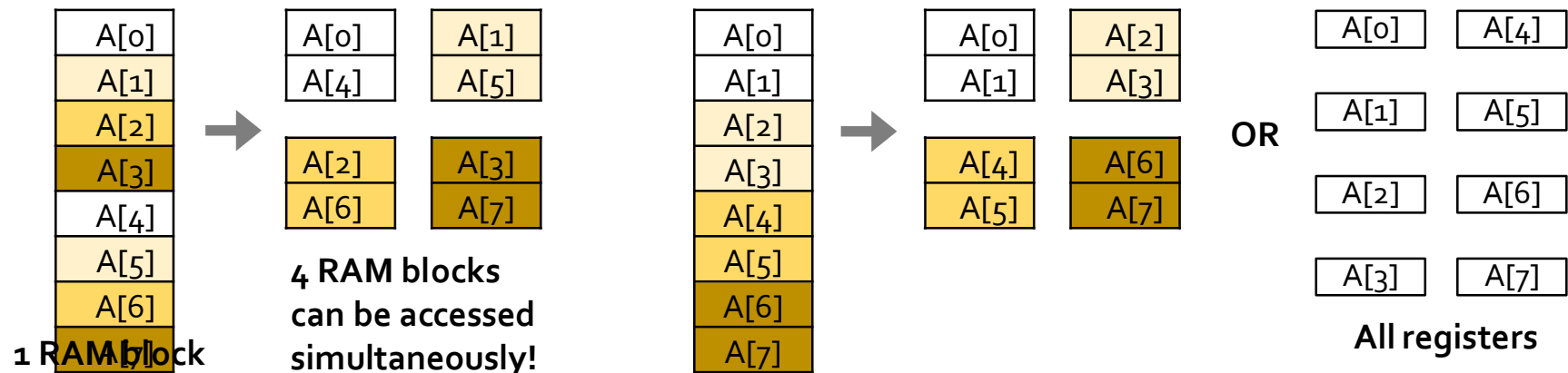
- Initially, an array is mapped to one (or more) block(s) of RAM (or BRAM on FPGA)
 - One block of RAM has at most **two ports**
 - At most **two** read/write operations can be done in **one clock cycle** – Parallelism is **2** (too low)
- An array can be **partitioned** and mapped to **multiple** blocks of RAMs





Array Partition – Memory Parallelism

- Initially, an array is mapped to one (or more) block(s) of RAM (or BRAM on FPGA)
 - One block of RAM has at most **two ports**
 - At most **two** read/write operations can be done in **one clock cycle** – Parallelism is **2** (too low)
- An array can be **partitioned** and mapped to **multiple** blocks of RAMs
 - Can also be partitioned into individual elements and mapped to registers
 - Only if your array is small otherwise the tool will give up





Loop Unrolling

- **Loop unrolling** to expose higher parallelism and achieve shorter latency
 - Pros
 - Decrease loop overhead
 - Increase parallelism for scheduling
 - Cons
 - Increase operation count, which may negatively impact area, power, and timing

Original Loop

```
for (int i = 0; i < N; i++)  
#pragma HLS unroll  
    A[i] = B[i] + C[i];
```

$N \times m$ cycles

Assume $A[i] = B[i] + C[i]$ takes m cycle



Unrolled Loop

```
A[0] = B[0] + C[0];  
A[1] = B[1] + C[1];  
A[2] = B[2] + C[2];  
...
```

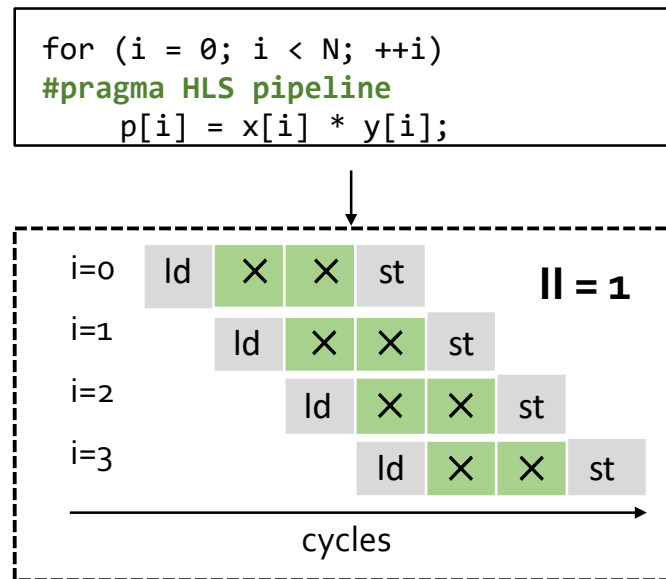
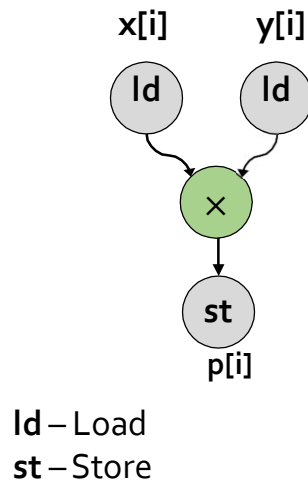
m cycle

Only if A , B , and C are fully partitioned!



Loop Pipelining

- **Loop pipelining** is one of the most important optimizations for high-level synthesis
 - Allows a new iteration to begin processing before the previous iteration is complete
 - Key metric: **Initiation Interval (II)** in # cycles





Put-together: Pipeline + Unroll + Partition

- The three techniques are frequently used together to boost computation efficiency

```
for (int i = 0; i < N; i++) {  
    for (int j = 0; j < M; j++) {  
        A[i][j] = B[i][j] * C[i][j];  
    }  
}
```

A

RAM block 1

A[0][0]	A[0][1]	...	A[0][N-1]
A[1][0]	A[1][1]	...	A[1][N-1]
...
A[M-1][0]	A[M-1][1]	...	A[M-1][N-1]

B

RAM block 2

B[0][0]			

C

RAM block 3

C[0][0]			



Put-together: Pipeline + Unroll + Partition

- The three techniques are frequently used together to boost computation efficiency

```
for (int i = 0; i < N; i++) {  
    for (int j = 0; j < M; j++) {  
#pragma HLS unroll  
        A[i][j] = B[i][j] * C[i][j];  
    }  
}
```

Compute in parallel

RAM block 1

A[0][0]	A[0][1]	...	A[0][N-1]
A[1][0]	A[1][1]	...	A[1][N-1]
...
A[M-1][0]	A[M-1][1]	...	A[M-1][N-1]

Compute in parallel

RAM block 2

B[0][0]			

Compute in parallel

RAM block 3

C[0][0]			

Memory ports limited by 2 → Need to partition

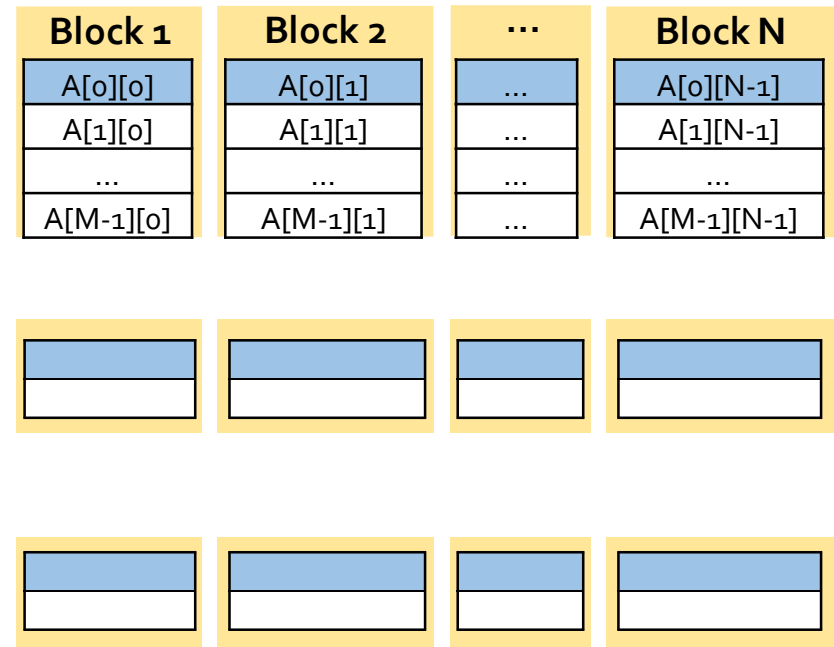


Put-together: Pipeline + Unroll + Partition

- The three techniques are frequently used together to boost computation efficiency

```
#pragma HLS array_partition variable=A dim=2 complete
#pragma HLS array_partition variable=B dim=2 complete
#pragma HLS array_partition variable=C dim=2 complete

for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
#pragma HLS unroll
        A[i][j] = B[i][j] * C[i][j];
    }
}
```



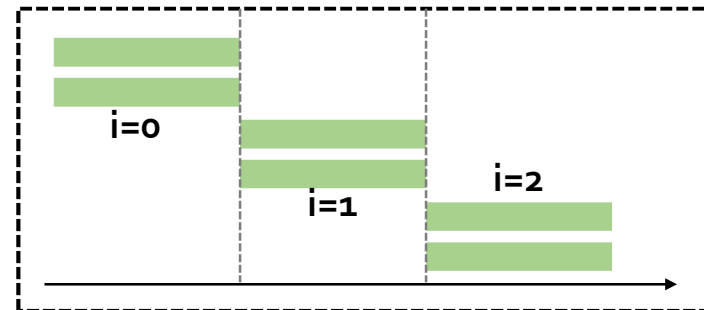


Put-together: Pipeline + Unroll + Partition

- The three techniques are frequently used together to boost computation efficiency

```
#pragma HLS array_partition variable=A dim=2 complete  
#pragma HLS array_partition variable=B dim=2 complete  
#pragma HLS array_partition variable=C dim=2 complete
```

```
for (int j = 0; j < M; j++) {  
    #pragma HLS unroll  
    A[i][j] = B[i][j] * C[i][j];  
}
```



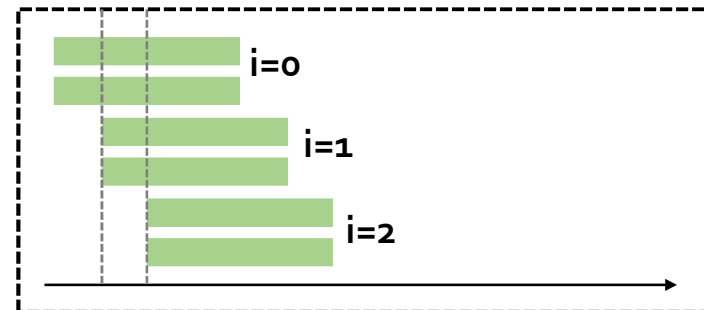
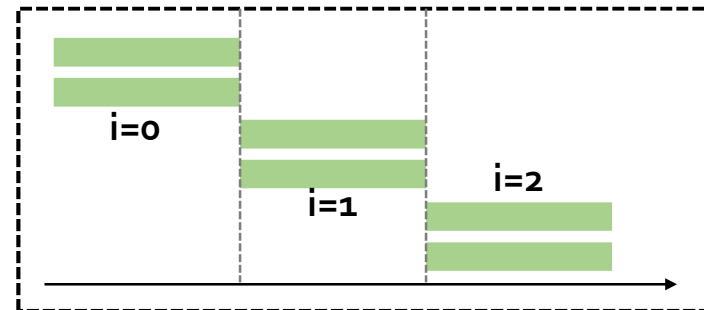


Put-together: Pipeline + Unroll + Partition

- The three techniques are frequently used together to boost computation efficiency

```
#pragma HLS array_partition variable=A dim=2 complete
#pragma HLS array_partition variable=B dim=2 complete
#pragma HLS array_partition variable=C dim=2 complete

for (int i = 0; i < N; i++) {
  #pragma HLS pipeline II=1
  for (int j = 0; j < 32; j++) {
    #pragma HLS unroll factor=8
    A[i][j] = B[i][j] * C[i][j];
  }
}
```



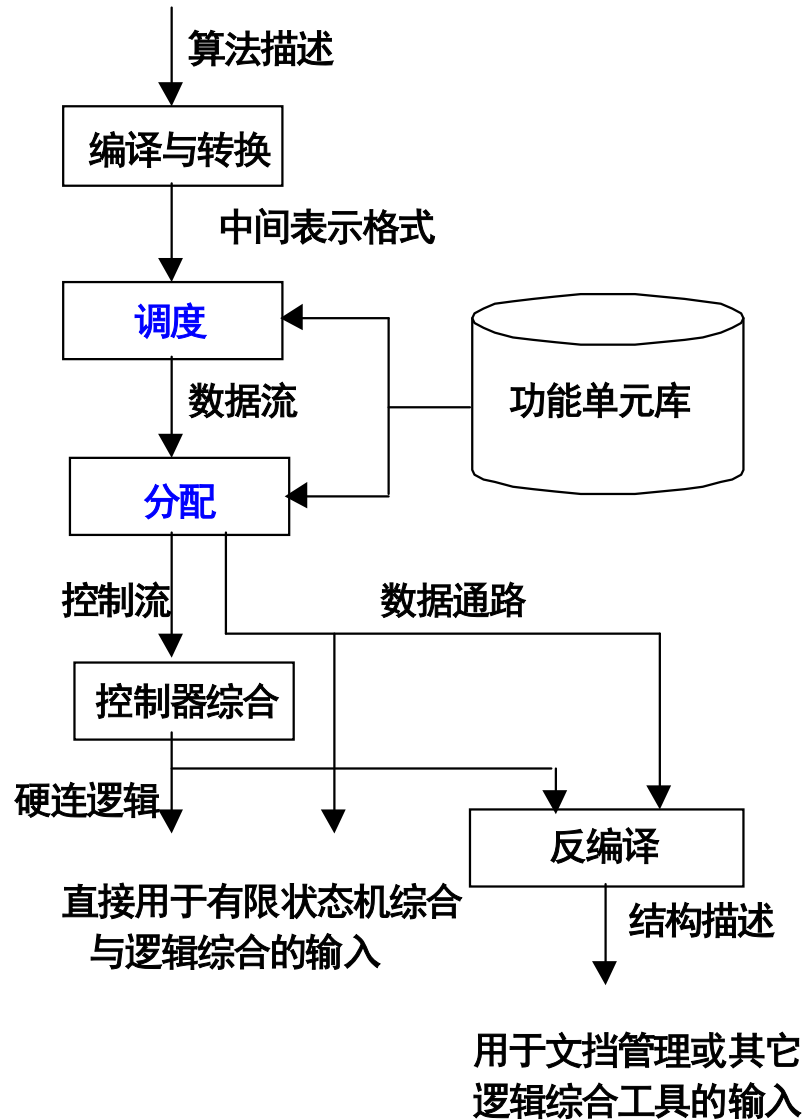


3.3 后端：高层次综合算法

- ◆ 高层次综合流程
- ◆ 数据内部表示
- ◆ 调度
- ◆ 分配
- ◆ 优化



3.3.1 HLS后端的流程



◆ **调度：** 将操作赋给控制步；

◆ **分配：**

- 将操作赋给功能单元；
- 将变量、数值赋给寄存器；



- 2024/3/13



行为描述的一个实例

process

variable v1, v2, v3: integer;

begin

v1 := in1; ----- 1

v2 := in2; ----- 2

while (v1 < v2) **loop** ----- 3

if (cond = '1') **then** ----- 4

v3 := v1 + v2; ----- 5

v2 := in1; ----- 6

else

v3 := v1 - v2; ----- 7

end if;

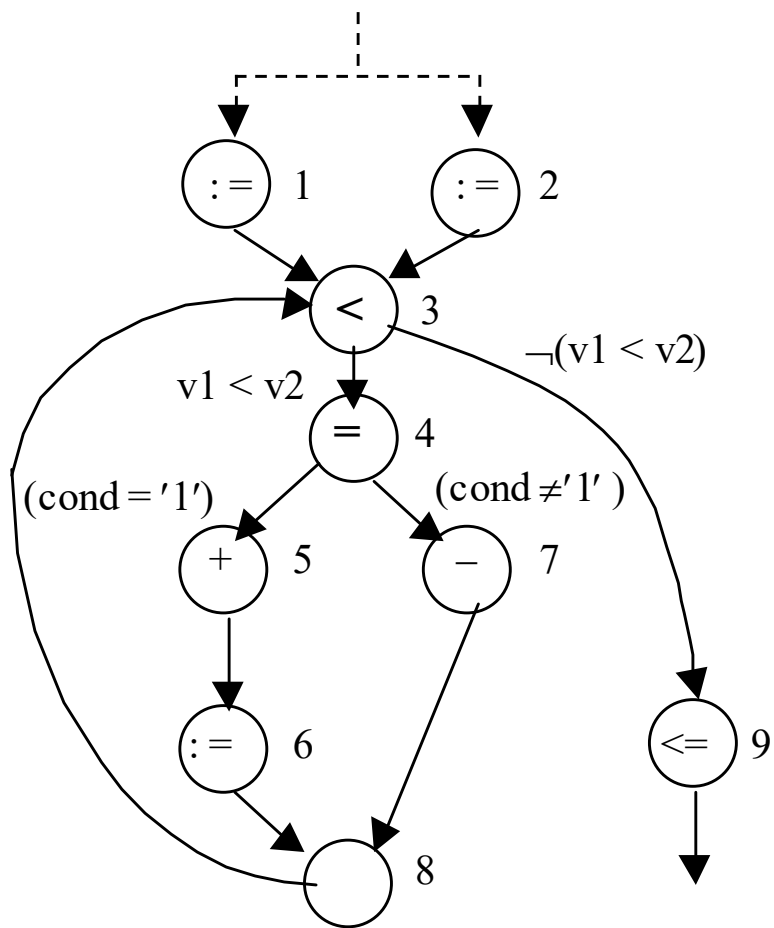
end loop; ----- 8

fout <= v3; ----- 9

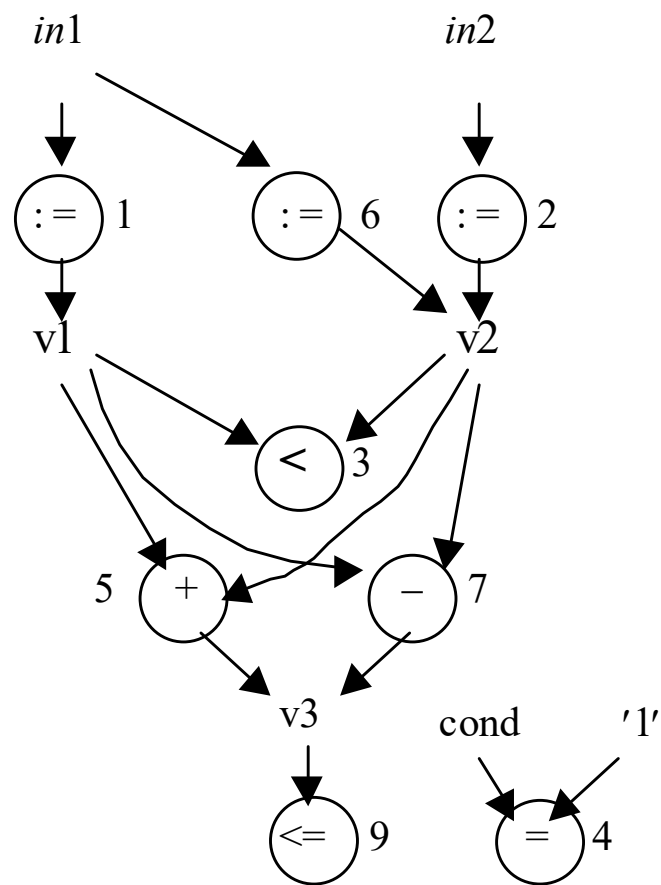
end process;



