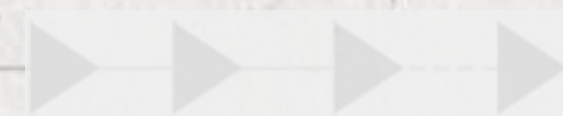


芯动力——硬件加速设计方法

第二章：高质量VerilogHDL描述方法(1)

邸志雄@西南交通大学
zxdi@home.swjtu.edu.cn



一、关于Verilog HDL的认知

常见认知

对verilog的误解

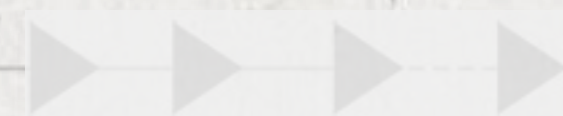
- 很多语法规则与C语言相似，书写时可参考C语言；
- 追求代码的整洁、简短；
- 着眼于代码书写，性能优化由综合器实现；
- 把Verilog代码当做了程序，把电路设计当成了编程；

正确认知

HDL: Hardware Description

硬件描述语言

- HDL语言仅是对已知硬件电路的文本表现形式编写前，对所需实现的硬件电路“胸有成竹”

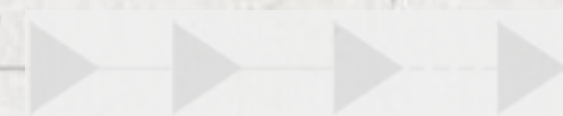


- Verilog HDL的基本功能之一是描述可综合的硬件电路。如何合理使用Verilog HDL描述高性能的可综合电路是本次课的主旨之一。
- 相比C语言，最显著的区别在与HDL语言具备以下硬件设计的基本概念：

互连(connectivity) : - wire型变量描述各个模块之间的端口与网线连接关系

并发(concurrency): - 可以有效地描述并行的硬件系统

时间(time): - 定义了绝对和相对的时间度量，可综合操作符具有物理延迟



二、Verilog HDL用于可综合描述的语句

可综合“四大法宝”

always

If-else

case

assign

可综合风格禁止出现

function

for

fork-join

while

testbench



1

If-else相关语句的硬件结构映射及优化

2

case相关语句的硬件结构映射及优化

3

慎用Latch

4

逻辑复制

5

逻辑共享

6

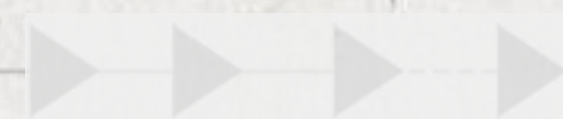
资源顺序重排

7

同步复位与异步复位

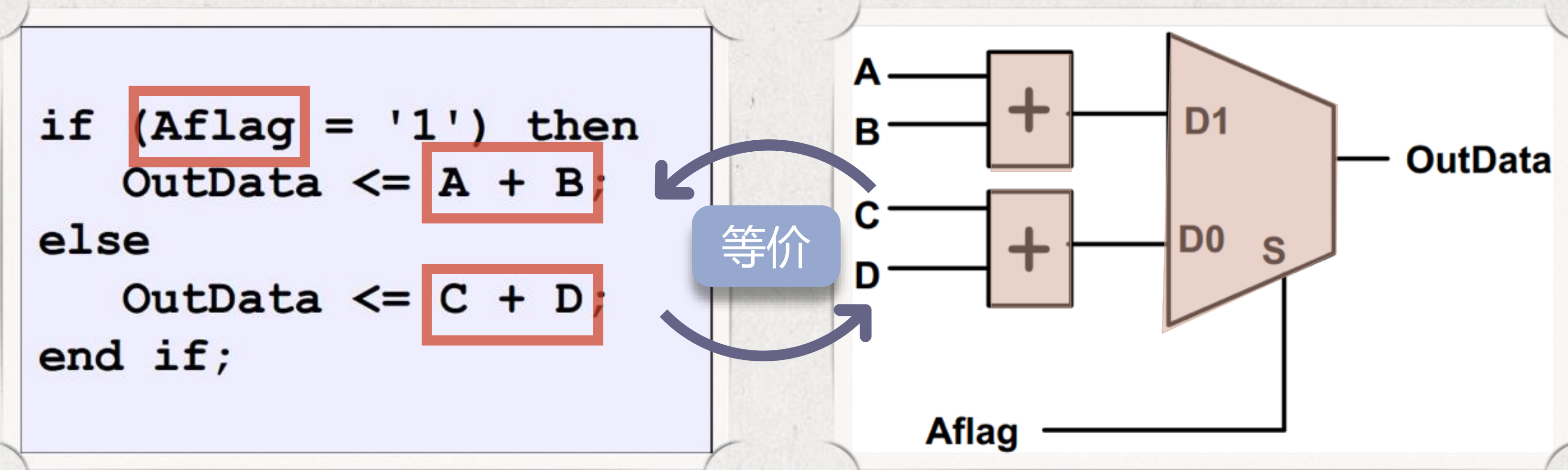
8

少用 “: ? ” 赋值语句



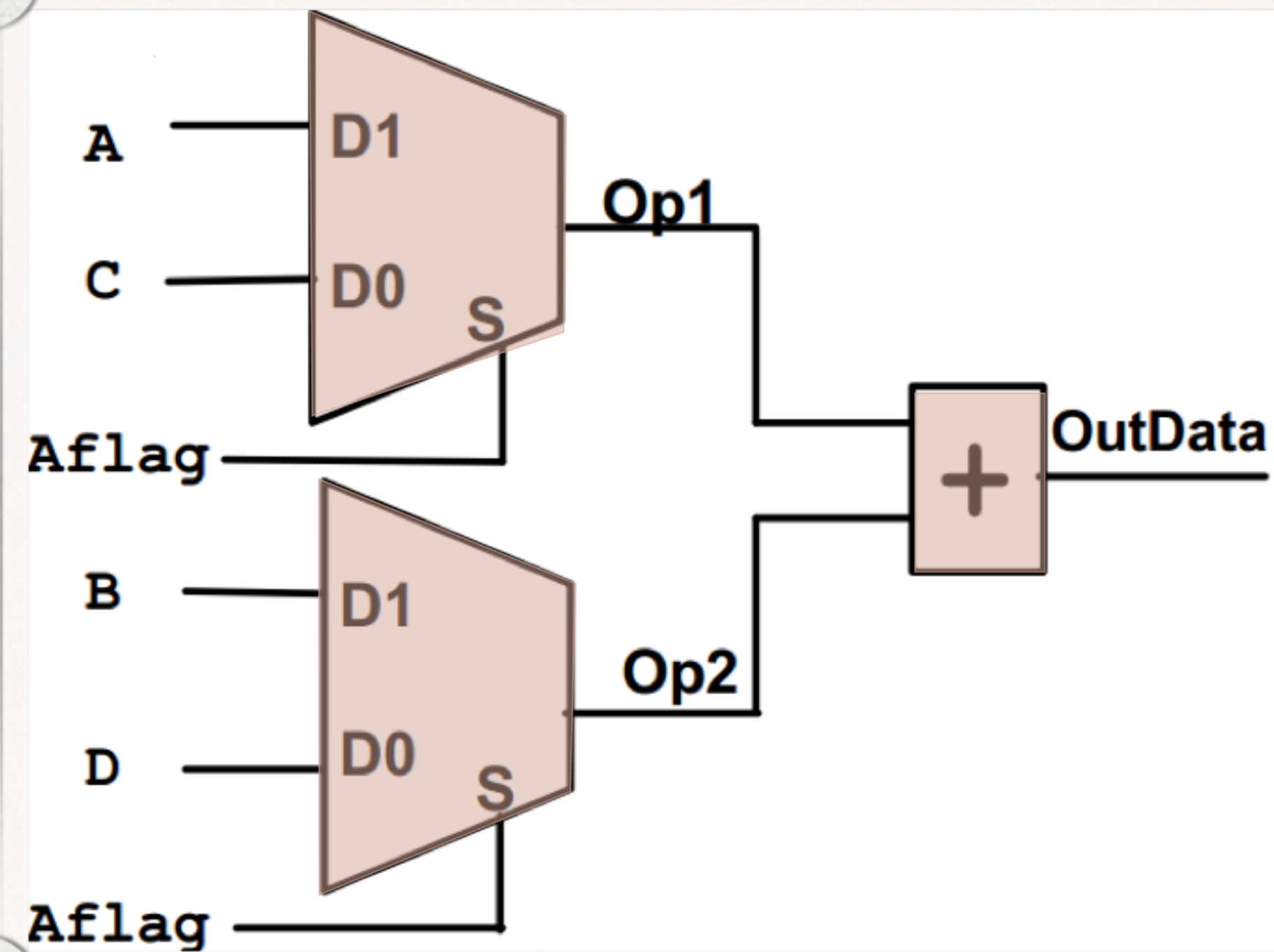


- 映射的硬件结构: Multiplexing Hardware (多路选择器) 输出结果由输入的选择条件决定

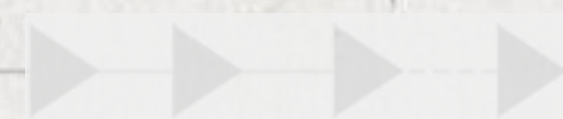
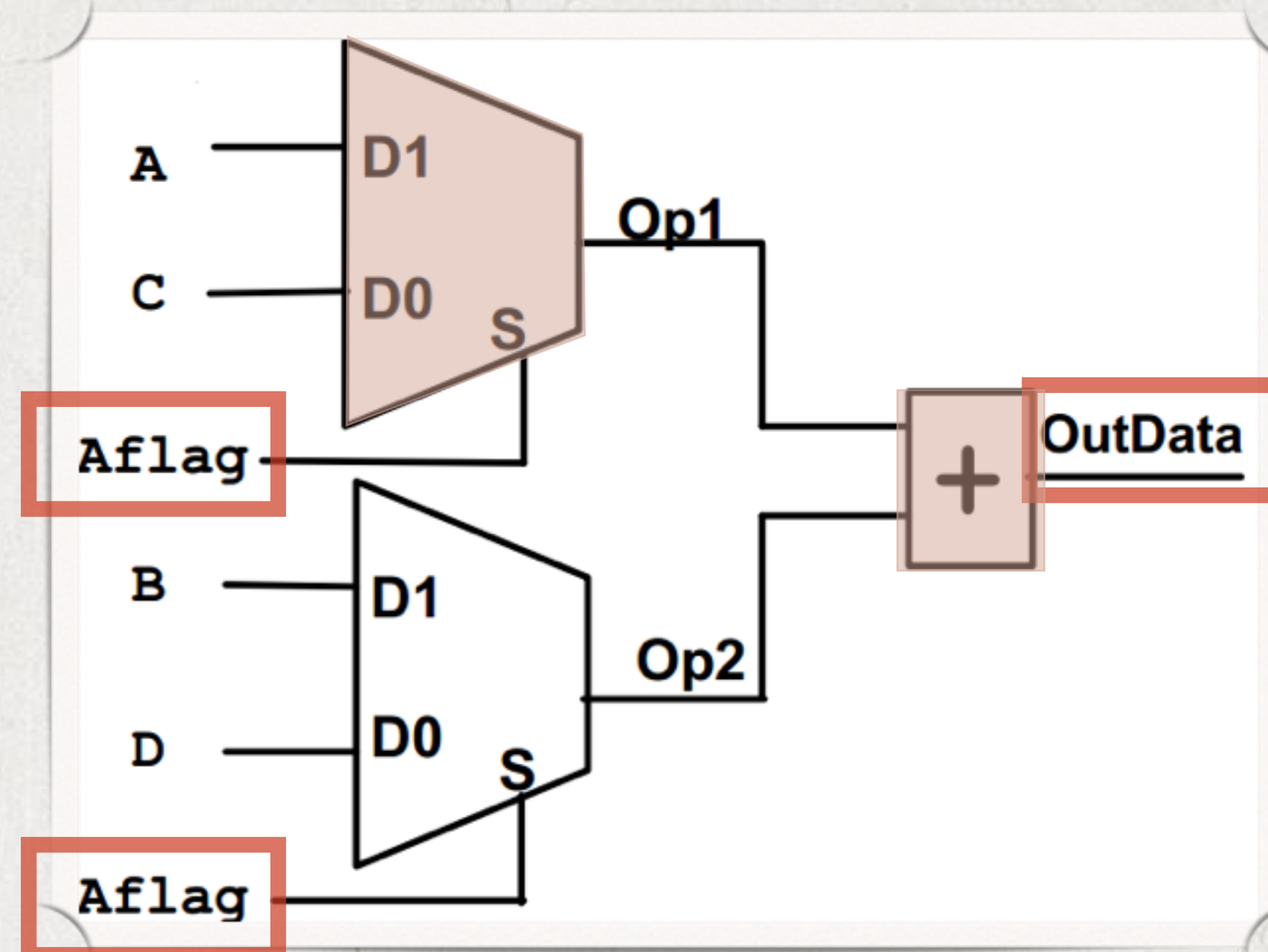
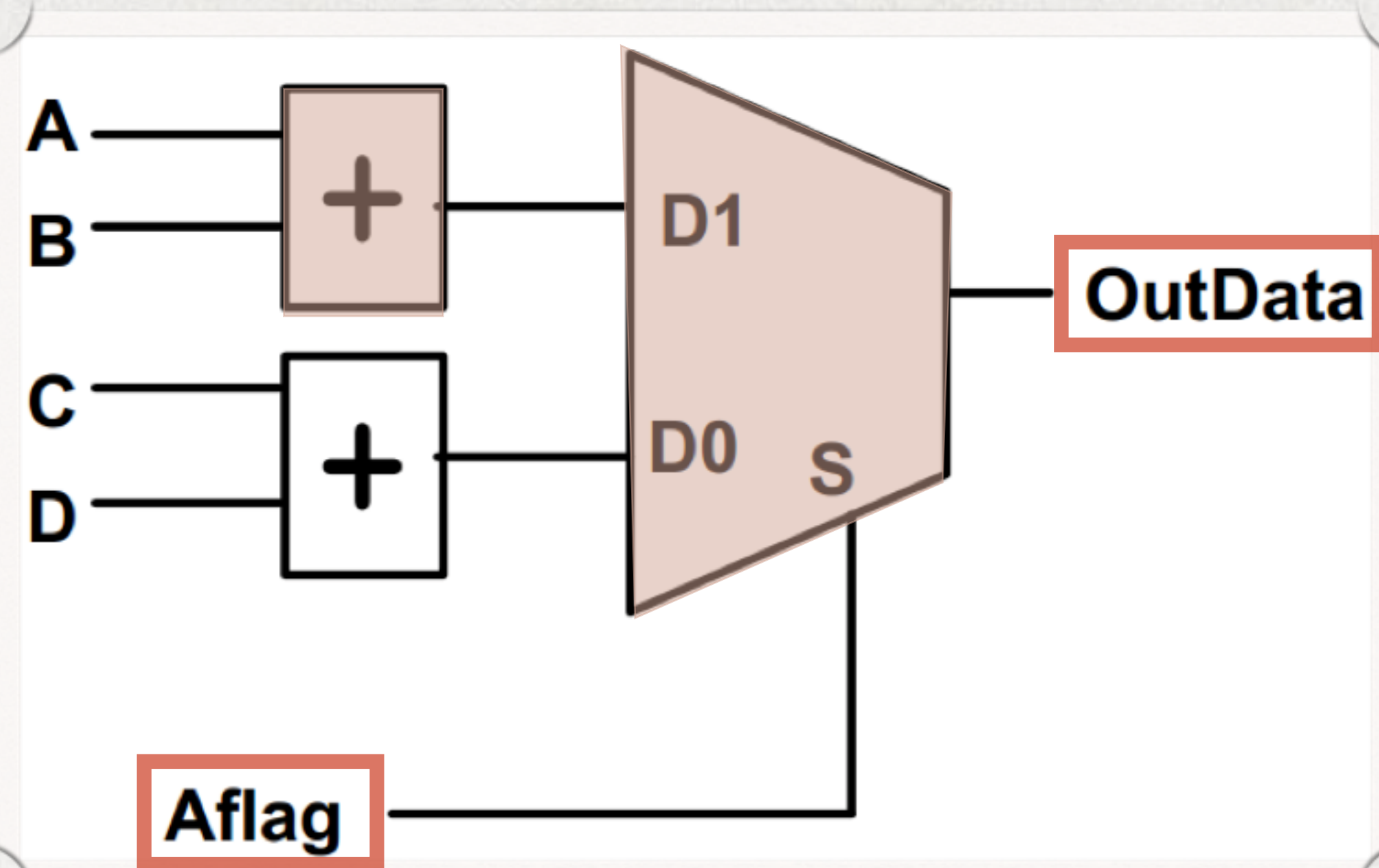