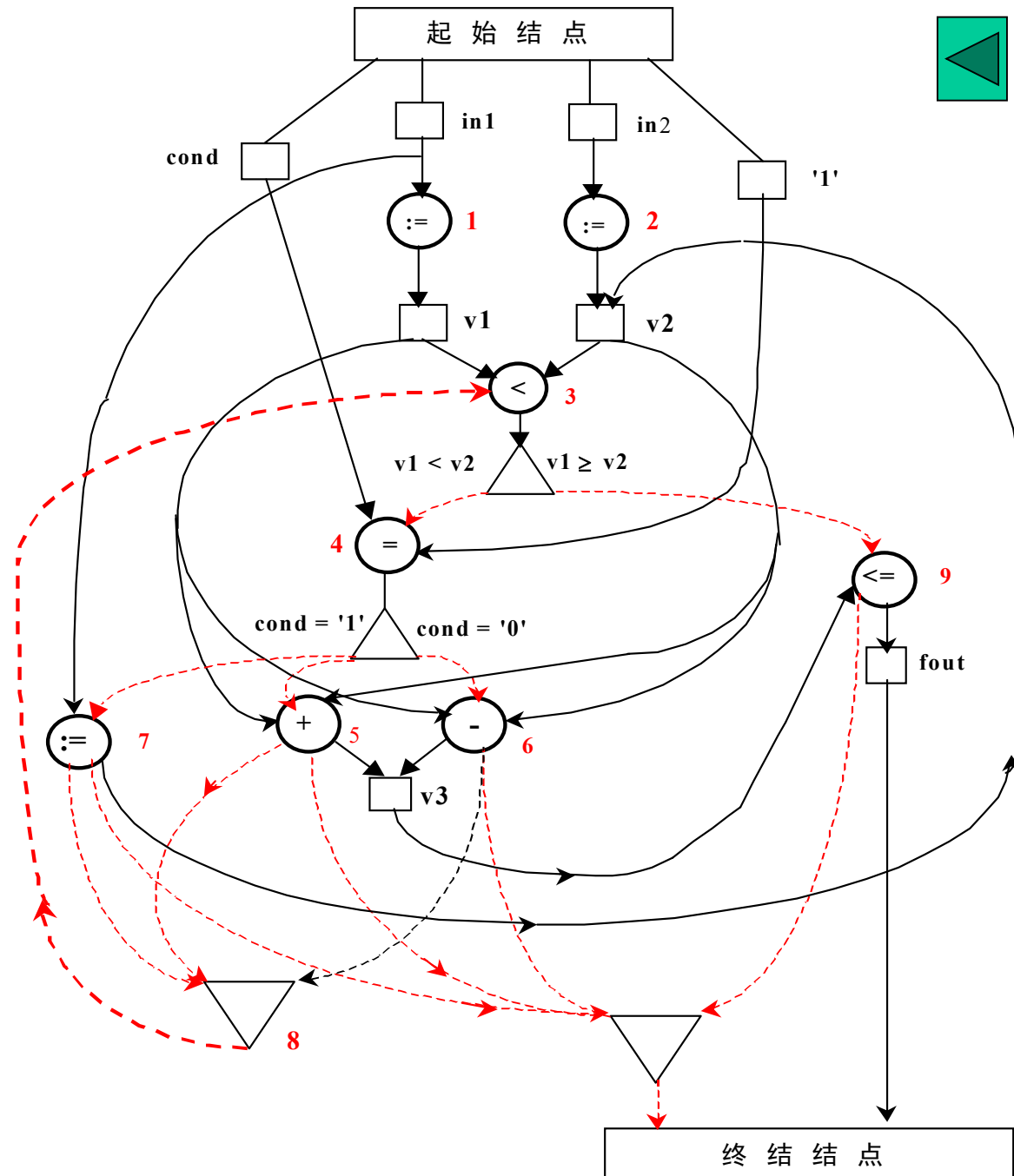
控制流图与数据流图



(a) 控制流图



(b) 数据流图




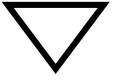




控制数据流图

◆ 将控制流图和数据流图结合在一起；

◆ 节点：

- 操作节点 
- 数据节点（传输节点） 
- 分支节点 
- 汇聚节点 

◆ 边：

- 数据相关边； 
- 控制相关边； 



3.3.1.2 调度与分配

- ◆ 高层次综合是实现从行为描述到结构描述的转换，其核心部分是**调度**（scheduling）和**分配**（allocation）。
- ◆ **调度**是将**操作**赋给**控制步**（control step）。
 - 一个控制步是一个基本时序单位，在同步系统中，通常对应于一个或几个时钟周期。
 - 调度的目标是：在满足约束条件的情况下，将操作赋给各控制步，以使给定目标函数最小。
- ◆ **分配**：
 - 将**操作**赋给相应的**功能单元**进行运算，
 - 将**变量**（或值）赋给**寄存器**加以存放。
 - 分配的目标在于使硬件资源的花费最少，硬件资源包括功能单元、存储单元和数据传输通路。
 - 数据通路的抽象结构必须用具体的部件实现，这一步也称作模块确定（module binding）。也可将其归入分配。



3.3.1.3 控制器综合

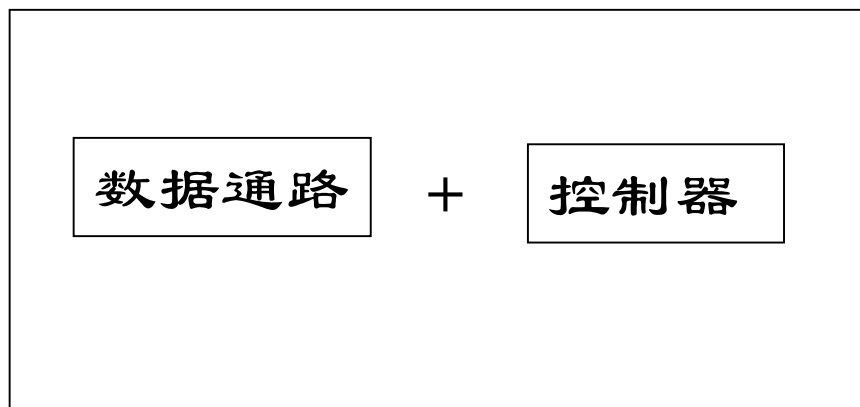
- 调度和分配的输出是控制器综合的输入,
- FSM（有限状态机）描述是二者的接口。





3.3.1.4 结果生成

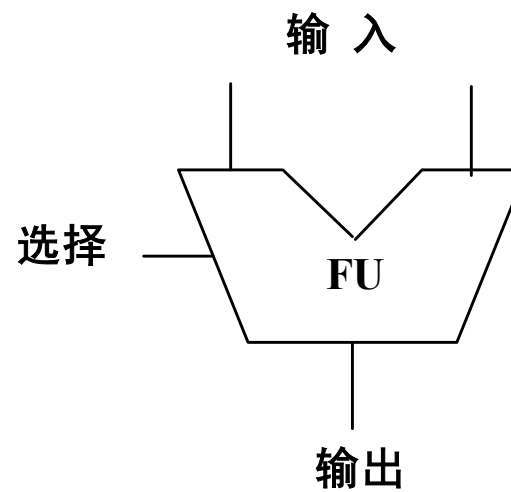
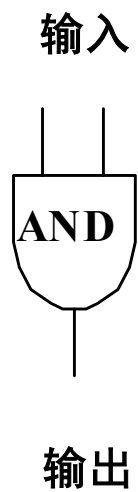
- ◆ 数字系统的行为描述 高层次综合后 RTL描述。
- ◆ 在RTL级，数字系统由数据通路和控制器组成。
- ◆ 控制器用一个有限状态机（或微程序）表示；
- ◆ 数据通路则由3类硬件模块组成的模块集合表示。这3类硬件模块是：
 - 功能单元；
 - 存储单元；
 - 互连线网。



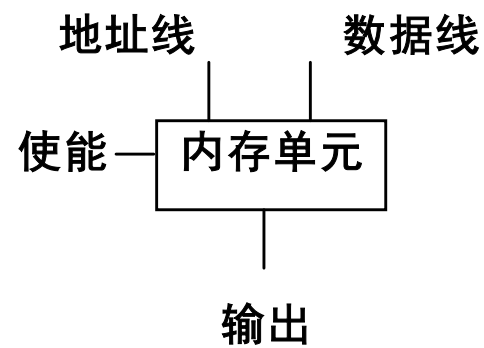
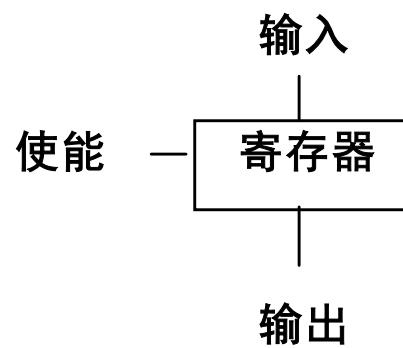


硬件模块图例

功能单元：



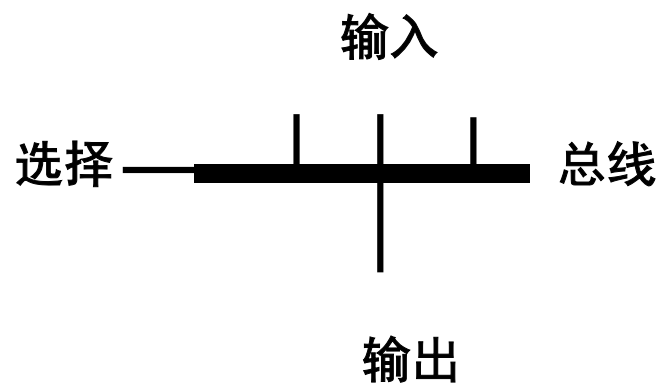
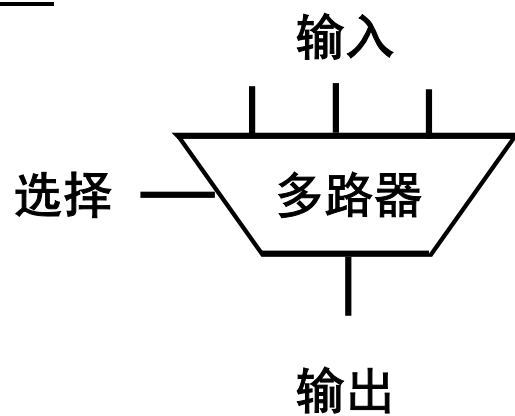
存储单元：





硬件模块图例（续）

互连线网：





(控制器 + 数据通路) 的实例

实现操作:

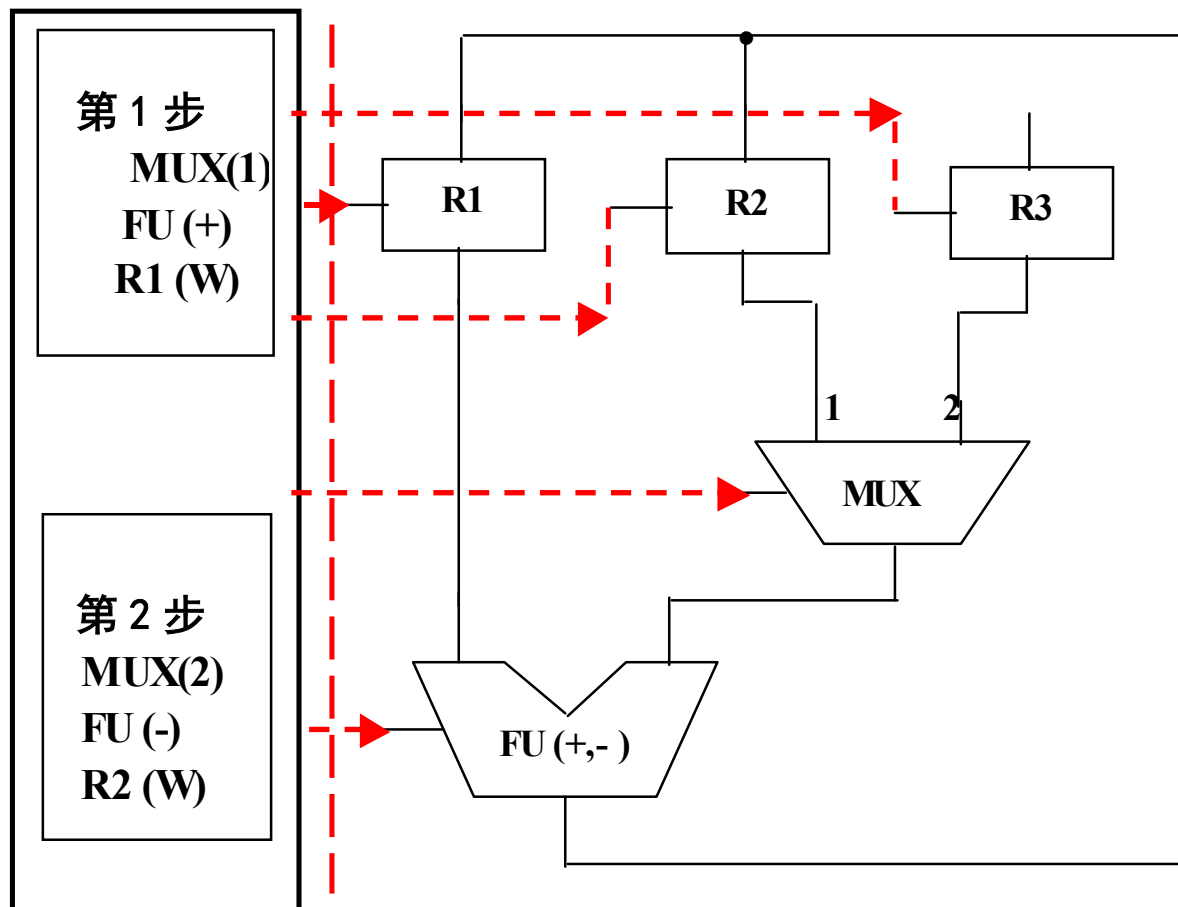
$R1 \leftarrow (R1) + (R2)$

实现操作:

$R2 \leftarrow (R1) - (R3)$

控制器

数据通路





3.3.1.5 设计空间搜索

- ◆ 高层次综合的主要目标：找到一个满足约束条件的最优解。需要检查庞大的设计空间。
 - (1) 基于规则、基于知识的专家系统引导设计空间的搜索。缺点在于搜索时间太长。
 - (2) 启发式算法：基于一些启发式规则，缩小问题的搜索空间，期望在合理的时间内得到问题的近似最优解或满意解。
 - (3) 缩小问题的领域，由于问题的领域缩小，使得设计空间迅速减小，易于找到最优解或满意解。



3.3.2 调度技术

- 3.3.1 调度的基本问题;
- 3.3.2 调度算法的分类;
- 3.3.3 ASAP调度算法与ALAP调度算法;
- 3.3.4 列表调度算法;
- 3.3.5 调度中控制结构的处理;
- 3.3.6 调度中的功能单元库;