- 重构if-else映射的硬件结构：减少了一个加法器，减小了硬件的面积

```
if (Aflag == 1'b1)
  begin
    Op1 <= A;
    Op2 <= B;
  end
else
  begin
    Op1 <= C;
    Op2 <= D;
  end
  OutData <= Op1 + Op2;
```



需根据输入约束，小心设计：先“加”后“选”，先“选”后“加”



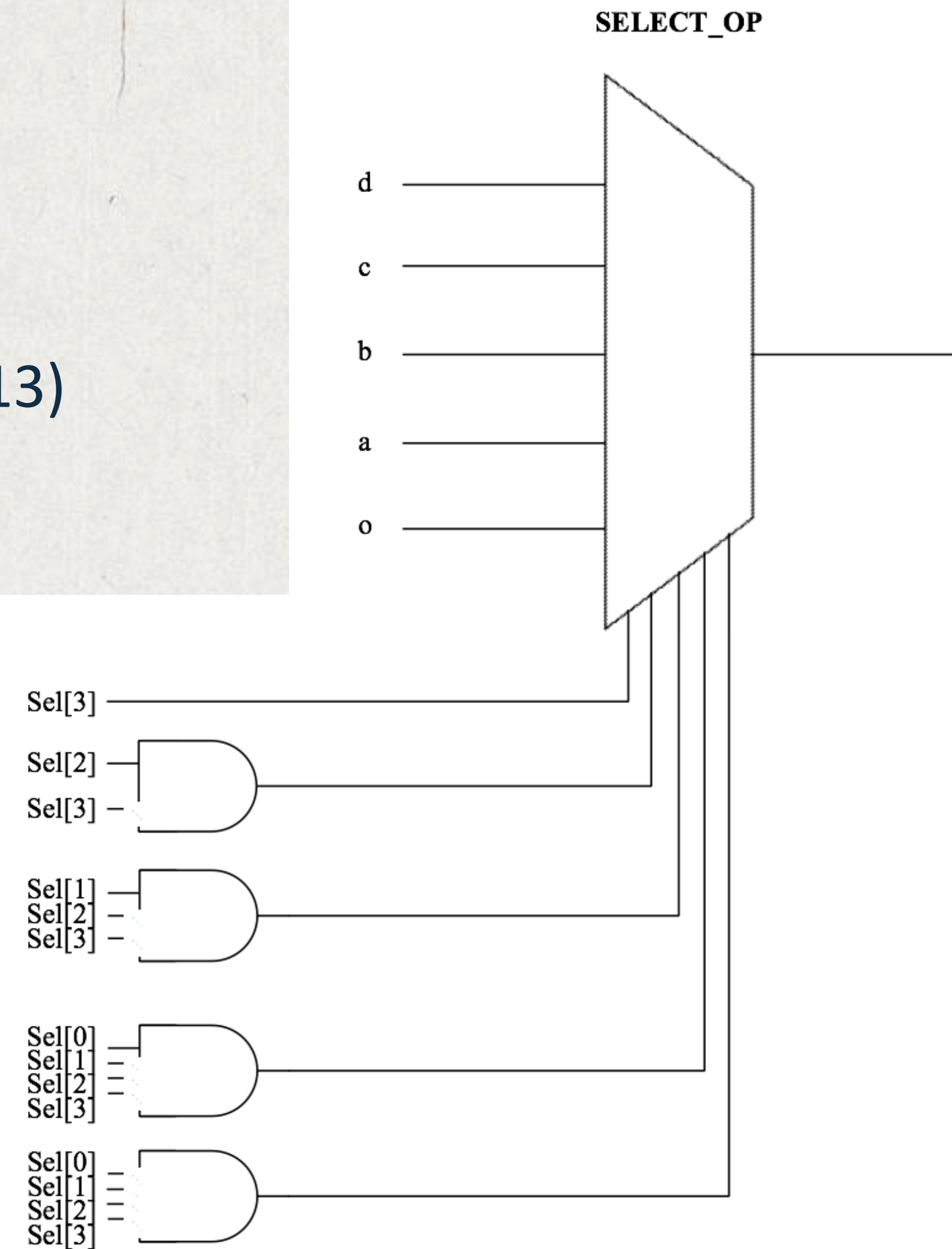
单if语句

1

无优先级的判断结构

2

```
always @(a or b or c or d or sel0 or sel1 or sel2 or sel3)
begin
    z=0;
    if (sel3)
        z=d;
    else if (sel2)
        z=c;
    else if (sel1)
        z=b;
    else if (sel0)
        z=a;
end
```



推荐初学者尽量使用单if语句(if...else if.. .else if...) 描述多条件判断结构

多if语句

1

具有优先级的判断结构

3

```
always @(a or b or c or d or sel0 or sel1 or sel2 or sel3)
```

```
begin
```

```
    z=0;
```

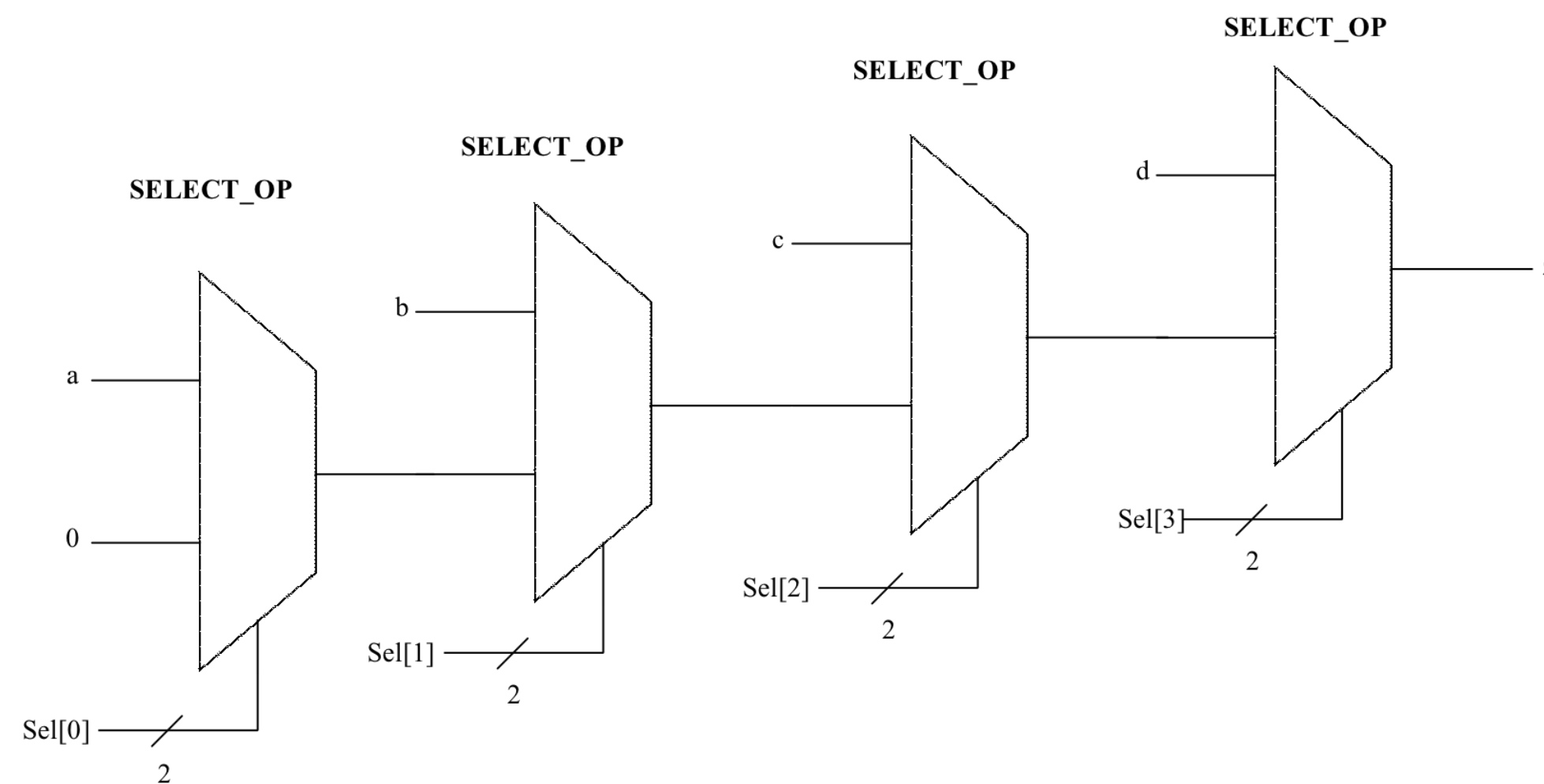
```
    if (sel0) z=a;
```

```
    if (sel1) z=b;
```

```
    if (sel2) z=c;
```

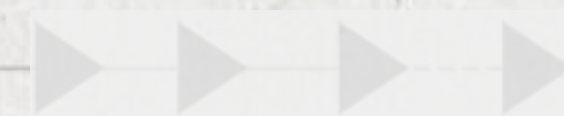
```
    if (sel3) z=d;
```

```
end
```



- 最后一级选择信号具有最高优先级
- 具有优先级的多选结构会消耗组合逻辑

不推荐



always @(a or b or c or d or sel0 or sel1 or sel2 or sel3)

begin

z=0;

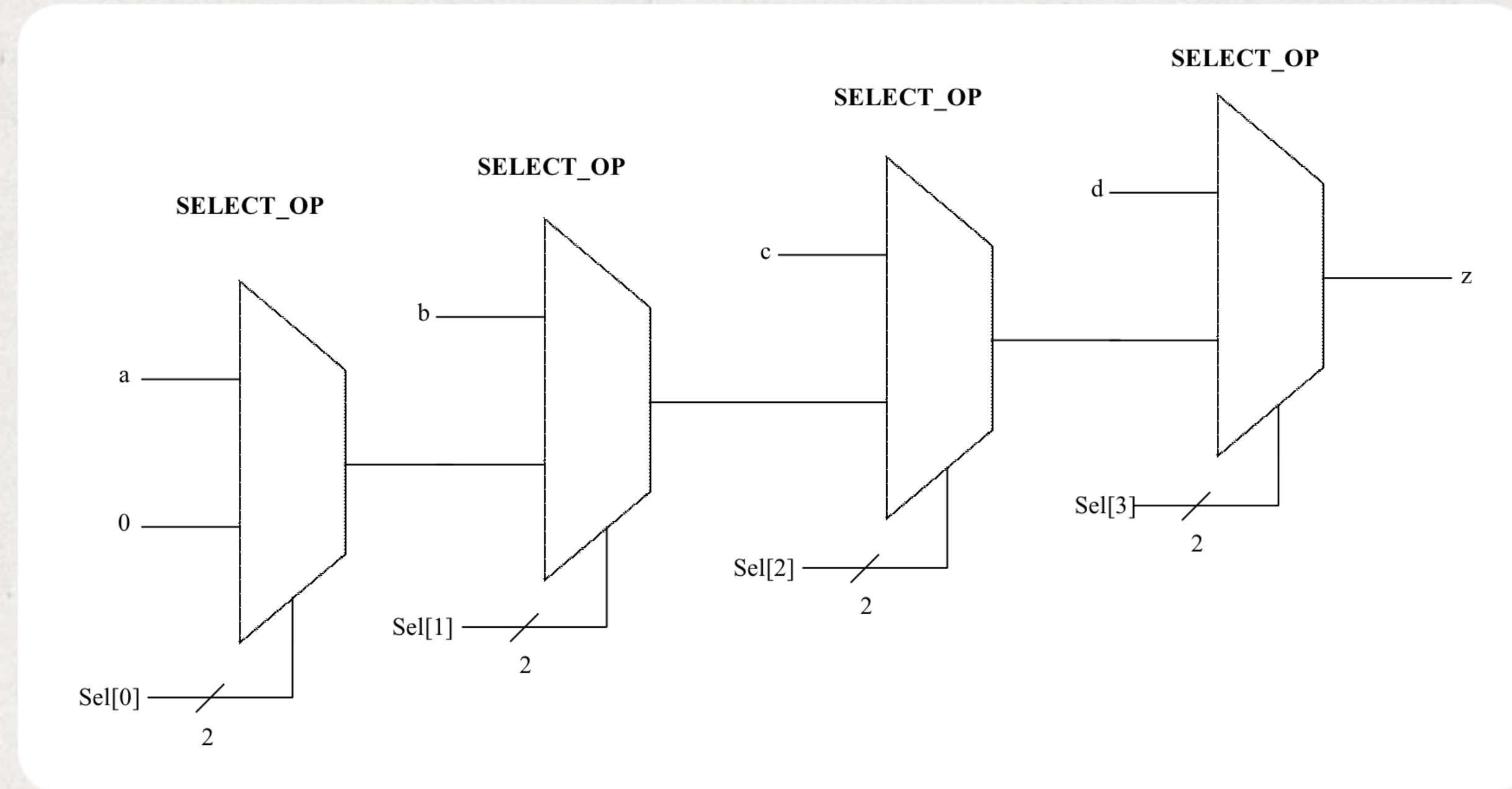
if (sel1) z=b;

if (sel2) z=c;

if (sel3) z=d;

if (sel0) z=a;

end



- 若某些设计中，有些信号要求先到达(如关键使能信号、选择信号等)，而有些信号需要后到达(如慢速信号、有效时间较长的信号等)，此时则需要使用if...if...结构。
- 设计方法：最高优先级给最迟到达的关键信号