3.3.2.1 调度的基本问题

- ◆ 调度的目标：

在满足给定的约束条件下，将行为描述中的各操作赋给相应的控制步，使得给定的目标函数最小。

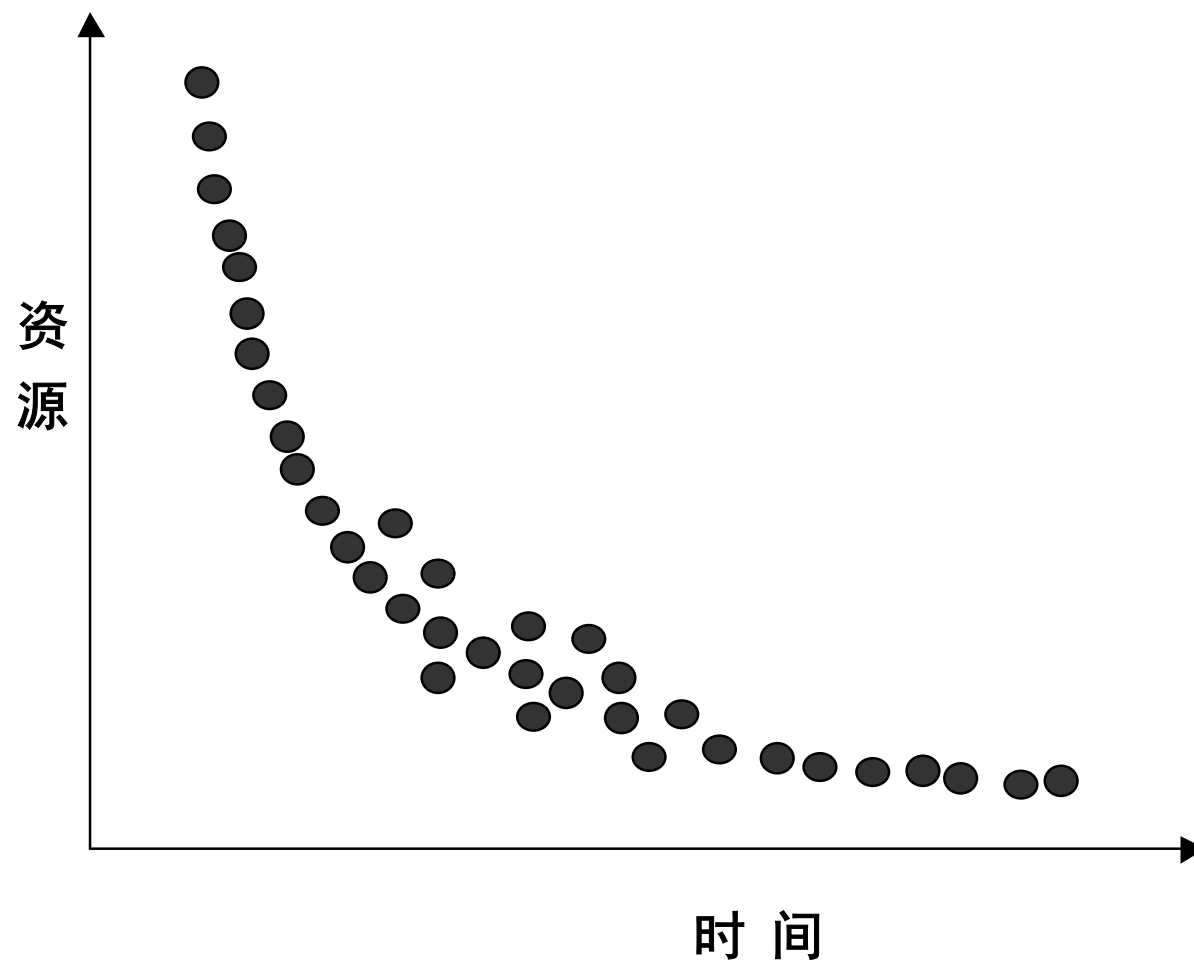
- ◆ 约束条件主要包括：延时、功耗和芯片面积等参数。

- ◆ 硬件本身的约束：

- 每个硬件模块在同一控制步中只能使用1次；
- 每个寄存器在同一时刻只能存储1个值。



调度与时间、资源之间的关系



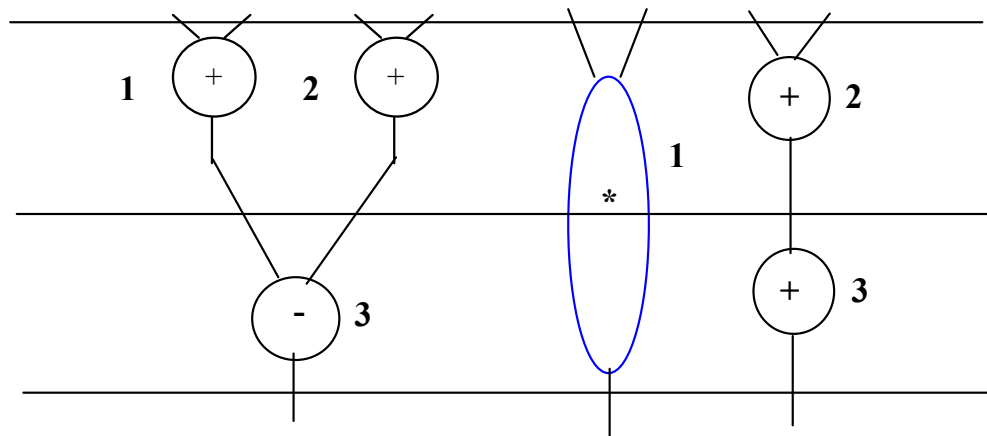


操作的调度类型

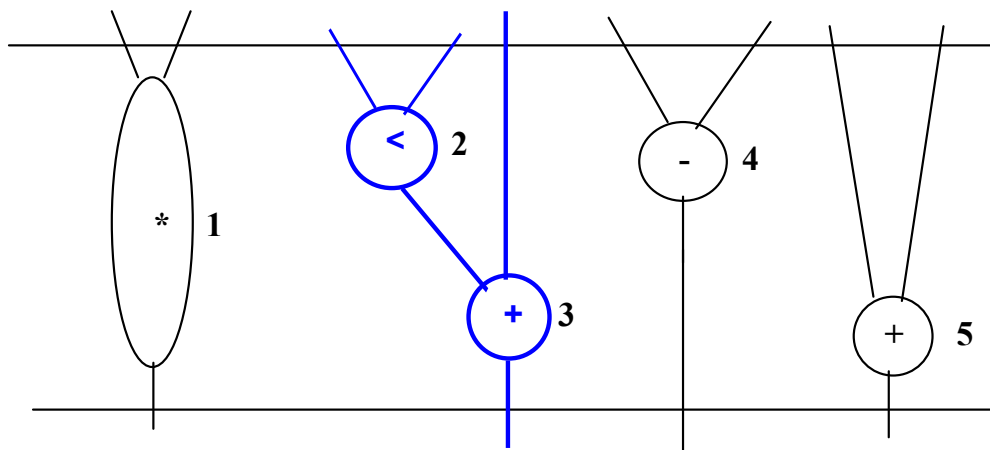
◆ 从调度的角度看，操作可分为：

- 单周期操作；
- 多周期操作；
- 链式操作；

◆ 如果把单周期操作作为一个基准的话，则执行时间较长的操作可以被安排成多周期操作，执行时间较短的可以安排成链式操作。



多周期操作



链式操作



3.3.2.2 调度算法的分类

◆ 依据调度算法分类：

- 变换法（transformation）；

初始方案 → 变换 → 最优调度

- 构造法（constructive）：

选择一个操作 → 对该操作进行调度

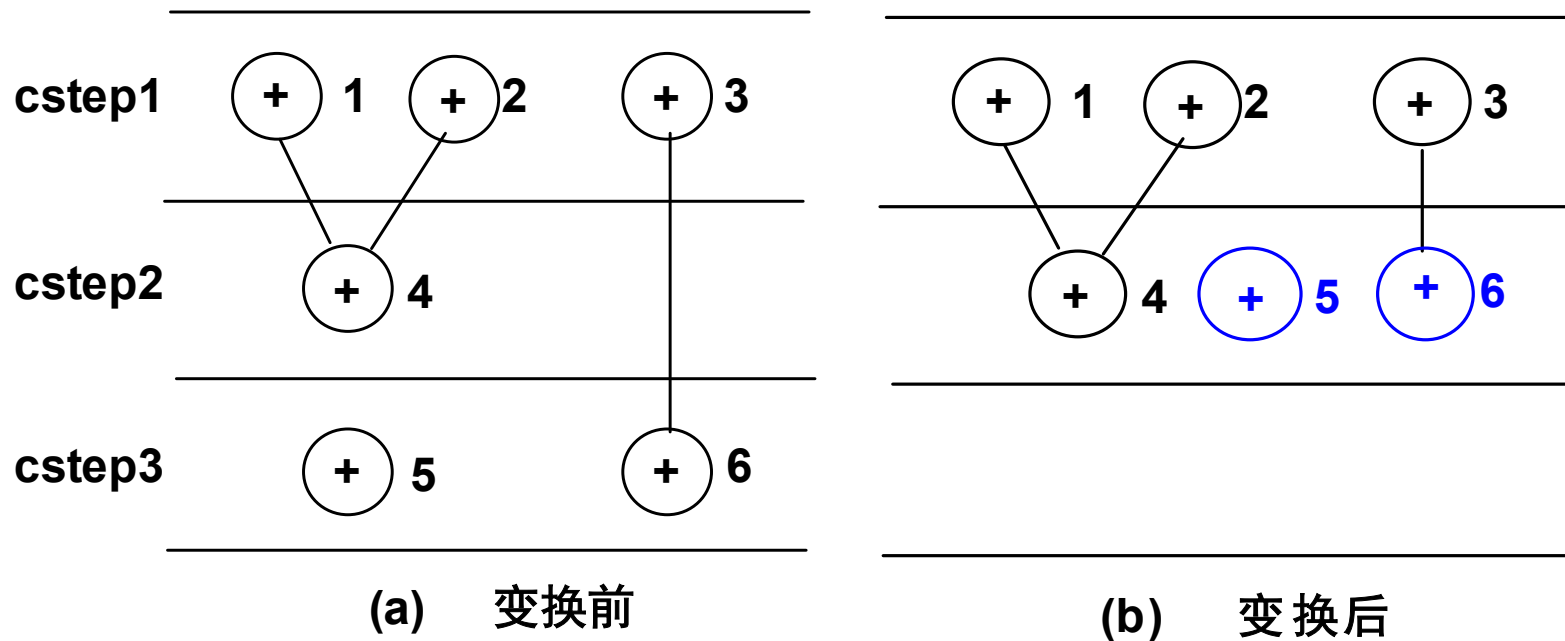


◆ 依据约束条件分类：

- 时间约束条件下的调度算法；
- 硬件资源约束条件下的调度算法；
- 时间约束与硬件资源约束条件下的调度算法；
- 无约束的调度算法。



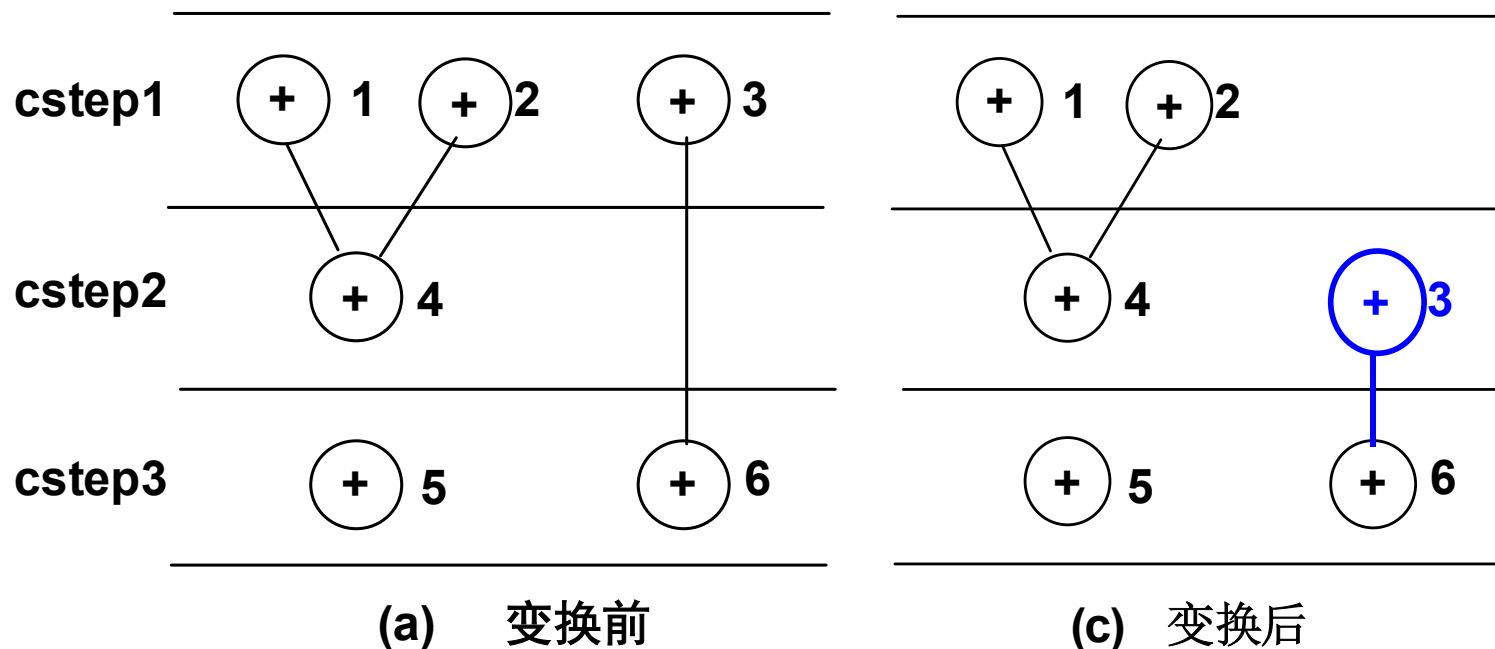
变换法举例



	加法器 (资源)	控制步 (周期)
变换前	3	3
变换后	3	2



变换法举例（续）



	加法器（资源）	控制步（周期）
变换前	3	3
变换后	2	3

3.3.2.3 ASAP调度算法与ALAP调度算法

- ◆ **ASAP调度算法 (as soon as possible):**
将所有操作赋予**最早可能**调度到的**控制步**。
- ◆ **ALAP调度算法 (as late as possible):**
将所有操作赋予**最迟可能**调度到的**控制步**。
- ◆ **ASAP与ALAP都属于无约束调度算法**，其调度结果尽管不能令人满意，但却给出了：
 - 可能的调度区间；
 - 资源的上限；
 - 控制步的下限。



针对某一操作 op_k 作ASAP调度

在不考虑硬件资源约束的条件下，操作 op_k 可以开始执行的最早时间称为该操作的最早控制步。用 $ASAP(op_k)$ 表示：

$$ASAP(op_k) = \begin{cases} \text{Max}_{Op_i \in OP_{pre}} \{ ASAP(op_i) + TD(op_i) \}, & \text{若 } OP_{pre} \neq \emptyset \\ 1, & \text{否则} \end{cases}$$

其中， OP_{pre} 是操作 op_k 的直接前趋集；

$$Op_i \in OP_{pre} ;$$

$TD(op_i)$ 是操作 op_i 的延迟时间，即实现操作 op_i 所需的控制步数，



针对某一操作 op_k 作ALAP调度

在不考虑硬件资源约束的条件下，操作 op_k 可以开始执行的最迟时间称为该操作的最迟控制步。用 $ALAP(op_k)$ 表示：

$$ALAP(op_k) = \begin{cases} \min_{op_i \in OP_{suc}} \{ ALAP(op_i) - TD(op_k) \}, & \text{若 } OP_{suc} \neq \emptyset \\ M_{step} - TD(op_k) + 1, & \text{否则} \end{cases}$$

其中， M_{step} 是所需的最大控制步数目；

OP_{suc} 是操作 op_k 的直接后继集； $op_i \in OP_{suc}$

$TD(op_k)$ 是 op_k 的延迟时间，即实现 op_k 所需的控制步数。



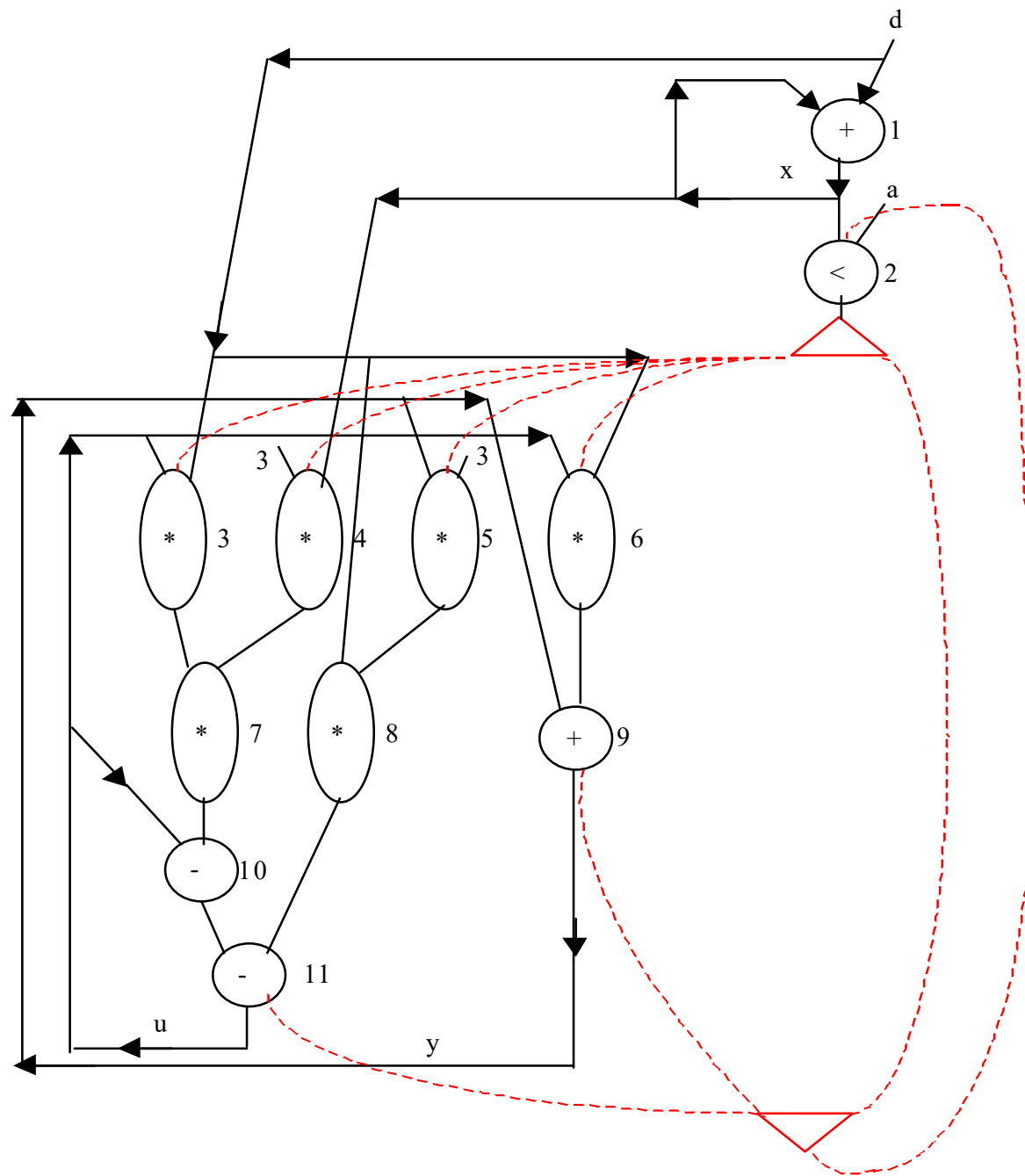
例子 (example) 的VHDL描述

```
process( xin, y in, uin )  
  
    variable  x, y, u :      integer;  
    variable  x1, y1, u1 :   integer;  
  
begin  
  
    x := xin;           y := y in;           u := uin;  
  
    while ( x < a ) loop  
        x1 := x + d;  
        y1 := y + u * d;  
        u1 := u - ( 3 * x * u * d ) - ( 3 * y * d );  
  
    end loop;  
  
    xout <= x;           yout <= y;  
  
end process;
```