case

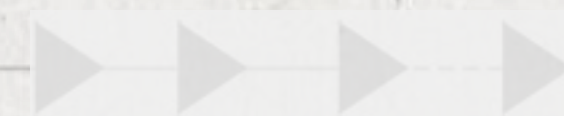
2

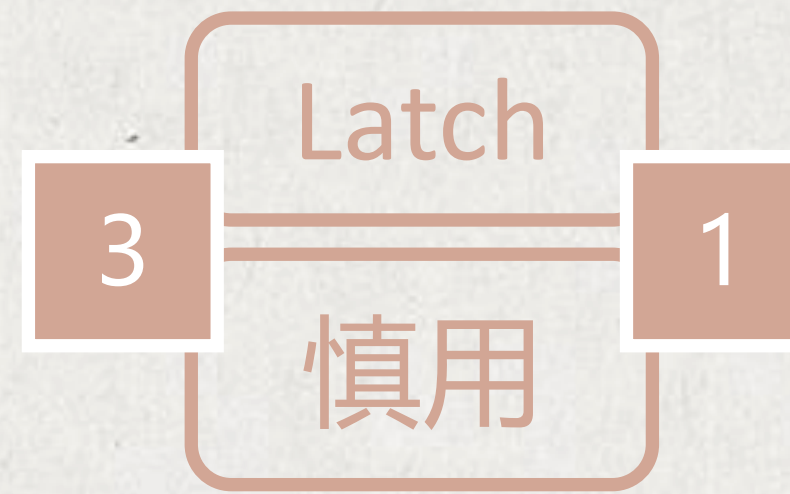
1

无优先级的判断结构

```
always @(a or b or c or d or sel0 or sel1)
begin
    case ({sel0, sel1})
        2'b00: z=d;
        2'b01: z=c;
        2'b10: z=b;
        2'b11: z=a;
        default: z=1'b0;
    endcase
end
```

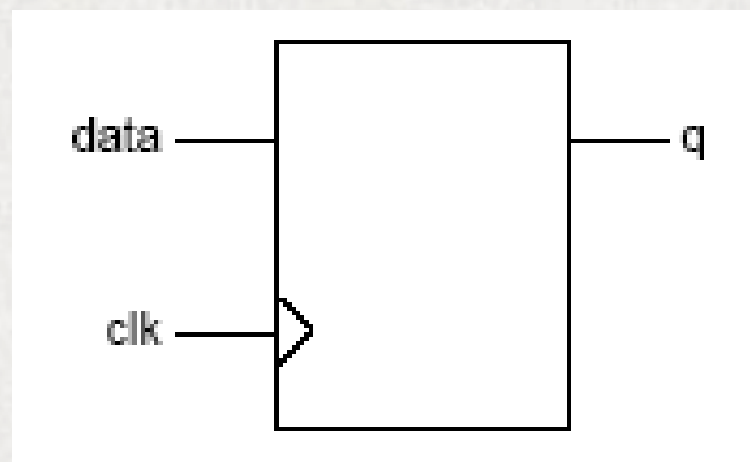
- 与单if语句的区别：条件互斥
- 多用于指令译码电路



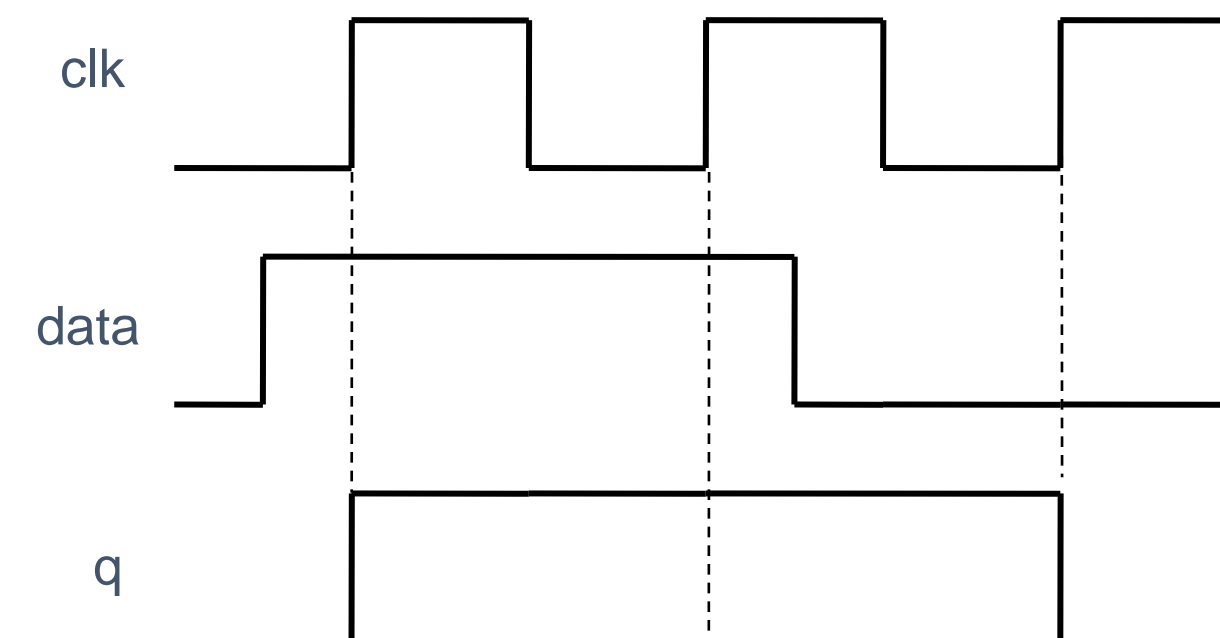


- 综合器很难解释latch，因此，除非特殊用途，一般避免引入latch。

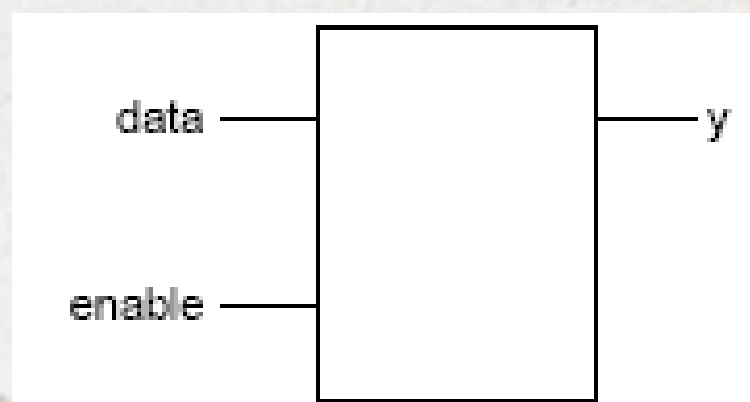
Register (Flip-flop, FF) — Edge Triggered



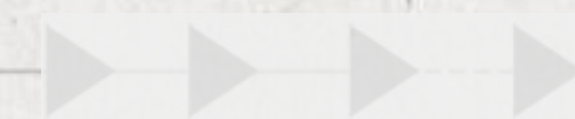
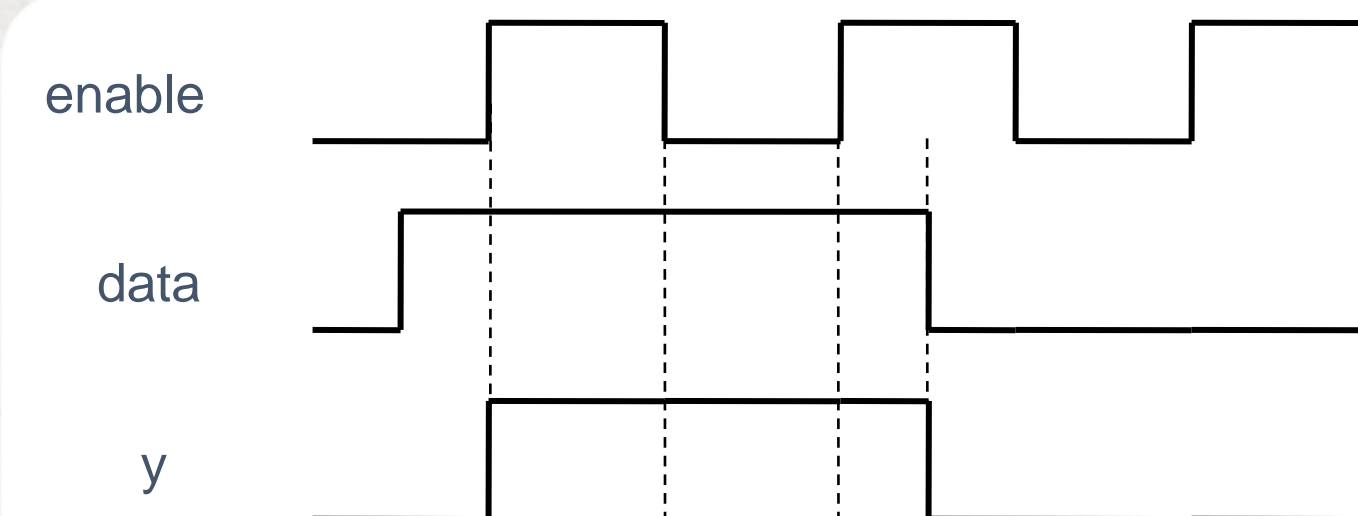
```
always @(posedge clk)
  q <= data;
```



Latch — Level Sensitive



```
always @(data or enable)
  if (enable)
    y = data;
```



- latch由电平触发，非同步控制。在使能信号有效时latch相当于通路，在使能信号无效时latch保持输出状态。DFF由时钟沿触发，同步控制。
- latch容易产生毛刺（glitch），DFF则不易产生毛刺。
- latch将静态时序分析变得极为复杂。

一般的设计规则是：在绝大多数设计中避免产生latch。latch最大的危害在于不能过滤毛刺。这对于下一级电路是极其危险的。所以，只要能用D触发器的地方，就不用latch。

- 易引入**latch**的途径：使用不完备的条件判断语句

缺少else

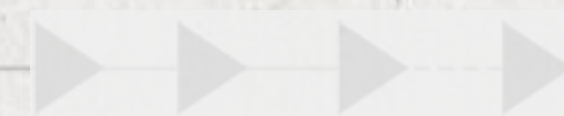
```
always @(cond1 or data in)
begin
    if(cond1==1 )
        data_out=data_in;
end
```

缺少default

```
case ({sel0, sel1, sel2})
    2'd0: z=d;
    2'd1: z=c;
    2'd2: z=b;
endcase
```

防止产生非目的性Latch的措施：

- 使用完备的if...else语句
- 为每个输入条件设计输出操作，为case语句设置default操作
- 仔细检查综合器生成的报告，latch会以warning的形式报告



3

Lfull-case和parallel-Case

2

综合器指令

实例：交通控制红绿灯，只有红、绿、黄三种情况，不会出现第四种

```
always @(a or b or c or sel)
```

```
case (sel)
```

```
2'b00: y = a;
```

```
2'b01: y = b;
```

```
2'b10: y = c;
```

```
endcase
```

```
endmodule
```

Statistics for case statements in always block at line 7 in file '.../mux3a.v'

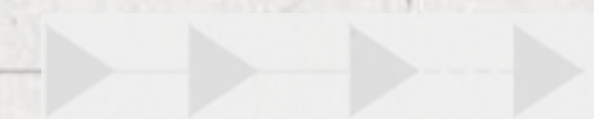
```
=====
| Line | full/ parallel |
=====
```

```
| 9 | no/auto |
=====
```

Inferred memory devices in process in routine mux3a line 7 in file '.../mux3a.v'.

```
=====
| Register Name | Type | Width | Bus | MB | AR | AS | SR | SS | ST |
=====
```

```
| y_reg | Latch | 1 | - | - | N | N | - | - | - |
=====
```



- full case: 告诉综合器, 当前case结构所列条件已完备

```
always @(a or b or c or sel)
  case (sel) // synopsys full_case
    2'b00: y = a;
    2'b01: y = b;
    2'b10: y = c;
  endcase
endmodule
```

Warning: You are using the full_case directive with a case statement in which not all cases are covered.

Statistics for case statements in always block at line 7 in file '.../mux3b.v'

=====

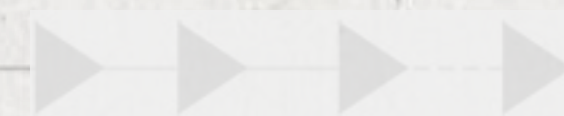
| Line | full/ parallel |

=====

| 9 | user/auto |

=====

报告中没有出现latch




```
always @(irq) begin
  {int2, int1, int0} = 3'b0;
  casez (irq)
    3'b1??: int2 = 1'b1;
    3'b?1?: int1 = 1'b1;
    3'b???1: int0 = 1'b1;
  endcase
end
```

如果条件不互斥，则存在优先级

Statistics for case statements in always block at line 6 in file
'.../intctl1a.v'

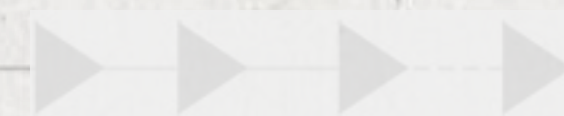
=====

Line	full/	parallel
------	-------	----------

=====

9	no/no	
---	-------	--

=====



Parallel-Case: 告诉DC, 所有条件均互斥, 且并行, 无优先权

```
always @(irq) begin
```

```
{int2, int1, int0} = 3'b0;
```

```
casez (irq) // synopsys parallel_case
```

```
    3'b1??: int2 = 1'b1;
```

```
    3'b?1?: int1 = 1'b1;
```

```
    3'b??1: int0 = 1'b1;
```

```
endcase
```

```
end
```