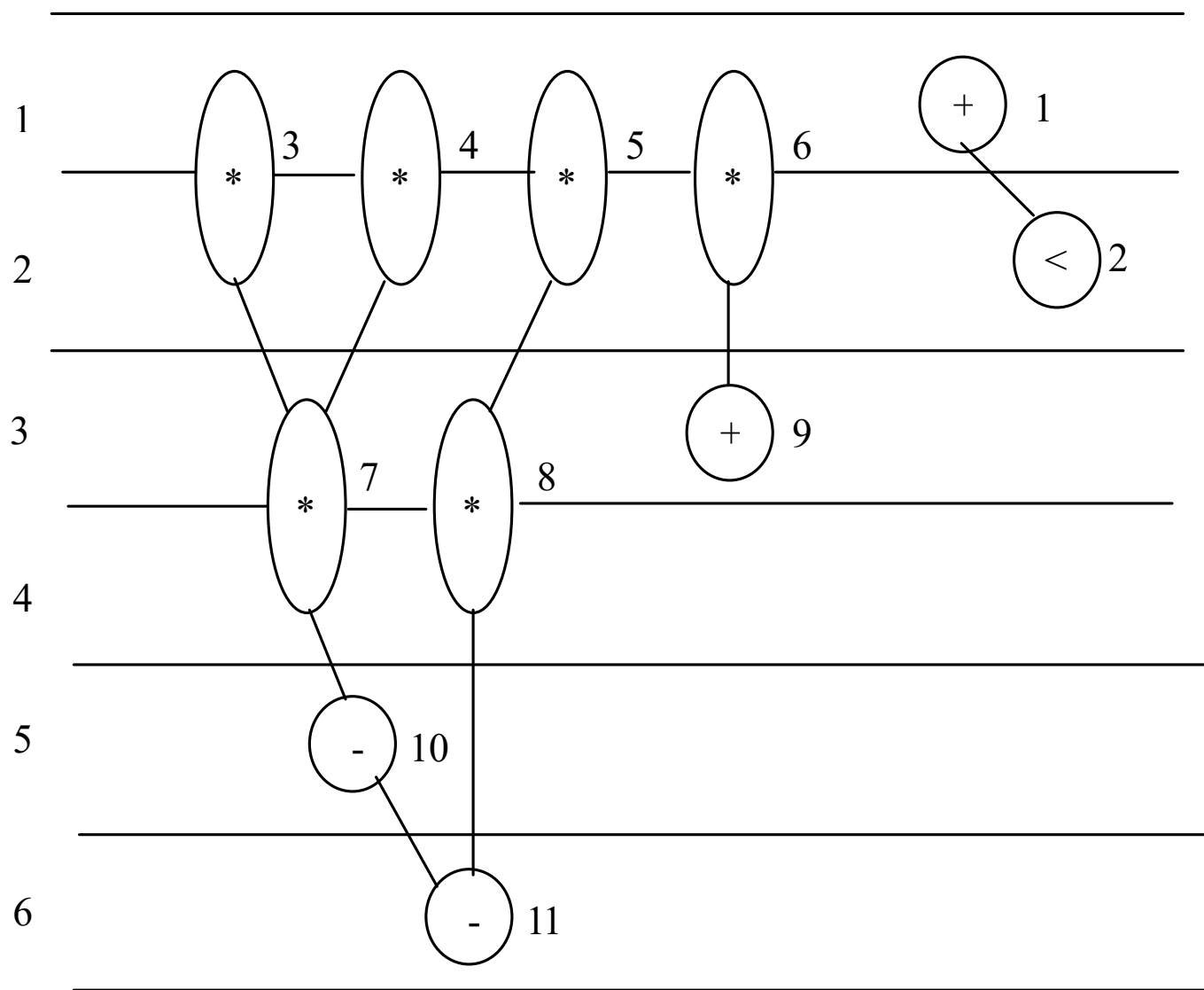




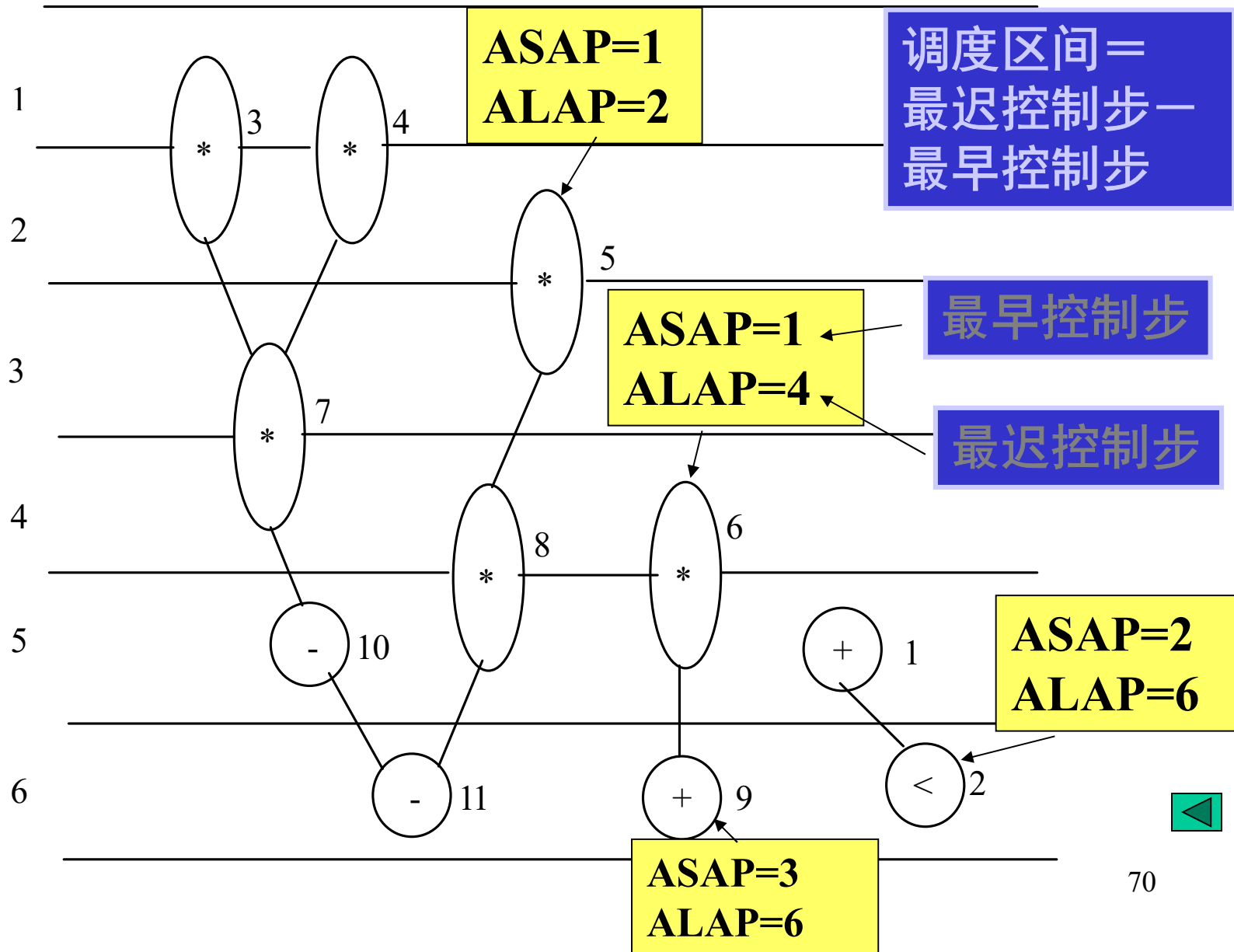
例子
(example)
的CDFG



例子 (example) 的 ASAP 调度结果



例子 (example) 的ALAP调度结果

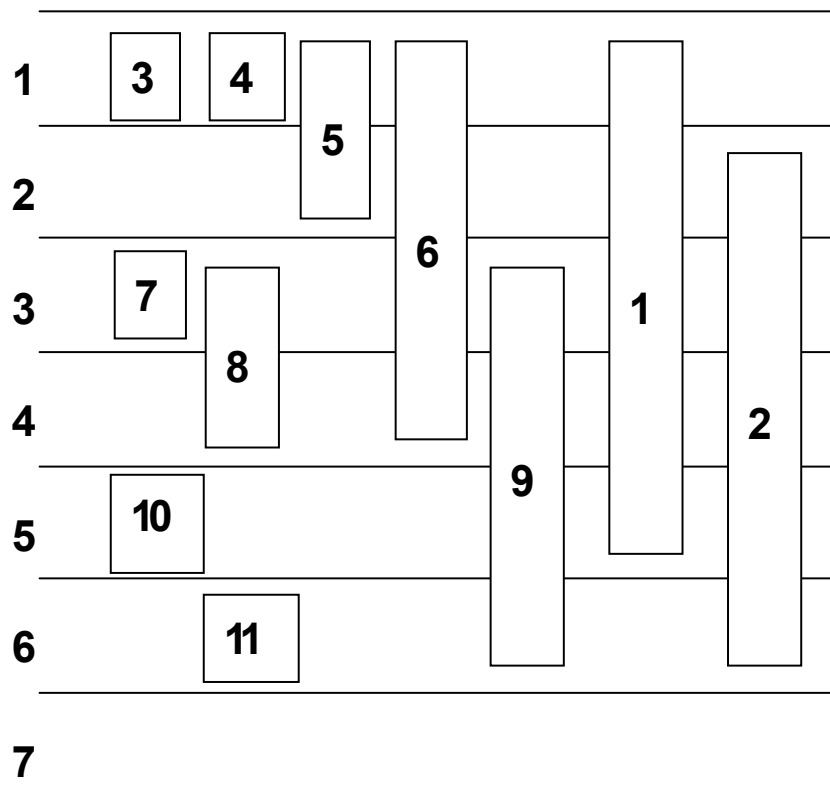




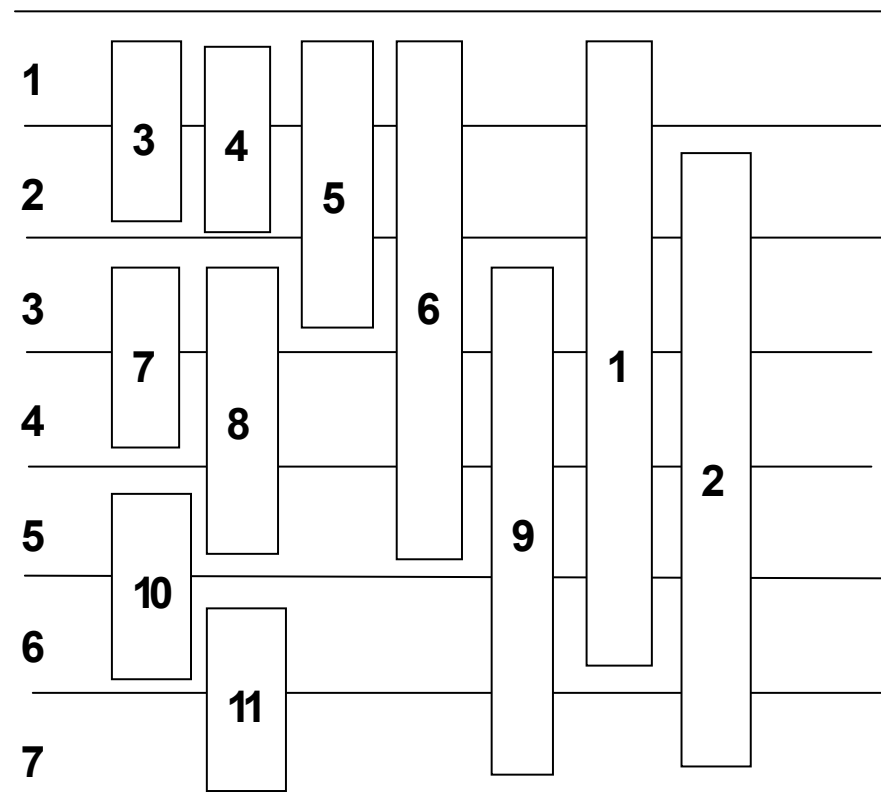
操作 op_k 的调度区间

◆ 操作 op_k 的调度区间:

$$SA(op_k) = [ASAP(op_k), ALAP(op_k)]$$



(a) 总控制步数为6



(b) 总控制步数为7



ASAP与ALAP

- ◆时间优化，未考虑面积约束
- ◆常作为构造初始解的算法，再进行优化



3.3.2.4 列表调度算法

列表调度 (list scheduling) 算法:

- (1) $\text{Current_Step} := 1$;
- Current_Step 代表当前控制步
- (2) 为当前控制步建立就绪队列, 并按优先级函数排序;
- (3) 在满足硬件资源约束的条件下, 将具有最高优先级的操作调度到当前控制步 (Current_Step);
- (4) 修改就绪操作队列;
- (5) 在满足硬件资源约束的条件下, 若还有操作可以被调度转 (3); 否则进入下一步;
- (6) $\text{Current_Step} := \text{Current_Step} + 1$;
- (7) 若还有操作尚未被调度, 转 (2);
- (8) 结束。

◆ 就绪:

- 就绪操作: 当前控制步可以执行的操作;
- 就绪队列: 就绪操作的集合;

◆ 优先级函数:

- 调度区间最小的操作;
- 处于最长路径上的操作。

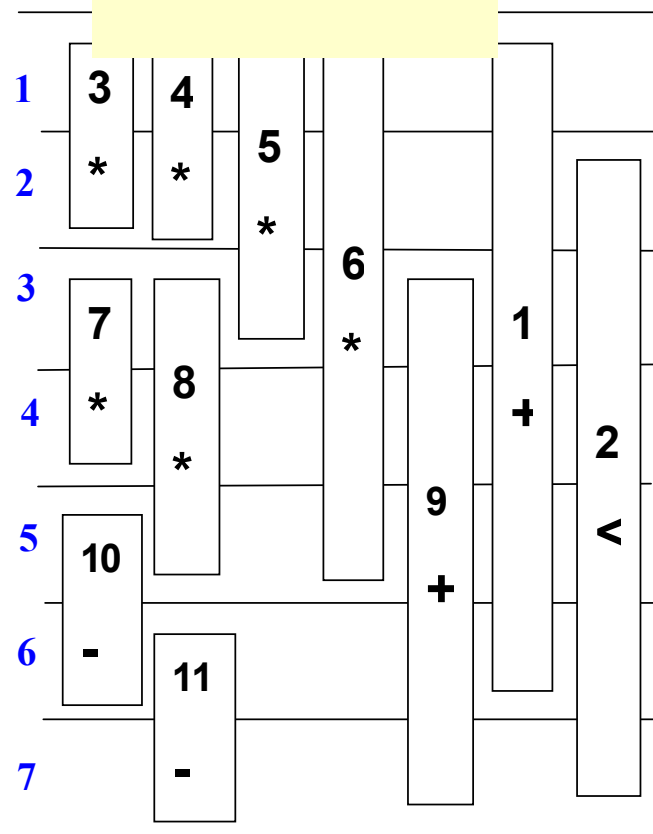


例子 (example) 的列表调度

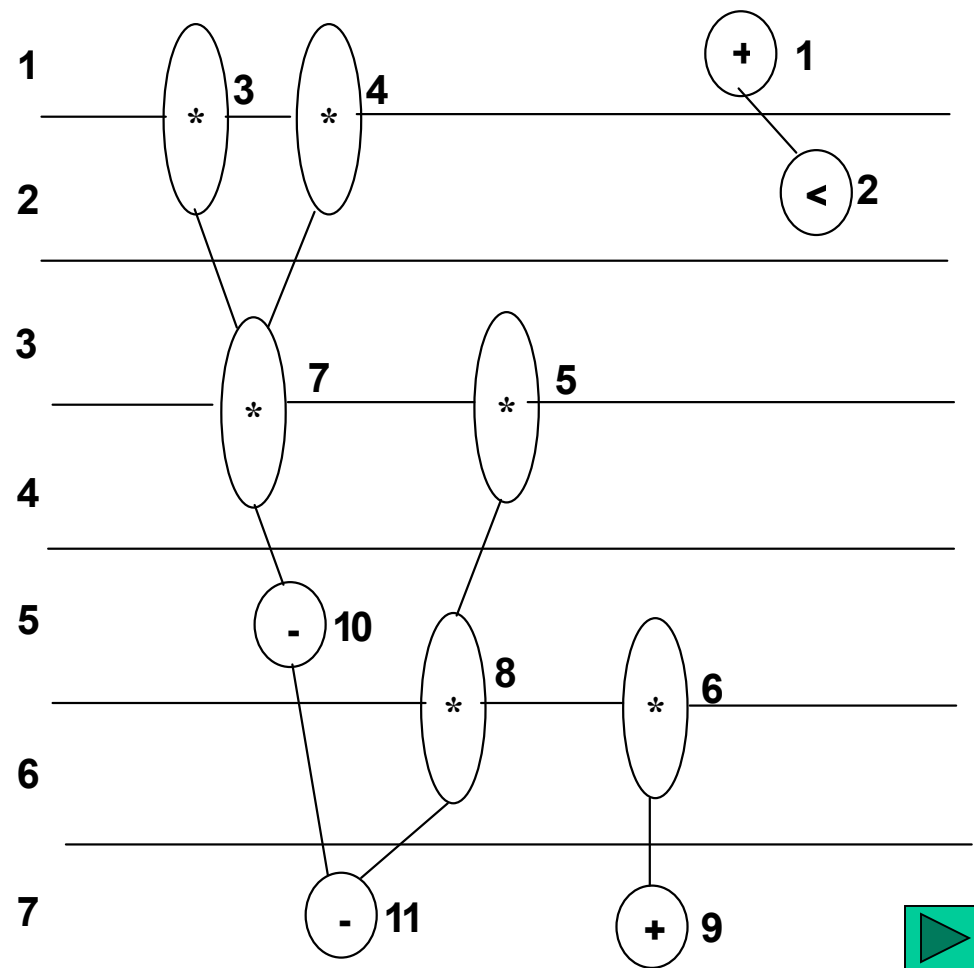
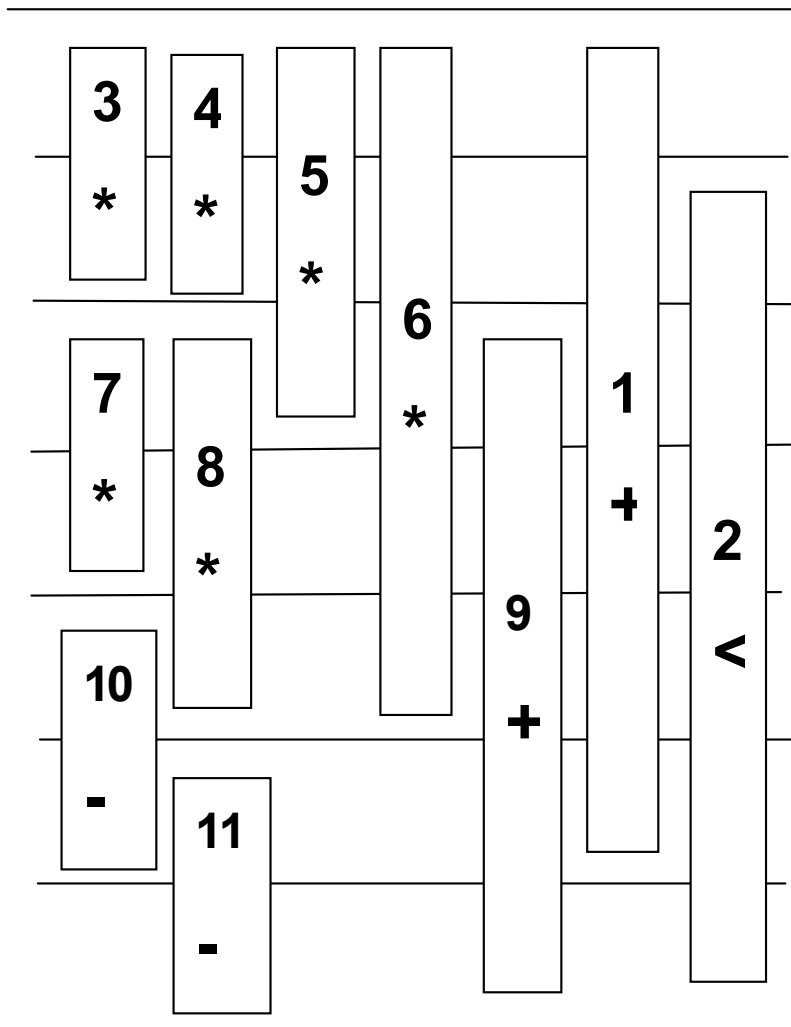
- ◆ 设硬件资源约束条件为：
 - 2个非流水线乘法器 (*)
 - 1个减法器 (-)
 - 1个加法和比较单元 (+, <)
- ◆ 优先级函数：
 - 调度区间
 - 路径长度
- ◆ 调度区间：总控制步数为7。
 - 调度过程见下页



调度区间：



例子 (example) 的列表调度 (过程)



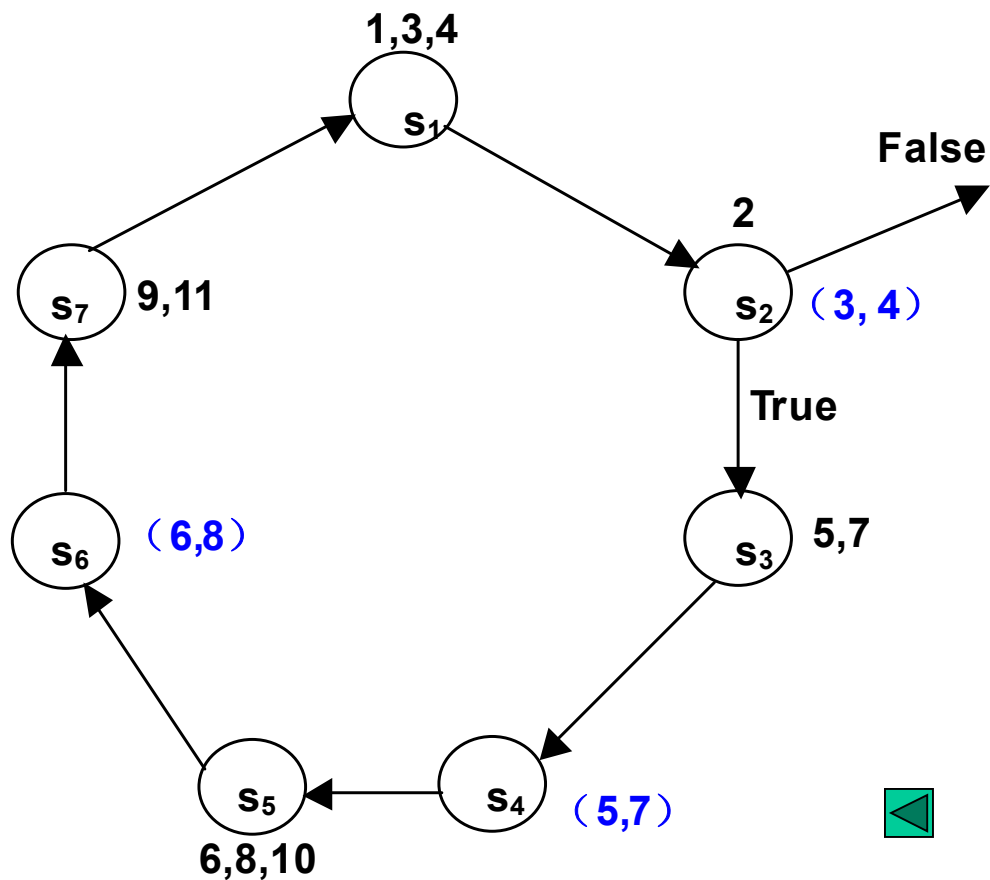


调度完成



控制器的FSM描述

- ◆ 调度完成后，即可得到操作调度的有限状态机（FSM）描述。
- ◆ FSM的每一个状态对应于一个控制步，在状态的旁边标记着该控制步所要执行的操作。
- ◆ 状态 s_4 和 s_6 是空状态，因为乘法操作是多周期操作，状态 s_4 和 s_6 下乘法操作正在继续执行。





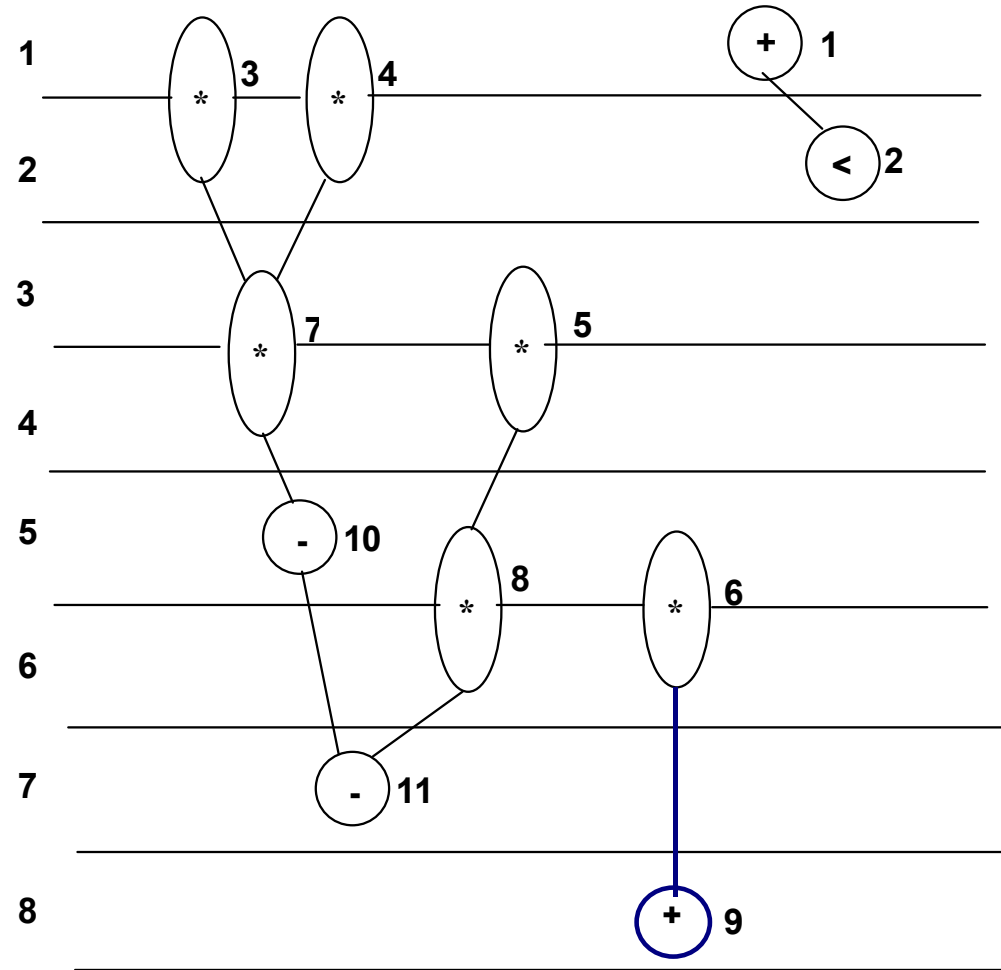
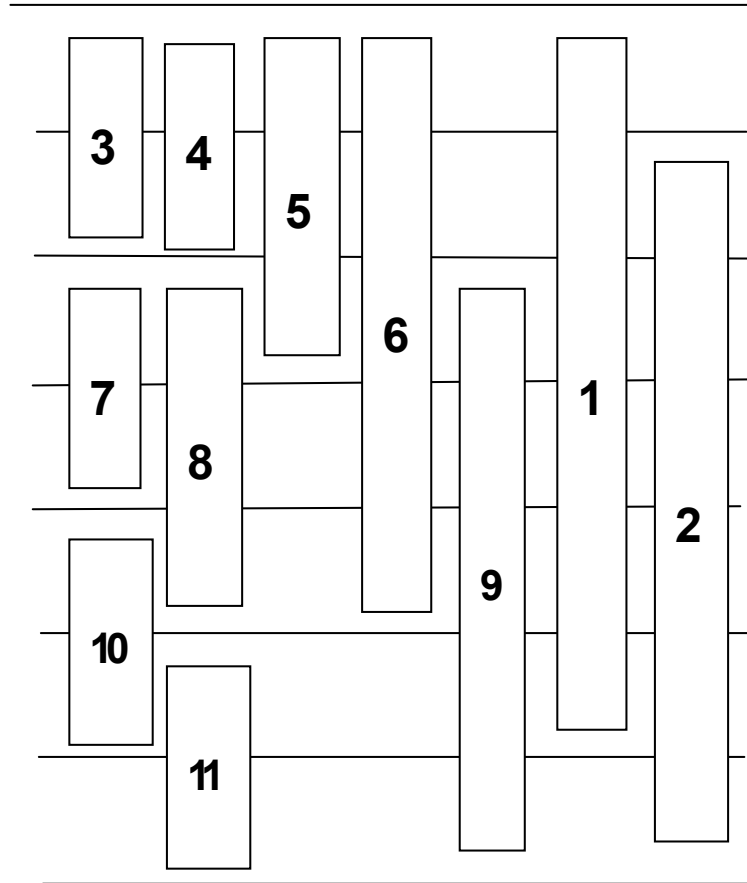
列表调度算法小结

- ◆ 属于构造性算法；
- ◆ （硬件）资源约束：
硬件资源约束改变，调度结果就可能改变。
- ◆ 就绪操作队列：即当前控制步中可以执行的操作集合；
- ◆ 优先级函数：
 - 调度区间小者；
 - 处于较长路径中者；
- ◆ 每安排一个操作，立即修改就绪队列。

- ◆ 举例：设例子（example）的约束条件改为：
 - 2 个乘法器（需要2个控制步）；
 - 1个运算单元（加、减、比较）；
 - 优先级函数不变。
- ◆ 调度结果：
由于硬件资源减少，控制步总数由7变为8。
(见下页)



资源约束改变后的调度结果





3.3.2.5 调度中控制结构的处理

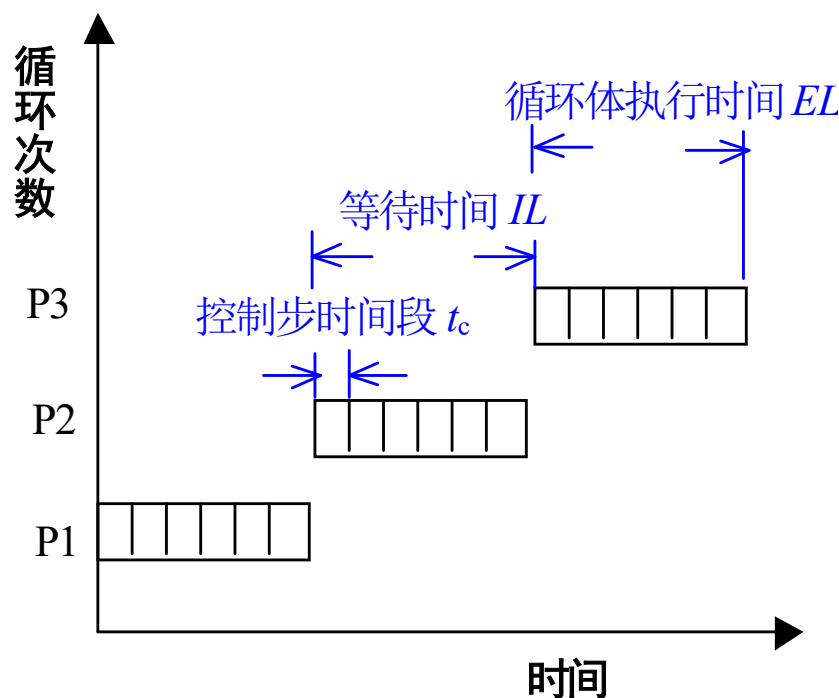
- ◆ 循环控制结构的处理;
- ◆ 分支控制结构的处理;



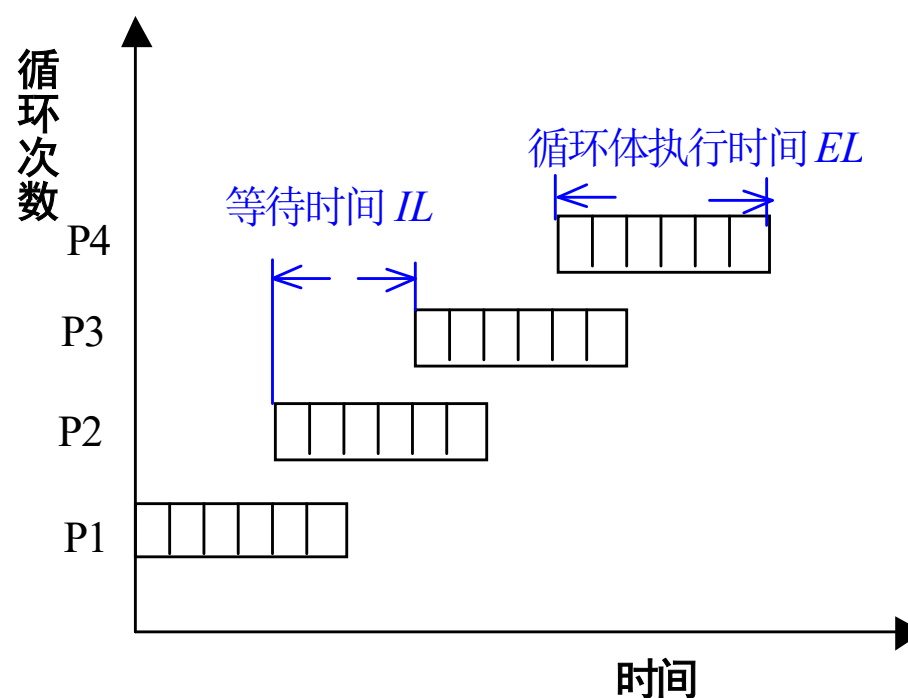
循环控制结构的处理

(流水线 / 非流水线方式)

当EL足够小，满足 $IL \geq EL$ 时，可以采用非流水线方式



(a) 非流水线设计方式



(b) 流水线设计方式

循环结构的两种处理方法



循环控制结构的处理（定义）

- ◆ 循环结构常见于数字信号处理（Digital Signal Processing, DSP）算法中；
- ◆ 设 f_c 为外界采样频率， t_c 为控制步时间段长度（硬件能力决定）；
- ◆ 为使采样数据不致丢失，每隔 n_c 个控制步系统必须接受一组新的采样数据，并加以处理。

$$n_c \leq \frac{1}{f_c \times t_c} \quad \text{即} \quad n_c \times t_c \leq \frac{1}{f_c}$$

- ◆ 外界对数据处理的要求由 f_c 明确表示，当控制步时间段长度 t_c 确定后，控制步数 n_c 就被确定。
- ◆ 循环体的执行时间EL（execution length）：（单位：控制步数）EL由硬件能力决定。
- ◆ 循环体的数据等待时间IL（input data latency）：两组采样数据处理过程起始点时间之差（单位：控制步数）
 - IL是调度过程中可以选择的一个参数，尽量选择 $IL = n_c$ 。



非流水线方式

- ◆ 非流水线方式: $n_c = IL = EL$
- ◆ 非流水线方式, 操作的调度相对简单:
 - 每一时刻只有一个控制步中的操作在执行, 只需考虑一个循环体中的操作调度。
 - 操作间不存在跨过程 (循环体) 的数据引用,



流水线方式

◆ 流水线方式：

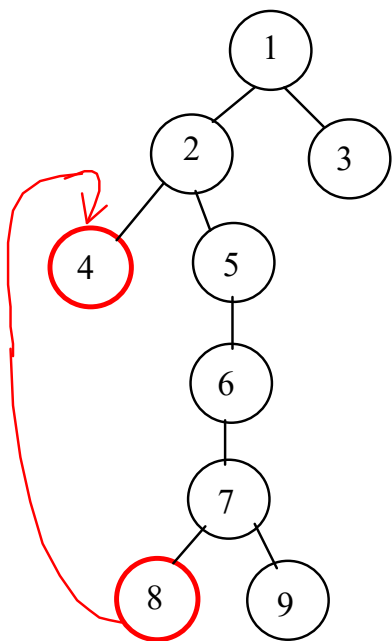
$$n_c = IL \leq EL$$

◆ 流水线方式： 折叠

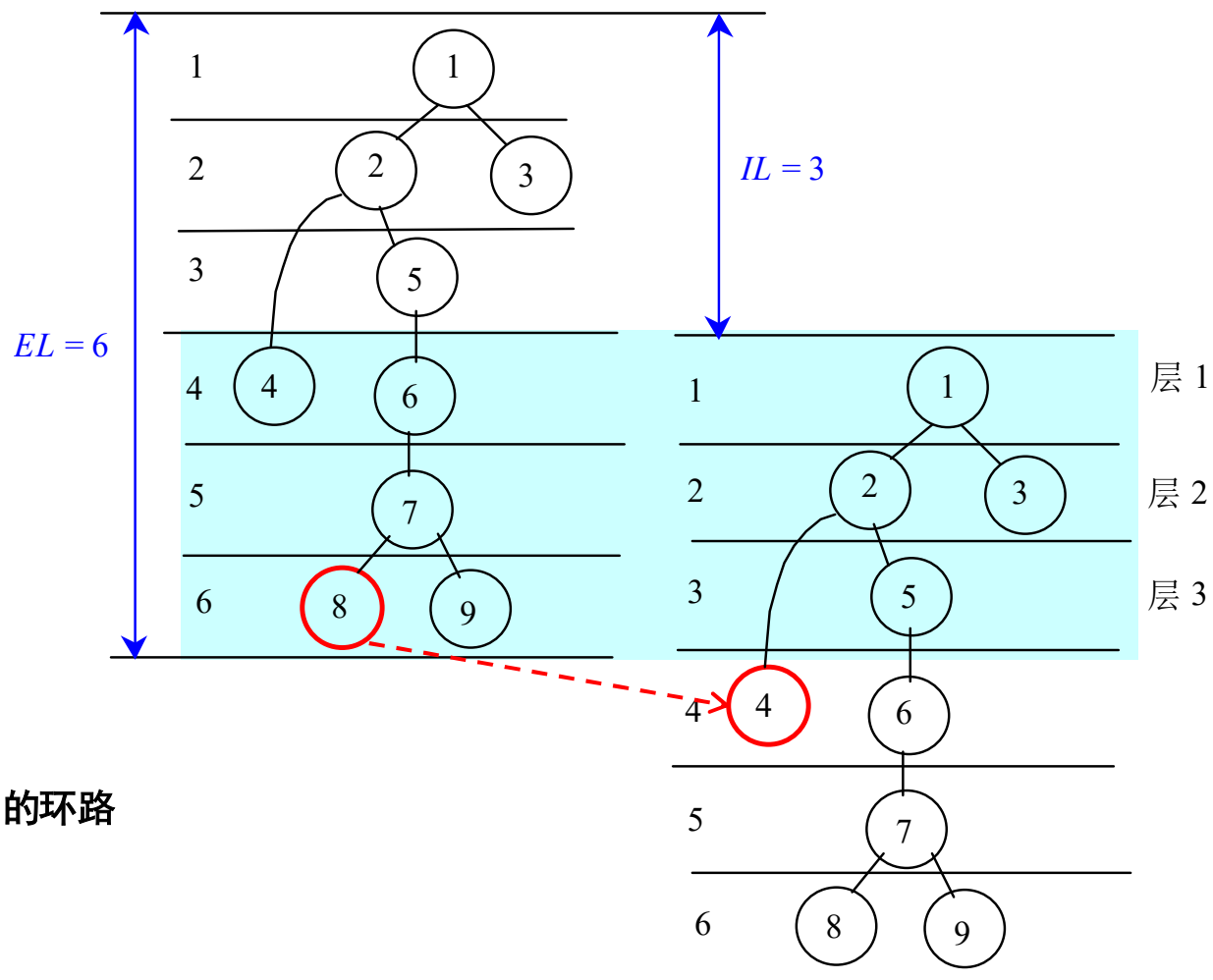
- 在同一控制步中，多个操作会同时出现，需统一调度；
- 控制环路切断，必须保证跨过程数据引用的正确性。



流水线方式下的循环折叠处理



(a) 控制数据流图中的环路



(b) 流水线设计方式下的循环折叠



分支控制结构的处理

◆ 操作的合并:

- 分支控制结构处理的**目的**: 将不同条件分支中的**操作合并**, 尽可能**共享功能单元**。
- **互斥操作**: 不可能在同一条件下执行的操作。**互斥操作可以合并**。
- **互斥操作举例**:

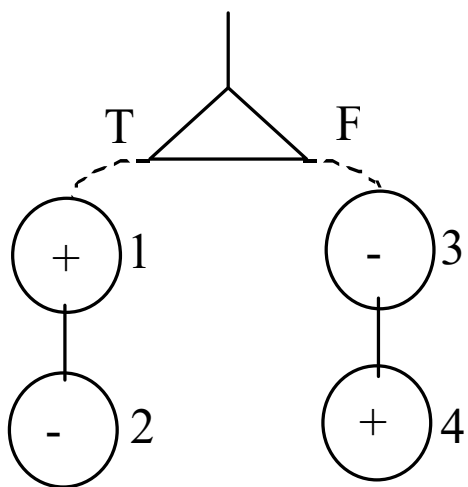
◆ 输出变量的合并:

- 2个仅被条件分支内操作引用的变量可以合并

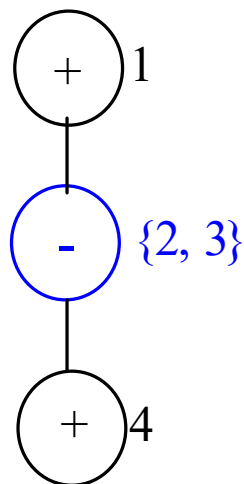


互斥操作合并带来的影响

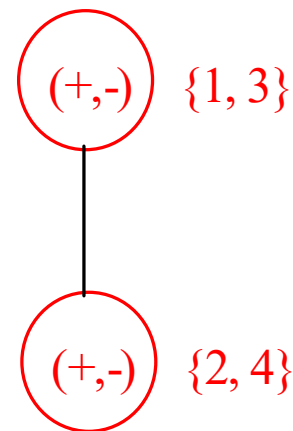
- ◆ 利益：共享功能单元，有可能降低造价。
- ◆ 可能带来损失：路径长度可能增加。
- ◆ 对于某一特定问题而言，带来的究竟是利还是弊要做具体分析。



(a) 未经处理的控制数据流图



(b) 合并方案1



(c) 合并方案2



分支控制结构的处理方法

- ◆ 局部合并法：

从带有嵌套分支结构的最内层开始，逐层向外进行分支操作的合并处理。

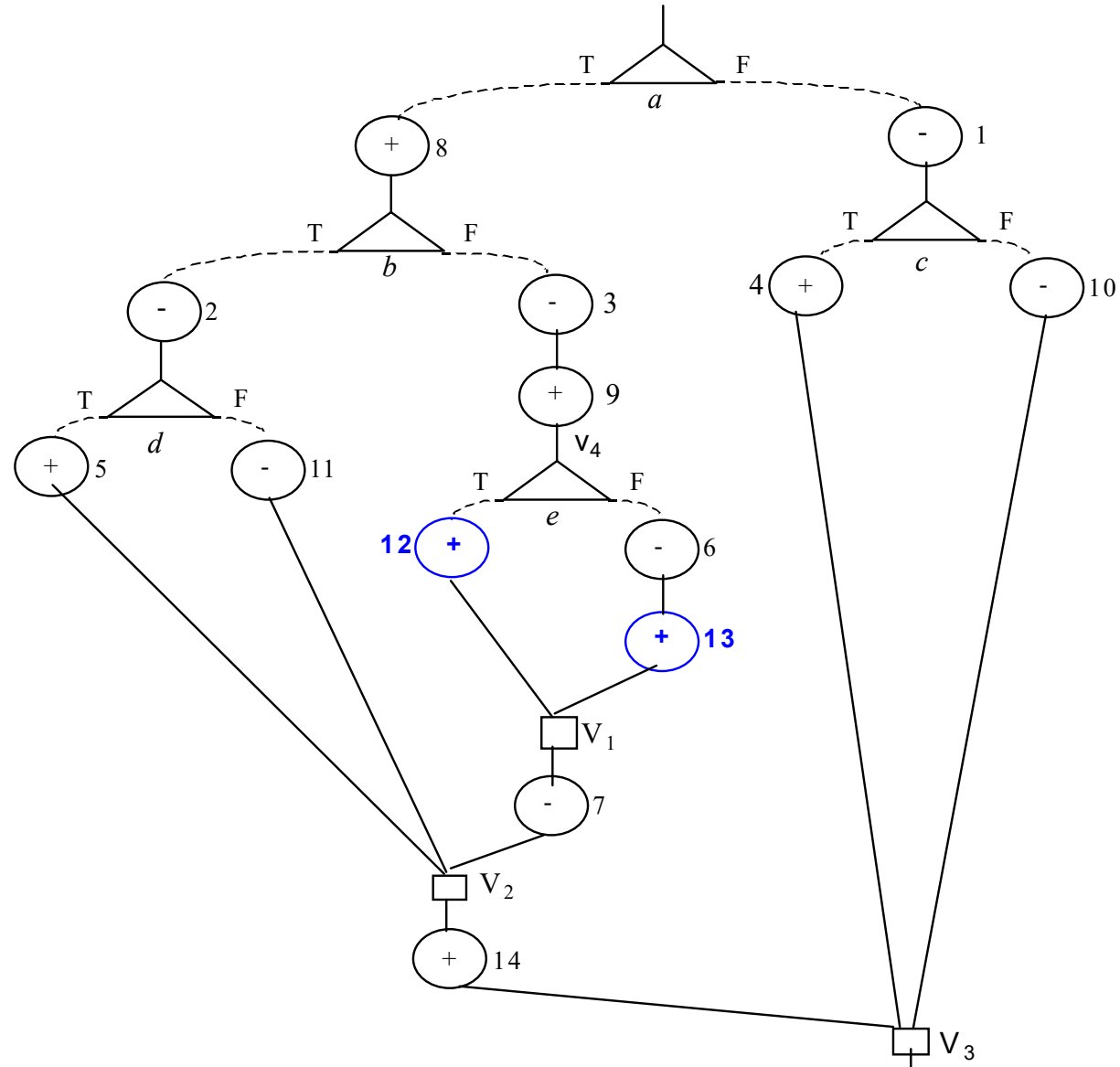
(从内向外，逐步进行)