Statistics for case statements in always block at line 6 in file

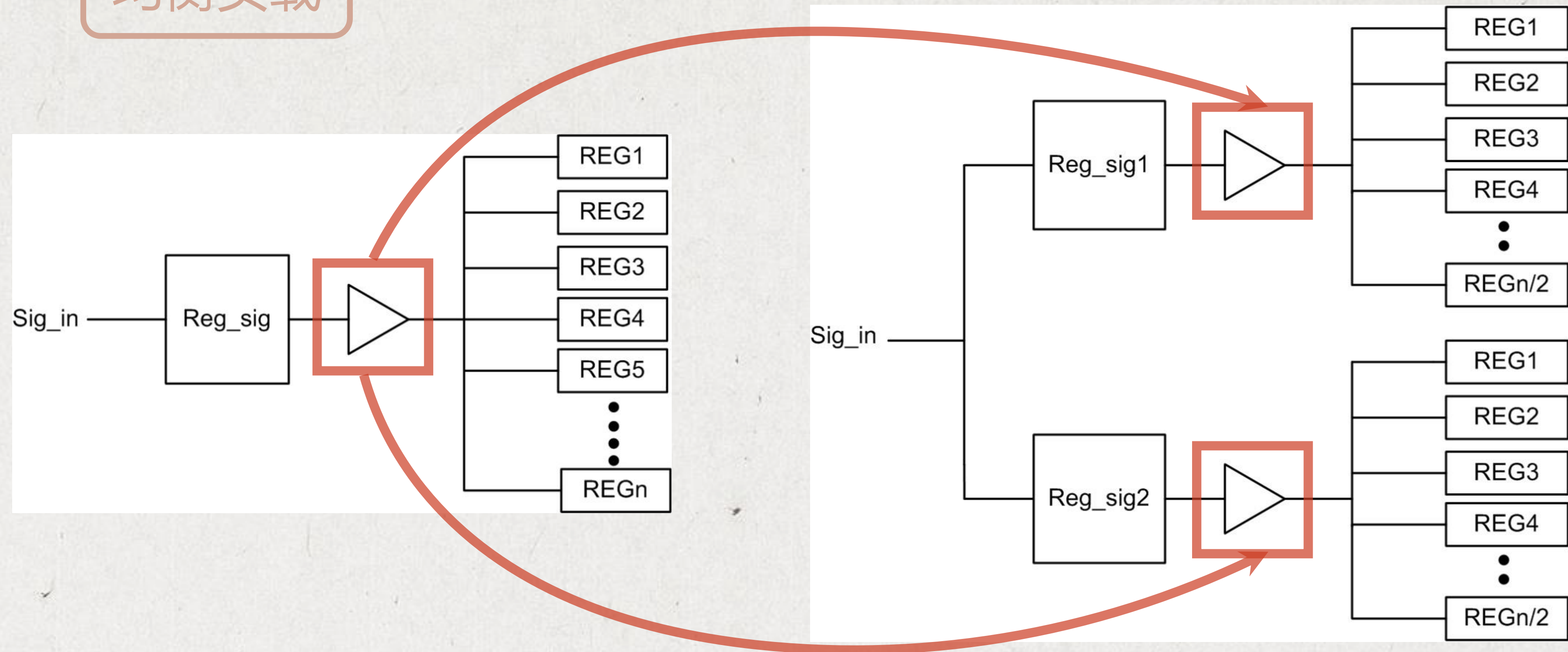
'.../intctl1a.v'

=====
| Line | full/ parallel |

=====
| 9 | no/user |

合理使用parallel case约束, 可以条件译码逻辑

4 逻辑复制 1
均衡负载



通过逻辑复制，降低关键信号的扇出，进而降低该信号的传播延迟，提高电路性能。

5

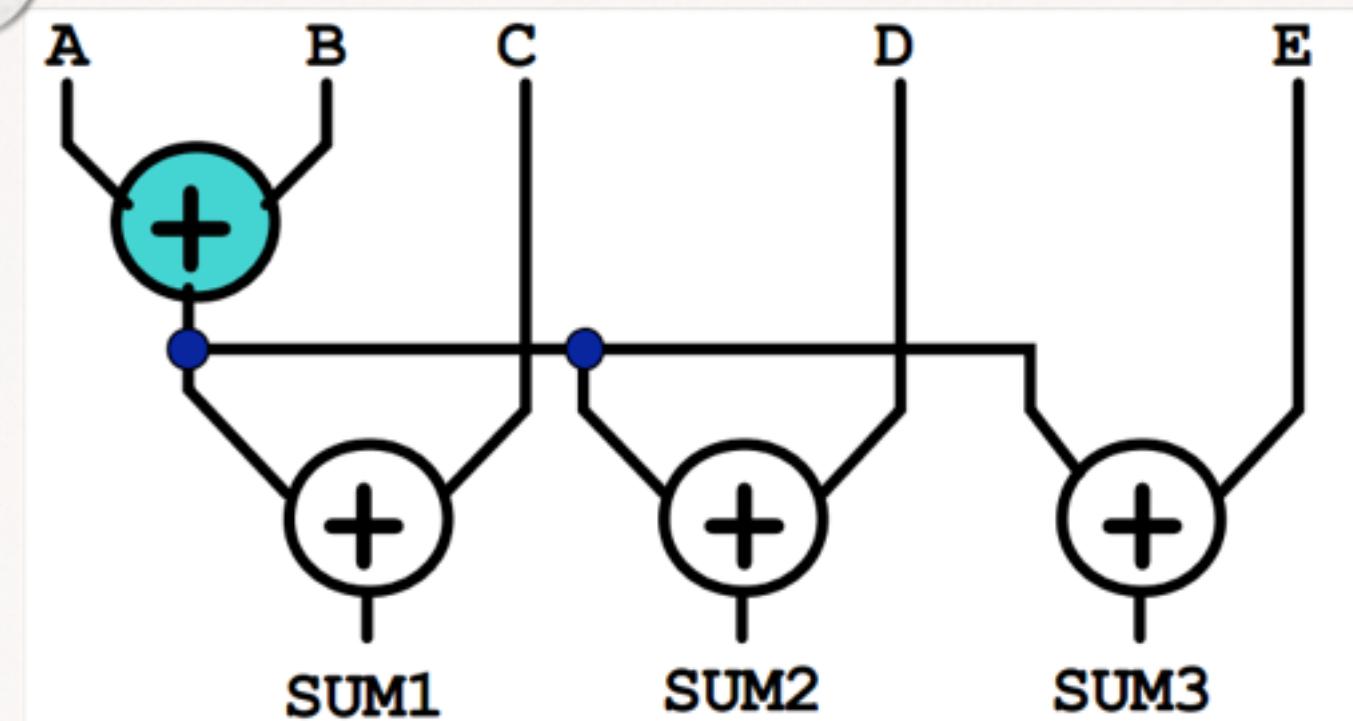
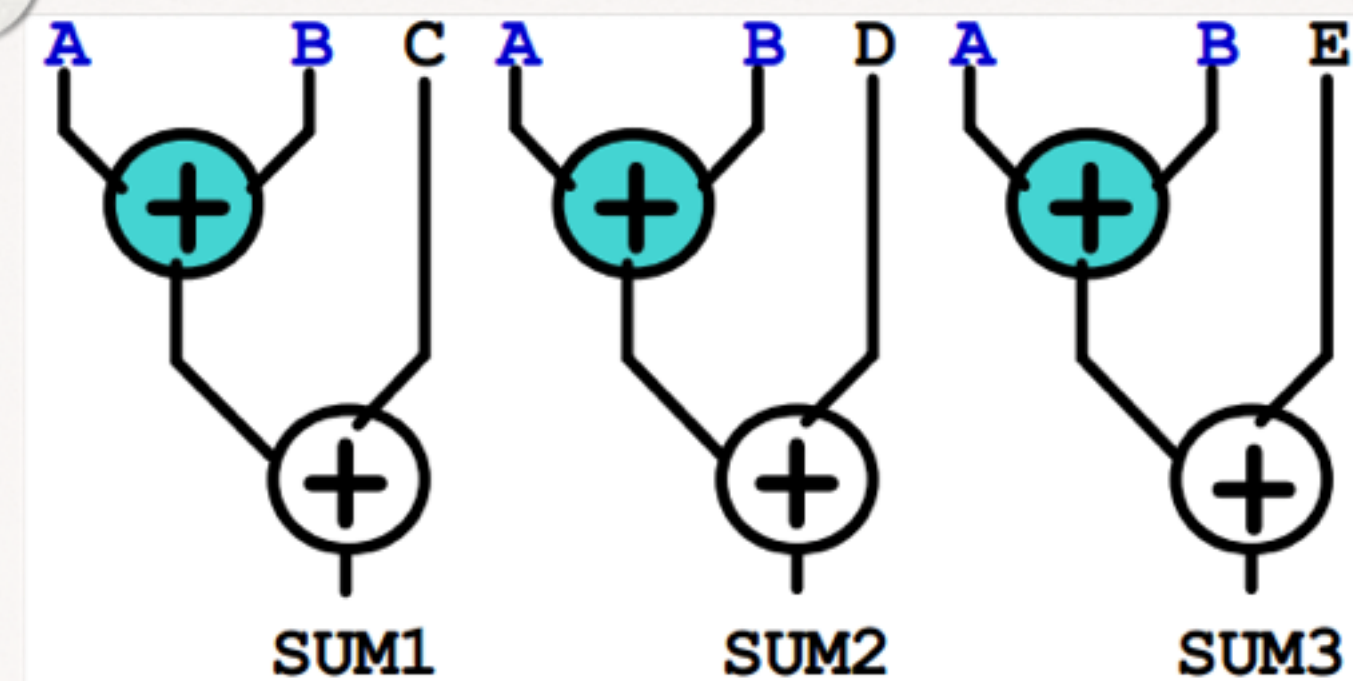
资源共享