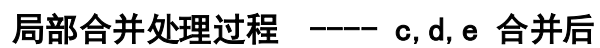
- ◆ 全局合并法：

统一考虑各嵌套层次条件分支中互斥操作的合并，以便这些互斥操作共享功能单元。



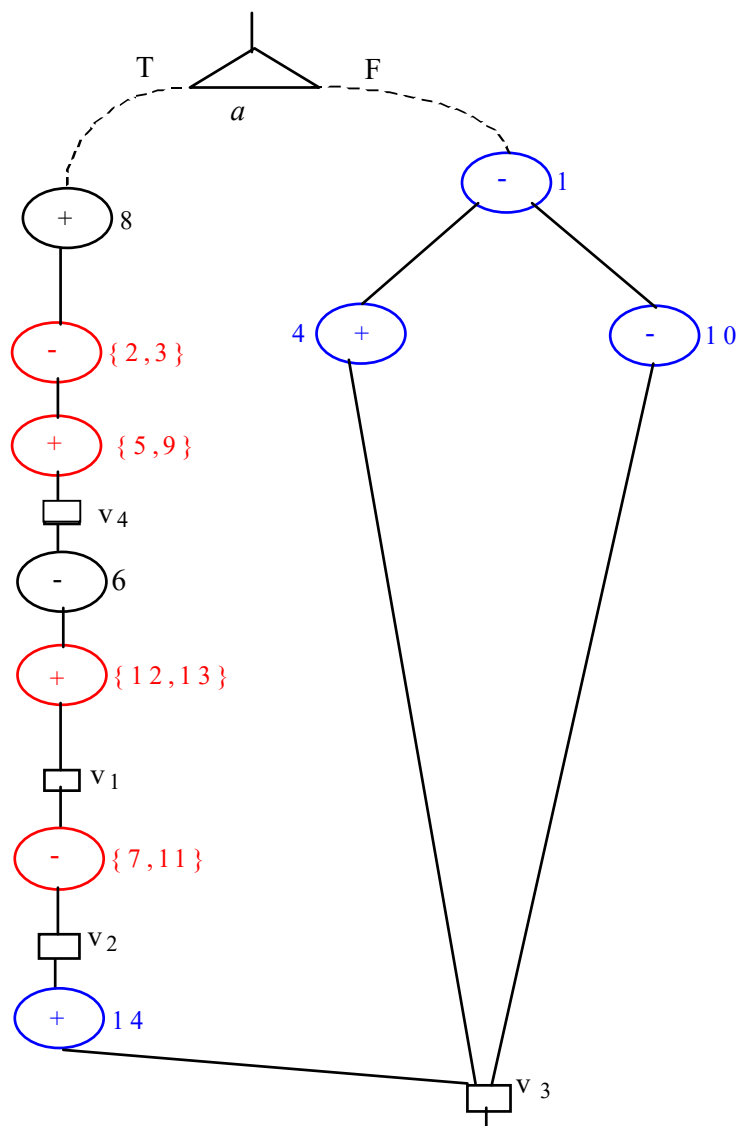
局部合并法（从内向外，逐步进行）



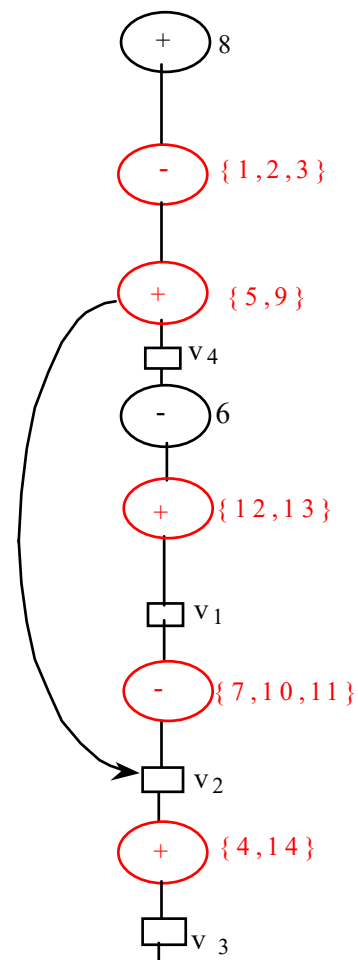




局部合并法（从内向外，逐步进行）



(b) b 合并后



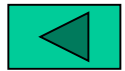
(c) a 合并后

局部合并处理过程



全局合并法

- ◆ 统一考虑各嵌套层次条件分支中互斥操作的合并，以便这些互斥操作共享功能单元。
- ◆ 互斥的判断： 条件向量的点积
- ◆ 条件向量：



- 维数：分支节点数，每个节点对应于条件向量的一个元素。
- 条件向量的值：
 - '0': 表示操作在该分支条件为假时执行；
 - '1': 表示操作在该分支条件为真时执行；
 - 'X': 表示操作的执行与该分支条件无关。



条件向量的点积

◆ 运算规则：

(1) 如果结果中出现 q 时，则

$$cv = cv_1 \cdot cv_2$$

其中 $cv_1^i \cdot cv_2^i = q$ 时 $cv^i = X$

$cv_1^i \cdot cv_2^i \neq q$ 时 $cv^i = cv_1^i \cdot cv_2^i$

(2) 不满足条件 (1) 时：

(条件向量点积的结果为空)

◆ 两个条件向量的点积运算结果若为空，表明这两个操作不是互斥操作；否则，他们是互斥操作，可以进行合并处理，点积运算结果就是合并所得操作的条件向量。

条件向量点积表

cv_1^i cv^i cv_2^i				
		0	1	X
0		0	q	X
1		q	1	X
X		X	X	X



点积应用举例

		<i>i</i>				
		1	2	3	4	5
•)	<i>cv</i>					
	<i>cv</i> ₁₂	1	0	X	X	1
	<i>cv</i> ₁₃	1	0	X	X	0
		1	0	X	X	q

➤ 结果中有 **q**，满足条件 (1)，用 **X** 置换 **q**，得：

$$cv = cv_{12} \cdot cv_{13} = \{1, 0, X, X, X\}$$

➤ 运算结果表明，操作 **12** 与操作 **13** 是互斥操作，
其合并后的操作 {12, 13} 的条件向量为
{ 1, 0, X, X, X }。



点积应用举例

		i				
		1	2	3	4	5
•.)	cv_{13}	1	0	X	X	0
	cv_{14}	1	X	X	X	X
		1	X	X	X	X

➤ 结果中没有 q ，满足条件 (2)，得：

$$cv = cv_{13} \cdot cv_{14} = \phi$$

➤ 运算结果表明，操作 13 与操作 14 不是互斥操作，不能进行合并。



点积应用举例

		<i>i</i>				
		<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>
<i>cv</i>		1	2	3	4	5
	<i>cv</i> ₅	1	1	X	1	X
•.)	<i>cv</i> ₁₂	1	0	X	X	1
		1	q	X	X	X

结果中有 **q**，满足条件 (1)，用 **X** 置换 **q**，得：

$$cv = cv_5 \cdot cv_{12} = \{1, X, X, X, X\}$$

- 运算结果表明，操作 5 与操作 12 是互斥操作，其合并后的操作 {5, 12} 的条件向量为 {1, X, X, X, X}。
- 从合并后的条件向量看出，操作 {5, 12} 只在条件 **a** 的分枝上，已从条件 **d** 及条件 **e** 的分枝上移出。— 全局合并!



条件向量的叉积运算

- ◆ 对条件向量的点积运算稍加修改，全局合并法即可简化为局部合并法。点积运算经修改后成为叉积运算；
- ◆ 条件向量 cv_1 与 cv_2 的叉积运算规则由叉积表和下列规则定义：

条件向量叉积表

cv_1^i cv^i cv_2^i		0	1	X
0		0	q	m
1		q	1	m
X		m	m	X



叉积运算规则

(1) 如果 $cv_1^i \times cv_2^i$ ($i=1,2,\dots,n$) 的结果中没有 **m**,

有且仅有一个 **q** 时, 则

$$cv = cv_1 \times cv_2$$

$$\text{其中 } cv_1^i \times cv_2^i = q \text{ 时 } cv^i = X$$

$$cv_1^i \times cv_2^i \neq q \text{ 时 } cv^i = cv_1^i \times cv_2^i$$

(2) 不满足条件 (1) 时:

$$cv = cv_1 \times cv_2 = \phi \quad (\text{条件向量叉积的结果为空})$$

➤ 两个条件向量的叉积运算中若出现 **m**, 表明相应的两个操作不在同一层次的条件分枝中。



叉积应用举例

		<i>i</i>				
		1	2	3	4	5
×)	<i>cv</i>					
	cv_{12}	1	0	X	X	1
	cv_{13}	1	0	X	X	0
		1	0	X	X	q

- 结果中没有 **m** 且只有 1 个 **q**，满足条件 (1)，用 **X** 置换 **q**，得：

$$cv = cv_{12} \times cv_{13} = \{1, 0, X, X, X\}$$

- 运算结果表明，操作 12 与操作 13 是同一层次上的互斥操作，其合并后的操作{12, 13}的条件向量为{ 1, 0, X, X, X}。



叉积应用举例

		<i>i</i>				
		<i>cv</i>	1	2	3	4
×)	<i>cv</i> ₁₃	1	0	X	X	0
	<i>cv</i> ₁₄	1	X	X	X	X
		1	m	X	X	m

➤ 结果中有 m，满足条件 (2)，得：

$$cv = cv_{13} \times cv_{14} = \phi$$

➤ 操作 13 与操作 14 不是互斥操作，不能进行合并。



叉积应用举例

$i \backslash cv$						
		1	2	3	4	5
×)	cv_2	1	1	X	X	X
	cv_{10}	0	X	0	X	X
		q	m	m	X	X

➤ 结果中有 m，满足条件 (2)，得：

$$cv = cv_2 \times cv_{10} = \phi$$

➤ 运算结果表明，操作 2 与操作 10 是不同层次上的互斥操作，不能进行合并。



3.3.2.6 调度中的功能单元库

◆ 功能单元库作用

- 为调度程序提供功能单元、操作及相互关系等信息
- 调度程序可以在调度时尽可能考虑调度方案对分配的影响。

◆ 功能单元库信息：

- 可以是半导体厂商提供的信息，
- 可以是已经设计好并经过验证的功能单元信息，
- 可以是功能单元的设计目标和约束集合。

◆ 功能单元本身的信息：

- 接口信息：输入、输出、控制端等。
- 物理参数：面积、时延、功耗等。
- 所实现的操作类型集：所能实现的操作类型组成的集合。
- 实现各个操作的控制码。
- 实现操作的方式：例如流水线方式、非流水线方式等。

◆ 功能单元可实现的操作：

- 操作的类型：一元、二元、算术或逻辑运算等。
- 操作的性质：操作的可交换性、是否符合结合律及分配律等。
- 能实现该操作的功能单元。



功能单元库举例

功能单元部分	名 称	符 号	输入个数	输出个数	面 积	时 延	实现的操作
	加法器	add	2	1	60	25	+
	减法器	sub	2	1	60	25	-
	ALU	ALU	2	1	90	25	+, -
操作部分	名 称	符号	输入个数	输出个数	结合律	操作数交换	可用功能单元
	加 法	+	2	1	Y	Y	add, ALU
	减 法	-	2	1	Y	N	sub, ALU



3.3.3 分配（allocation）技术

- ◆ 分配的目标是实现调度的结果，
 - 调度结果为分配提供了以下3类实体：
 - 操作；
 - 变量；
 - 数据传输。
 - 调度完成后，由分配完成硬件实现。
- ◆ 调度时必须考虑硬件资源约束。硬件资源的最终决定是在分配之后而不是在分配之前，因此，调度时的硬件资源约束条件只能是用户给定的或调度算法产生的一种估计值。
- ◆ 调度和分配相互依赖



分配技术

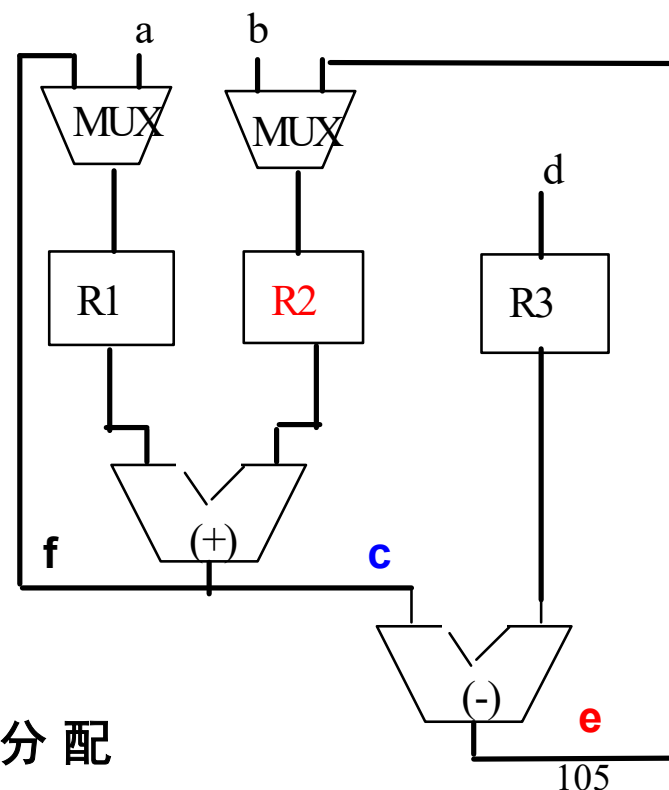
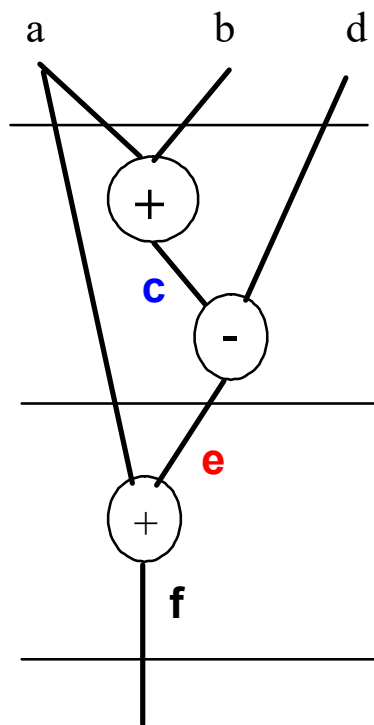
- ◆ 分配（allocation）的任务：
 - 把操作赋给相应的功能单元进行运算；
 - 把变量赋给相应的（寄存器、内存单元）加以存放；
 - 把数据传输通路赋给相应的（多路器、总线...）进行数据传输。
- ◆ 分配的目的：共享硬件模块。
- ◆ 模块确定：如果实现某一指定操作类型的功能单元有2种以上，就需要指定其中的1个来实现该类操作。



分配技术 (变量分配)

- ◆ 变量 c 没有跨越控制步的边界，不需要存放到寄存器中，而直接由加法器的输出端口传输到减法器的输入端口。
- ◆ 变量 e 跨越控制步的边界，需要存放到寄存器 (R2) 中。

控制步	操作
1	$c := a + b;$ $e := c - d;$
2	$f := a + e;$



对调度结果进行变量分配



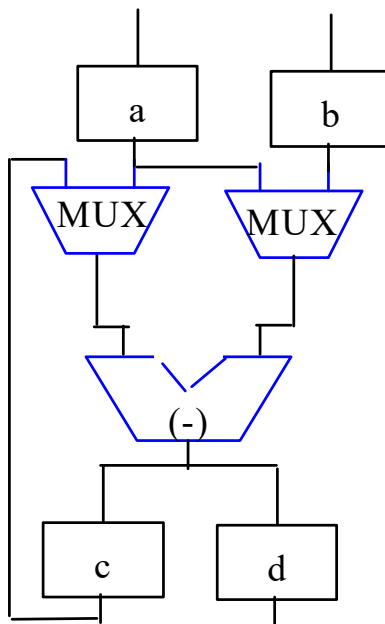
分配技术（操作分配）

◆ 同一调度结果的 2 个不同分配方案：

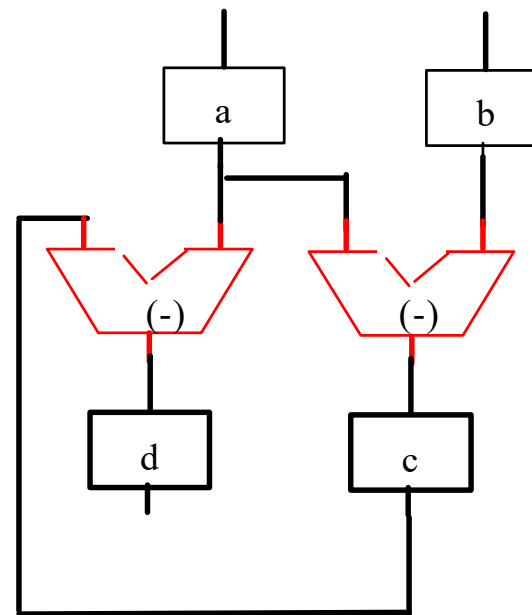
	减法器数量	多路器数量
方案(b)	1	2
方案(c)	2	0

控制步	操作
1	$c := a - b ;$
2	$d := c - a ;$

(a) 调度结果



(b) 一个减法器的实现



(c) 两个减法器的实现



分配算法

- ◆ 启发式算法：贪婪算法， 模拟退火算法...;
- ◆ 线性规划方法（Linear Programming）：
- ◆ 图论算法：团划分算法， 图着色算法....



基于图着色的分配算法

- ◆ 调度结果 → 冲突图 → 图着色算法求解分配问题；
每次只进行一种行为实体（例如操作）的分配。
- ◆ 冲突图（conflict graph）由二元式组成：
$$G = \{ V, E \}$$
 - 结点代表操作（或变量）；
 - 边代表操作（或变量）之间存在冲突（不能共享硬件资源）；
- ◆ 目的： 功能单元（寄存器）个数尽量少
→ 颜色数目尽量少



图着色算法

- ◆ 图着色算法是对冲突图中所有结点进行着色，任何冲突结点不得具有相同颜色。

将图着色算法用于分配问题，则是不同颜色的结点不得共享硬件资源。

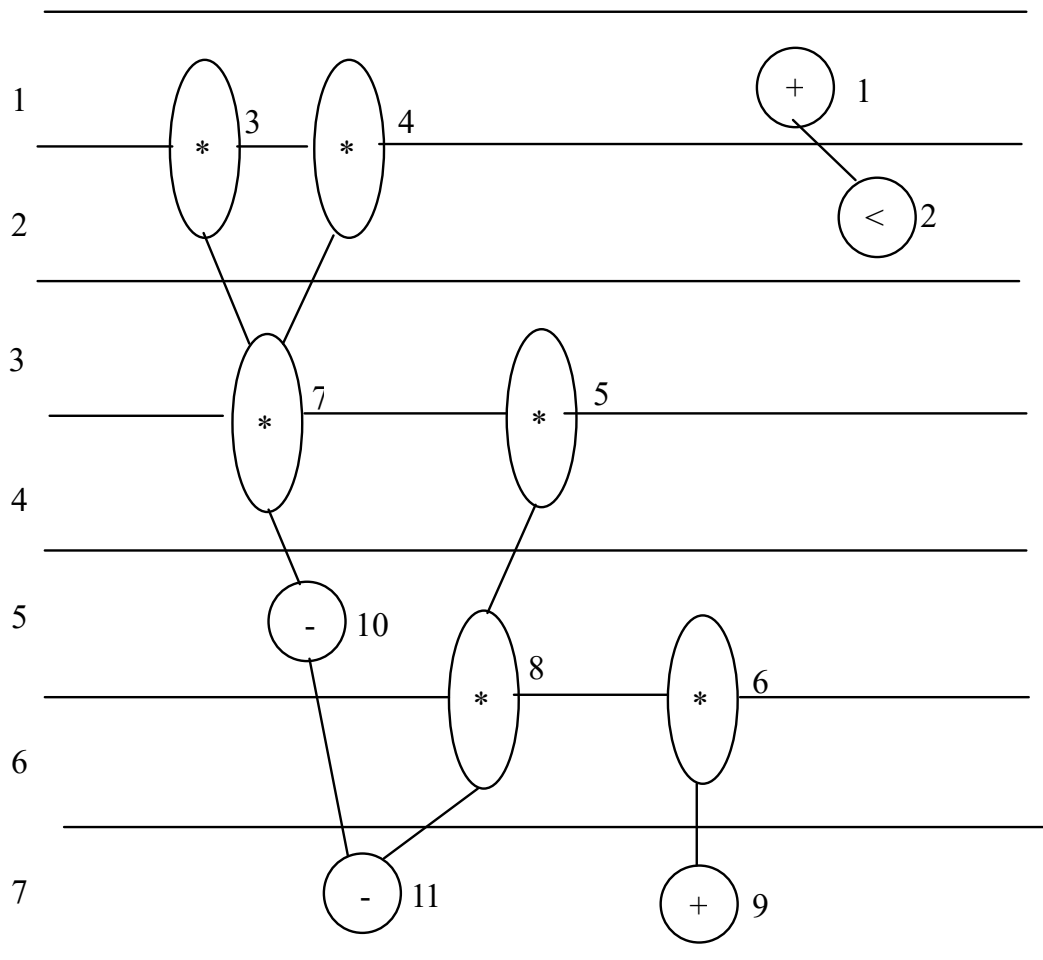
- ◆ 图着色算法的过程如下：

- (1) 在冲突图中任意选取一个尚未着色的结点；
- (2) 对被选取的结点着以与其相邻（有边和其相连）已着色结点不同的颜色；
- (3) 重复上述步骤，直至所有结点均已着色。

- ◆ 上述图着色算法是最简单明了的算法描述，实际上是基于冲突图的结群，使用贪婪算法进行结群，其着色效果不一定最优（颜色最少）。还可以改进！

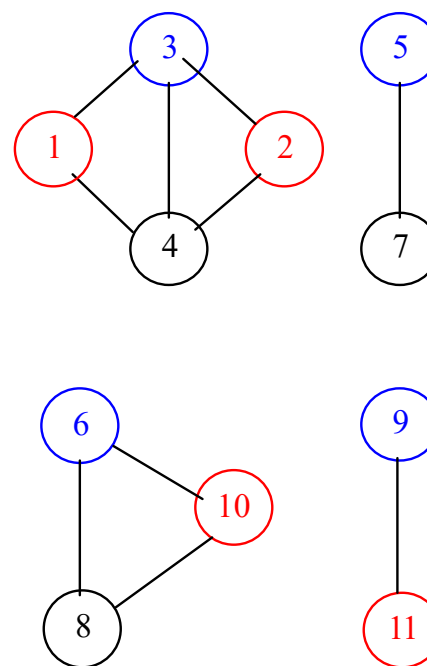


图着色算法举例



(a) 调度结果
(不管调度时是否考虑资源, 分配时未考虑资源约束)

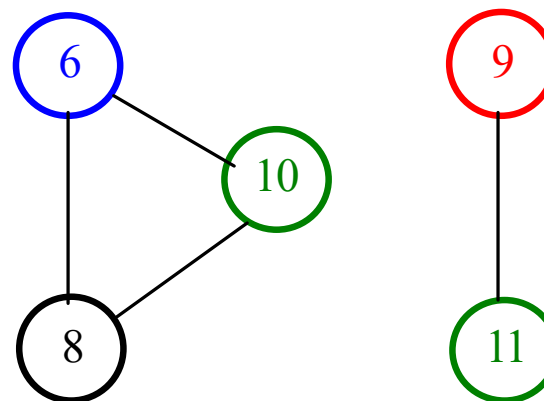
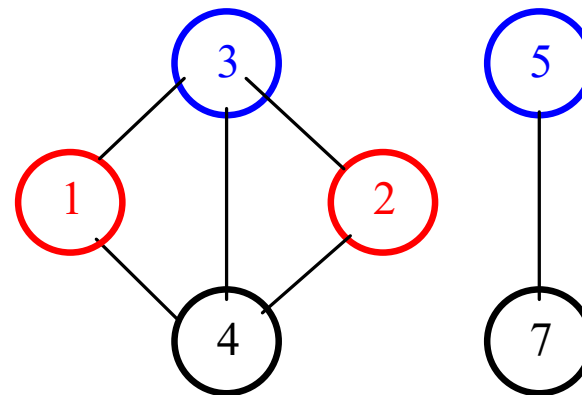
{1, 2, 10, 11}
{3, 5, 6, 9}
{4, 7, 8}



(b) 与调度结果
对应的冲突图

硬件资源约束条件下的图着色分配算法

- ◆ 假定上述例子中的硬件资源约束条件为：
 - 2个乘法器；
 - 1个减法器
 - 1个实现{ +, < }的功能单元。
- ◆ 分配结果为：
 - { 1, 2, 9 }
 - { 3, 5, 6 }
 - { 4, 7, 8 }
 - { 10, 11 }



硬件约束条件下的分配方案



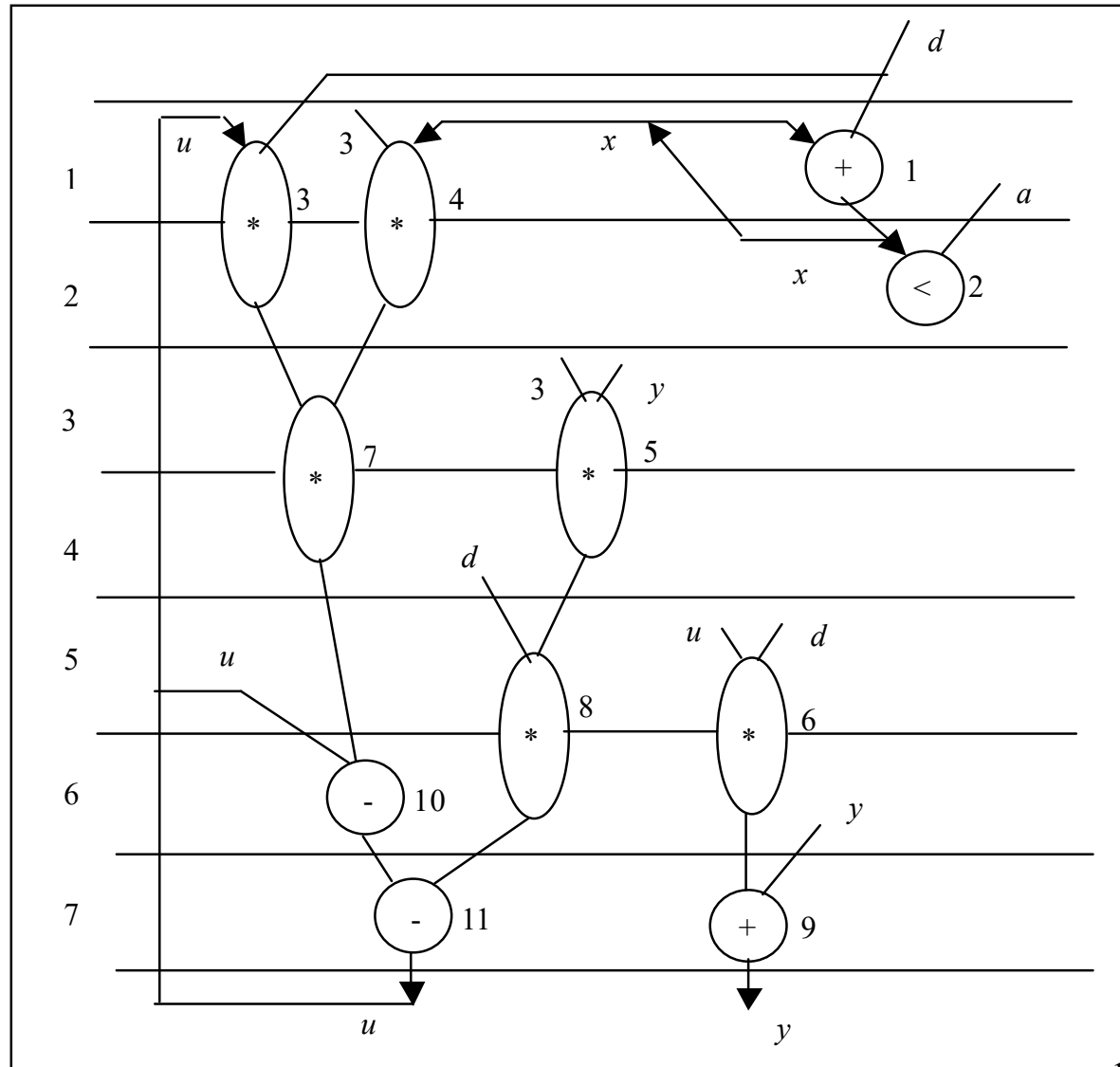
图着色算法用于变量分配

- ◆ 调度结果 → 变量生存期（表 / 图）
 - （变量冲突图）
 - 变量分配到寄存器（生存期冲突的变量不可共享寄存器）



变量分配的实例

调度
的结果：



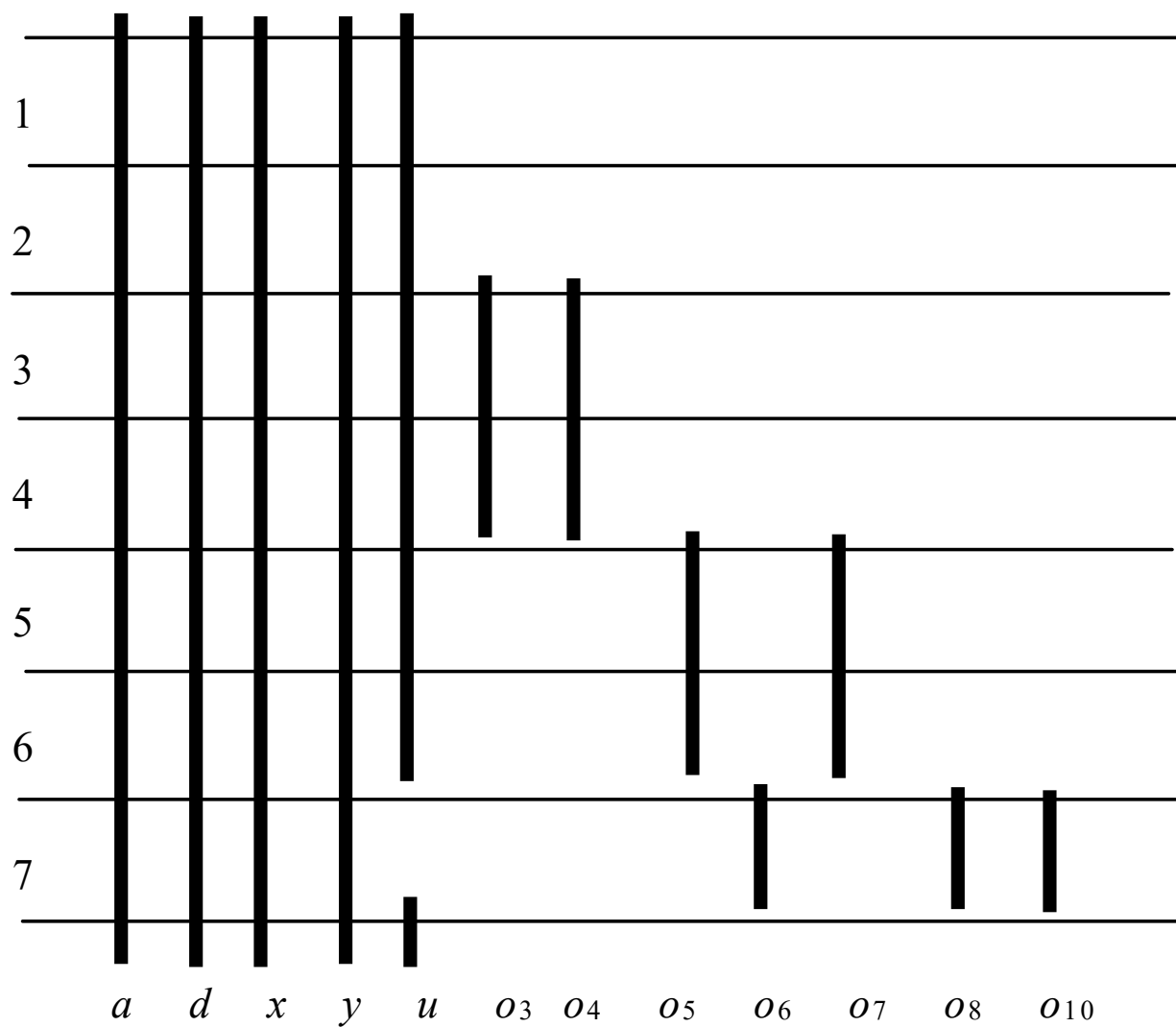


变量生存期表

变 量	产生时间 (控制步)	最后引用时间 (控制步)	备 注
x	1(1)	1(2)	循环变量
y	7(1)	7(2)	循环变量
u	7(1)	6(2)	循环变量
a	0	7	外部输入
d	0	7	外部输入
o_3	2	4	操作 3 的输出变量
o_4	2	4	操作 4 的输出变量
o_5	4	6	操作 5 的输出变量
o_6	6	7	操作 6 的输出变量
o_7	4	6	操作 7 的输出变量
o_8	6	7	操作 8 的输出变量
o_{10}	6	7	操作 10 的输出变量

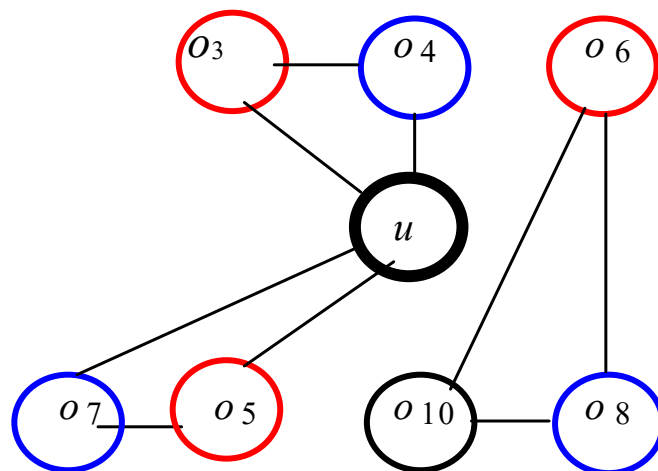
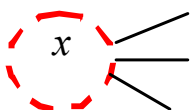
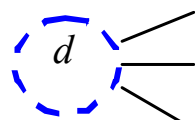
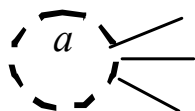
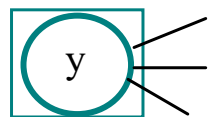


变量的生存期图

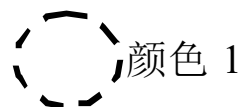




变量的冲突图



图例:



说明: 结点 a , d , x , y 和其他任何一个结点之间均应有一根连线(表示冲突), 但为了避免过多的连线使图形不清晰, 这些连线被略去。



变量分配到寄存器

变量	a	x	d	y	u	o_3	o_4	o_5	o_6	o_7	o_8	o_{10}
颜色号	1	2	3	4	5	6	7	6	6	7	7	5
寄存器	R1	R2	R3	R4	R5	R6	R7	R6	R6	R7	R7	R5



互连线路的分配

- ◆ 对于每个功能单元和寄存器的各个输入端口，重复以下步骤：
 - 若连线个数 m 大于1，分配一个 m 选1的多路器；
 - 将 m 根连线分别连接在多路器的 m 个输入端上；
 - 将多路器的输出端连接在功能单元或寄存器的输入端口上。
- ◆ 实现同一数据传输，可以用多路器也可以总线，上面的算法使用了多路器，可以用某种启发性算法决定用哪一个。



3.3.4 高层次综合中的优化技术

◆ 目的：

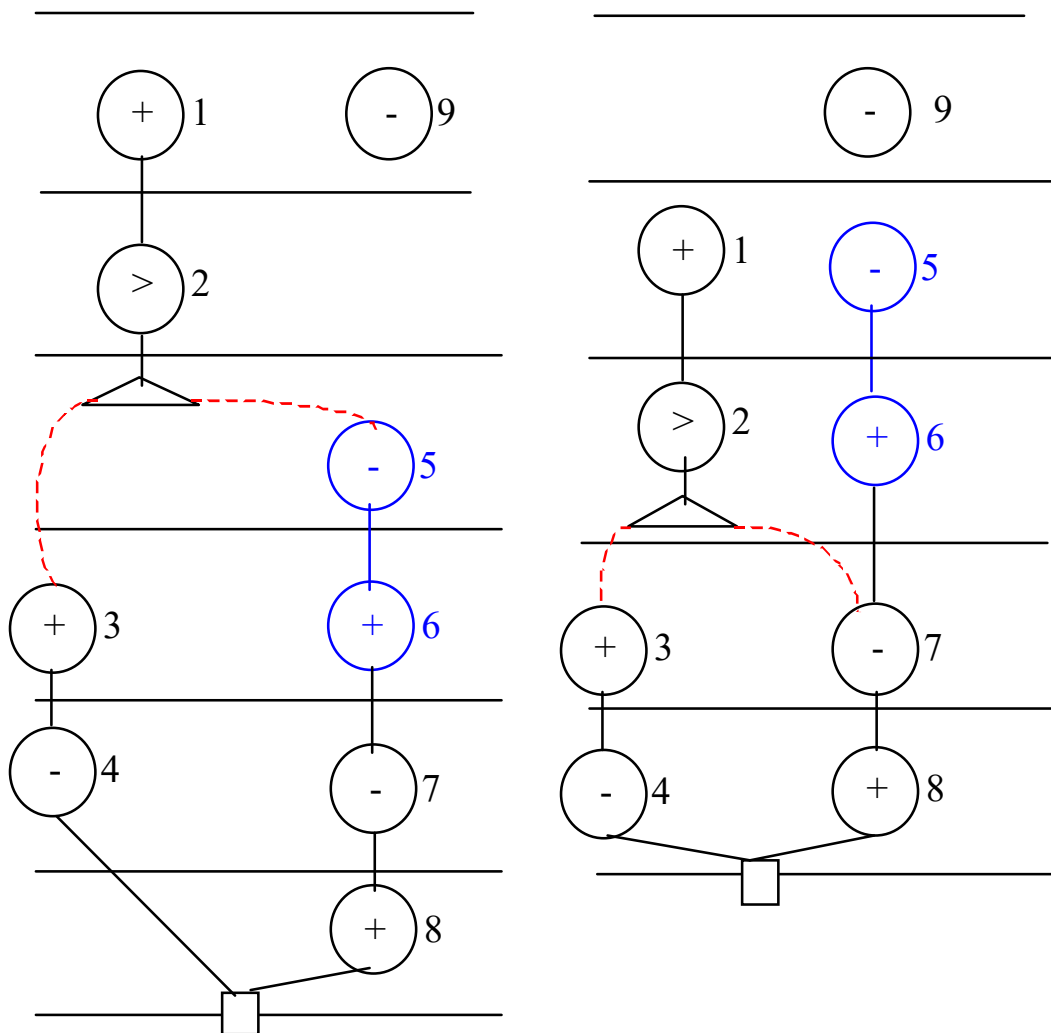
- 减少关键路径中的操作数目 → 减少控制步数；
- 减少所需硬件资源的数量；

◆ 方法：对CDFG进行操作

- 将操作从分支结构中移出分支；
- 将操作从分支结构外移入分支；
- 控制数据流图的结构变换：用结合律和分配律处理 { +, -, *... } 等操作；



3.3.4.1 将操作从条件分支中移出_{出/入} (例)



(a) 具有条件分支的控制数据流图

(b) 将操作移出条件分支

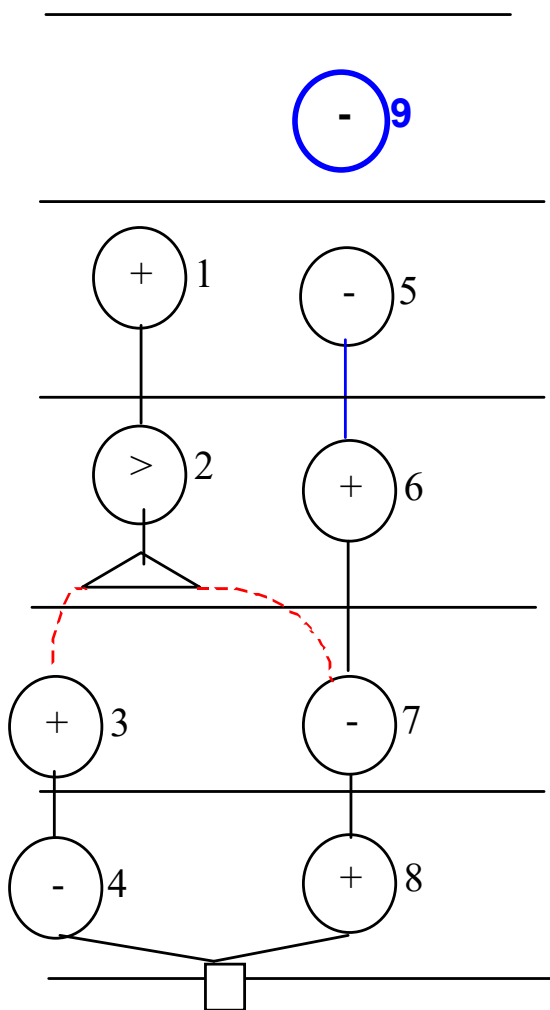


将操作从条件分支中移出（分析）

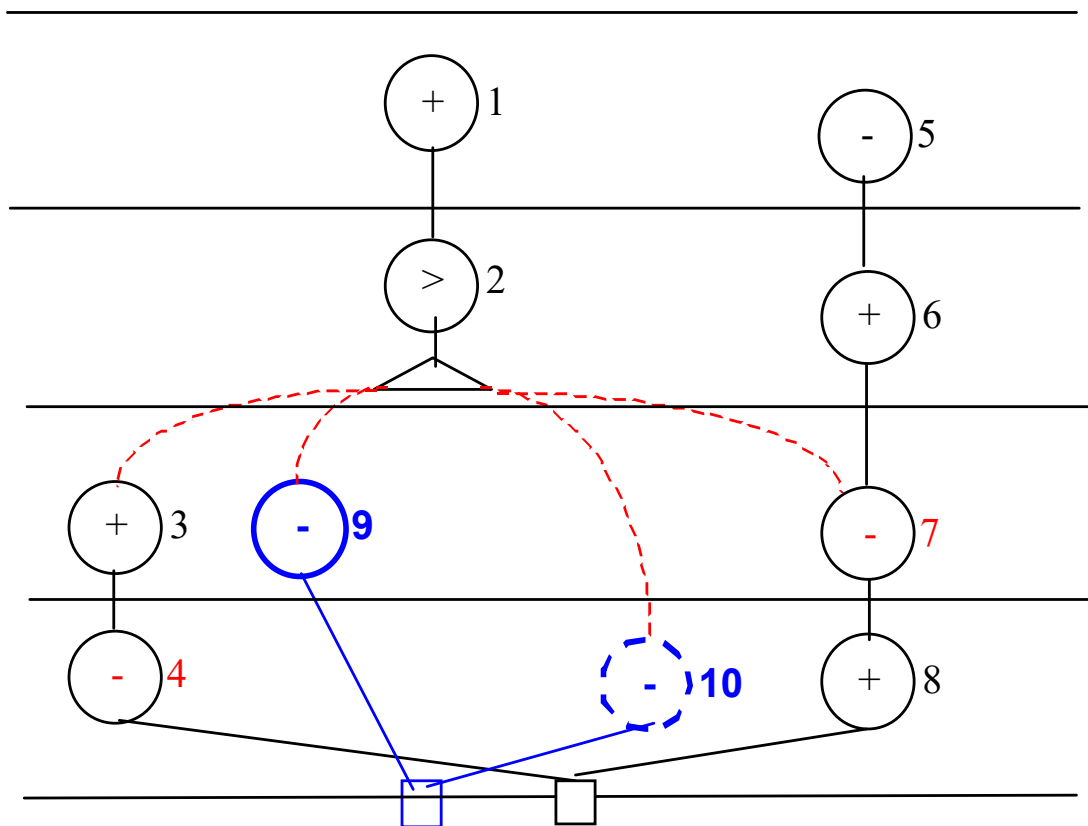
- ◆ 若操作被移出条件分支，而使该操作在条件操作之前执行，则该操作就不再属于这个条件分支，无论条件分支处于那一枝该操作都必须执行。
- ◆ 操作从条件分支中移出的利弊分析：
 - 利：移出条件分支有可能使该操作被调度到较早的控制步，有可能导致控制步总数的减少。
 - 弊：减少了与其他分支上的操作形成互斥而共享功能单元的机会。
- ◆ 对于某一特定问题，操作移出条件分支带来的是利还是弊要作具体分析。



将操作从条件分支外移入（例）



(a) 未移动前



(b) 将操作 9 复制为 (9, 10), 移入分枝



将操作从条件分支外移入（分析）

- ◆ 若要把条件分支外的操作移入条件分支，必须保证该操作在条件分支取任何值的情况下都得到执行。因此，必须对该操作加以复制，分别移入每一个条件分支。
- ◆ 移入的操作在自己所在的条件分支中可以与其他操作进行互斥操作的资源共享，设法在不增加硬件资源的条件下达到减少控制步数的目的。
- ◆ 对于前页的例子：
 - 操作10与与操作4分别处于互斥的条件分支中，可以合并成操作{4, 10}，以便共享单元。
 - 操作9与操作7互斥，可以合并成操作{7, 9}，共享单元。
- ◆ 转换后的控制数据流图所需控制步数目为4。



3.3.4.2 控制数据流图的结构变换

- ◆ 目的:

在满足硬件资源约束条件和不改变控制数据流图语义的基础上，将控制数据流图**关键路径**上的**操作**转换到**非关键路径**上去

➔ 关键路径上的操作个数减少 ➔ 关键路径长度的减少 ➔ 所需控制步的总数减少 ➔ 提高速度。

- ◆ 方法:

利用**结合律**与**分配律**的变换规则。



结合律的变换规则

类型	I	II	III
1	$a + (b + c)$	$(a + c) + b$	$(a + b) + c$
2	$a * (b * c)$	$(a * c) * b$	$(a * b) * c$
3	$a + (b - c)$	$(a - c) + b$	$(a + b) - c$
4	$a * (b / c)$	$(a / c) * b$	$(a * b) / c$
5	$a - (b - c)$	$(a + c) - b$	$(a - b) + c$
6	$a / (b / c)$	$(a * c) / b$	$(a / b) * c$
7	$a - (b + c)$	$(a - c) - b$	$(a - b) - c$
8	$a / (b * c)$	$(a / c) / b$	$(a / b) / c$

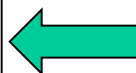




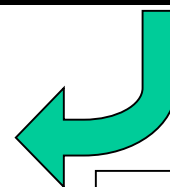
结合律的变换规则（续）

效应：

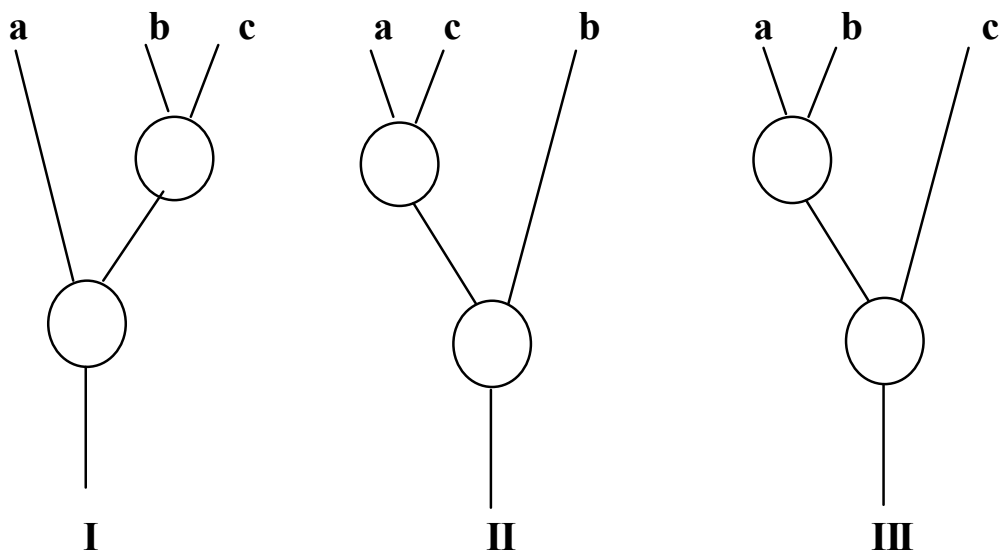
- 部分结合律变换会引起**相关操作类型的变化**（例如变换5、变换6、变换7和变换8）；
- 适当利用这种变化可使硬件资源得到充分利用；



类型	I	II	III
1	$a + (b + c)$	$(a + c) + b$	$(a + b) + c$
2	$a * (b * c)$	$(a * c) * b$	$(a * b) * c$
3	$a + (b - c)$	$(a - c) + b$	$(a + b) - c$
4	$a * (b / c)$	$(a / c) * b$	$(a * b) / c$
5	$a - (b - c)$	$(a + c) - b$	$(a - b) + c$
6	$a / (b / c)$	$(a * c) / b$	$(a / b) * c$
7	$a - (b + c)$	$(a - c) - b$	$(a - b) - c$
8	$a / (b * c)$	$(a / c) / b$	$(a / b) / c$



图形表示



结合律变换3类表达式对应的控制数据流图结构



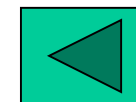
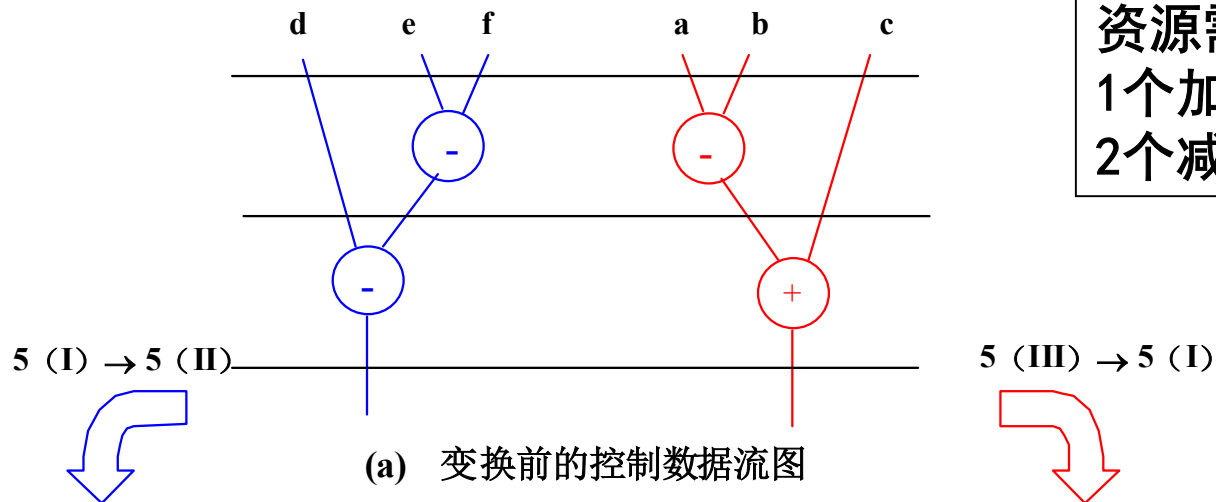
操作类型变化引起硬件资源需求变化

- ◆ 部分结合律变换会引起相关操作类型的变化（例如变换5、变换6、变换7和变换8），
 - ➔ 适当利用这种变化可使硬件资源得到充分利用。
- ◆ 这种变换的实例见下页：
 - 图(a)是变换前的控制数据流图，其中包含1个加法操作和3个减法操作，调度到2个控制步中。
 - 假设加法与减法分别用单个加法器和减法器实现，则需要1个加法器和2个减法器。
 - 对图(a)施加结合律变换 $5(I) \rightarrow 5(II)$ ，结果示于图(b)
 - 其中含2个加法操作和2个减法操作，只需要1个加法器和1个减法器即可实现。
 - 对图(a)施加结合律变换 $5(III) \rightarrow 5(I)$ ，结果示于图(c)，
 - 其中含4个减法操作，可以用2个减法器实现。

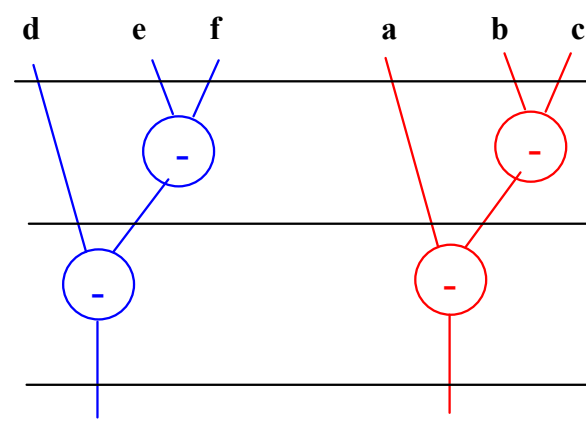
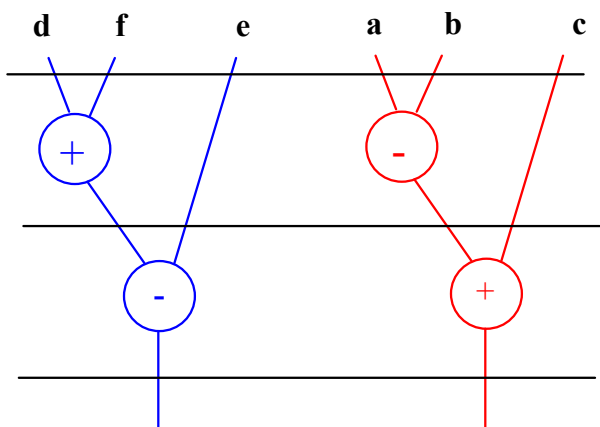


结合律变换的实例

资源需求:
1个加法器
2个减法器



资源
需求
降低



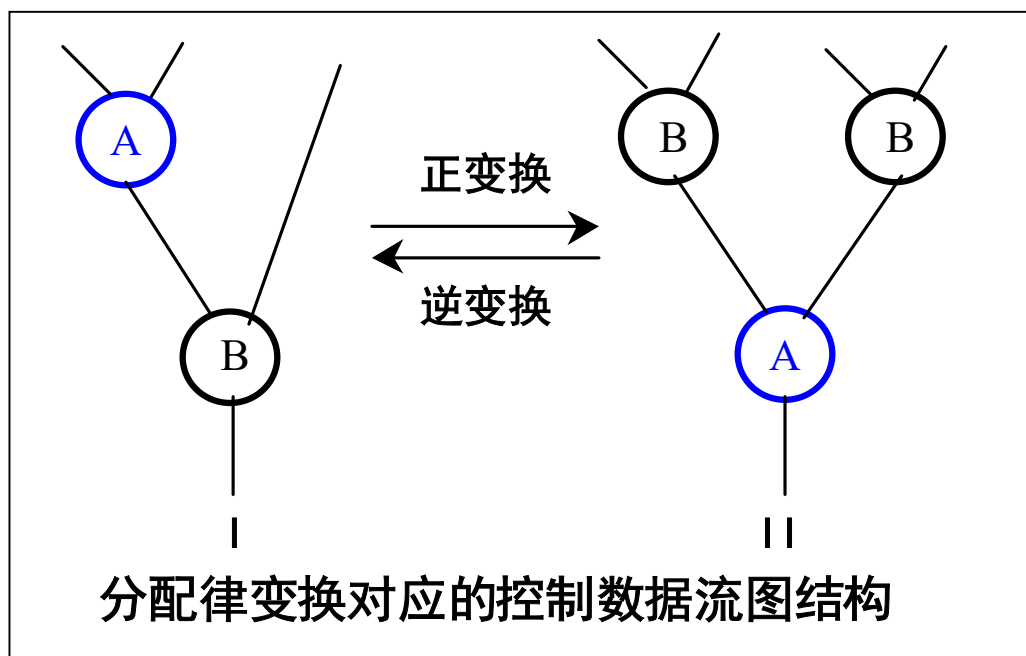


分配律的变换规则

类型	I	II
1	$(a + b) * c$	$a * c + b * c$
2	$(a - b) * c$	$a * c - b * c$
3	$(a + b) / c$	$a / c + b / c$
4	$(a - b) / c$	$a / c - b / c$

A代表：
 $\{+, -\}$;

B代表：
 $\{\times, \div\}$



正变换:

I → II

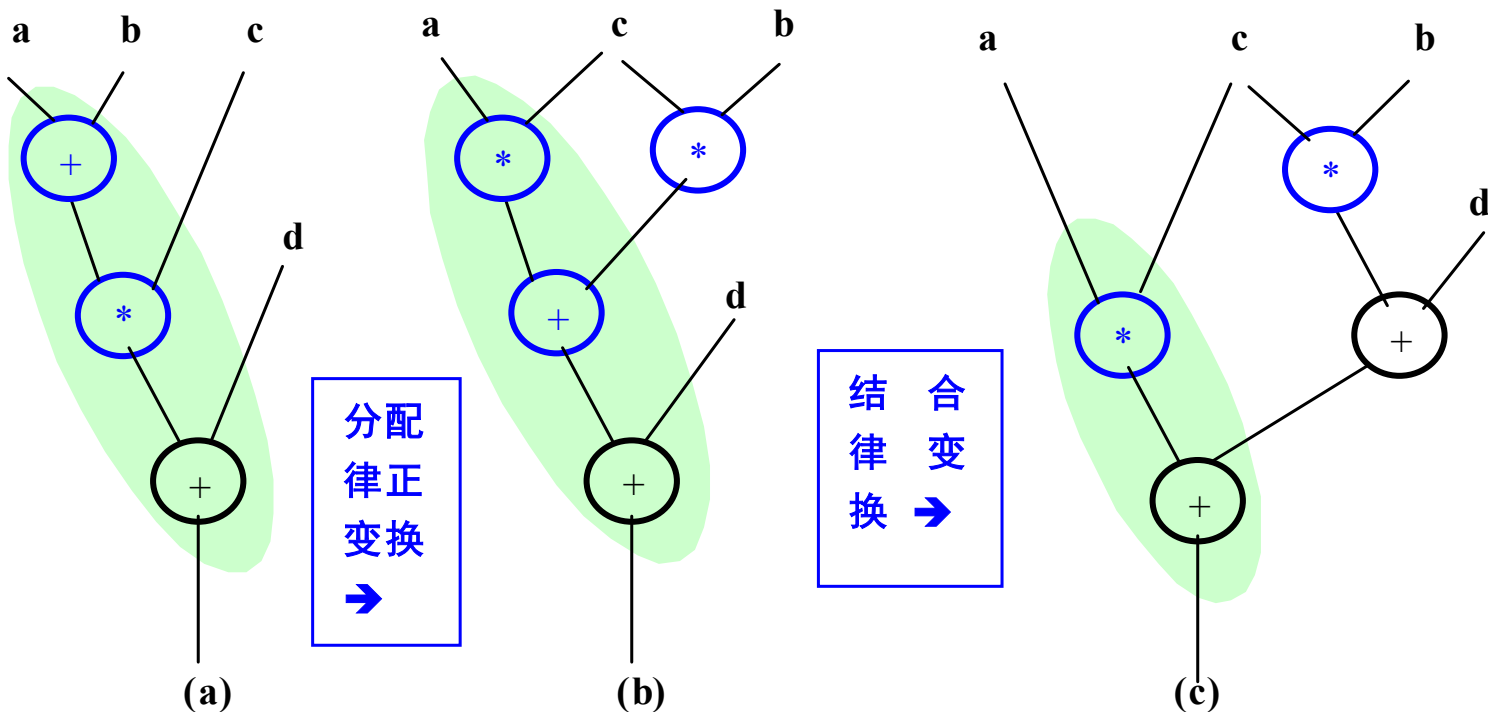
逆变换:

II → I



分配律变换的实例

- ◆ 利用结构变换缩短控制数据流图关键路径长度：
- ◆ 假设图中阴影部分为关键路径上的一段，加法操作和乘法操作均在一个控制步内完成。
- ◆ 关键路径长度：由3缩短为2！





本章小结

- ◆ 高层次综合的实质是从行为描述到电路结构描述（一般指RTL级）的转换。
- ◆ 高层次综合的核心算法是调度和分配。
- ◆ 硬件设计者用行为描述的方式描述自己的设计，有如下优点：
 - 简练；
 - 概念清晰；
 - 易于修改和排除错误；
 - 缩短设计周期。
- ◆ 高层次综合和较低层次的逻辑综合相比，技术复杂性增大很多。 → 把困难交给EDA工具！
- ◆ 高层次综合技术目前还不够成熟。