1

减小面积

```
SUM1 <= A + B + C;  
SUM2 <= A + B + D;  
SUM3 <= A + B + E;
```

```
temp := A + B;  
SUM1 <= temp + C;  
SUM2 <= temp + D;  
SUM3 <= temp + E;
```



资源顺序重排

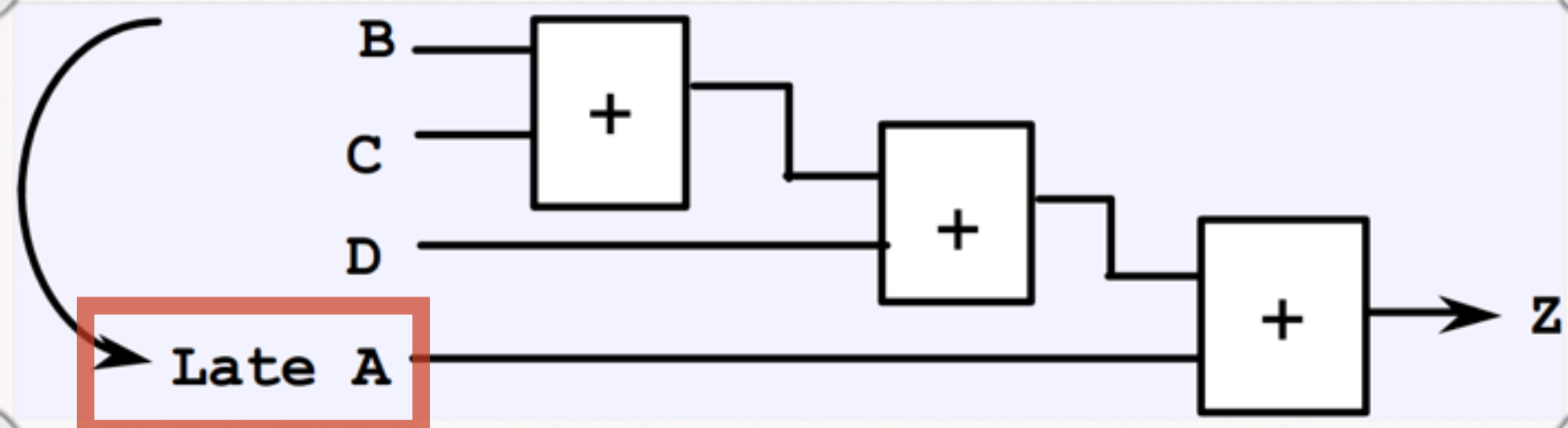
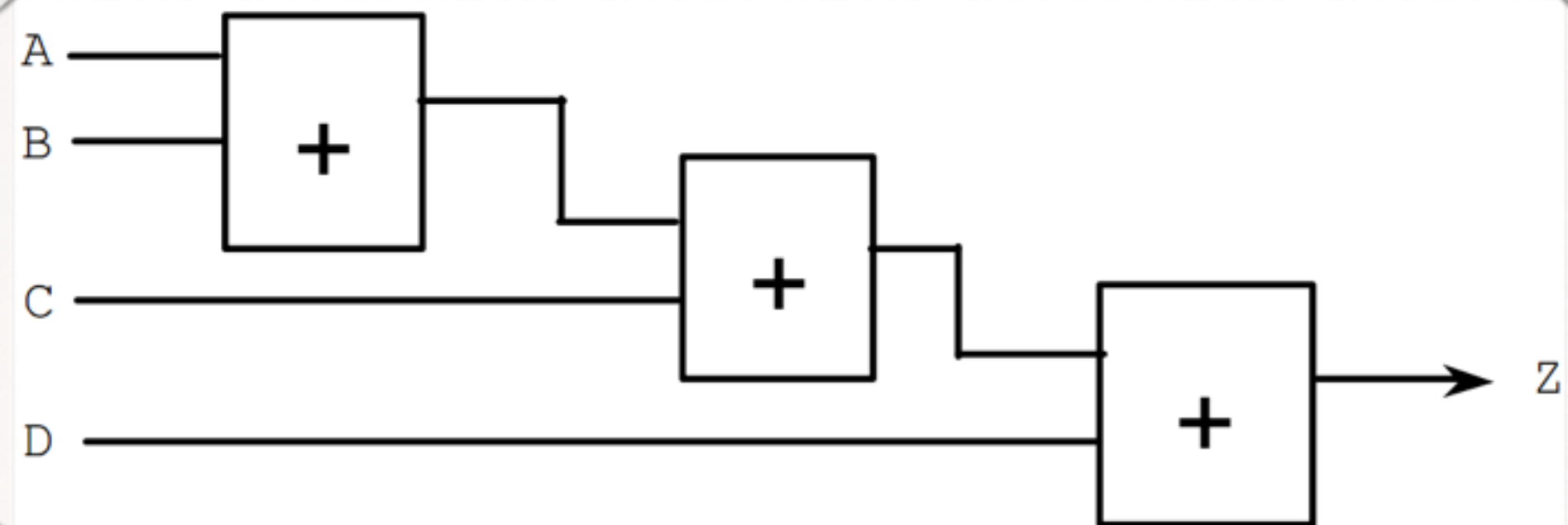
6

1

降低传播延时

$$Z=A+B+C+D$$

$$Z=((B+C)+D)+Late_A$$



7

": ? " 仅用于连线

1

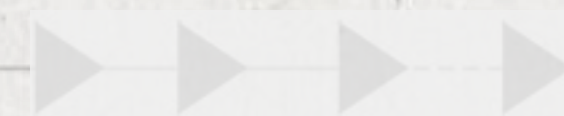
always用于逻辑运算

```
assign a = (b=1)? ((c && d) ? 1'b1:1'b0) :1'b0;
```

关于assign:

- 仅用于信号连接
- 难以阅读, 且多层嵌套后很难被综合器解释

```
always @ ( * )  
begin  
    if (b==1'b1)  
        if(c && d==1'b1)  
            a=1'b1;  
        else a=1'b0;  
    else a=1'b0;  
end
```





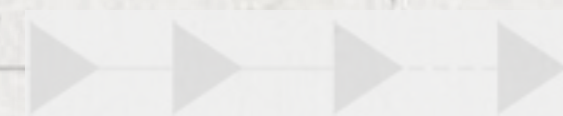
三、可综合风格

完整的always敏感信号列表

- 所有的组合逻辑或锁存的always结构必须有敏感信号列表。这个敏感信号列表必须包含所有的输入信号。
- 原因：综合过程将产生一个取决于除敏感列表中所有其它值的结构，它将可能在行为仿真和门级仿真间产生潜在的失配。

每个always敏感信号列表对应一个时钟

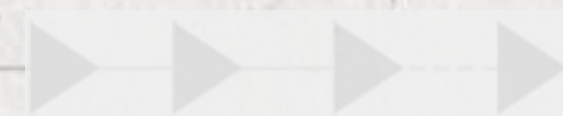
- 在综合过程中，每个Verilog always敏感信号列表只能对应一个时钟。
- 原因：这是将每一个过程限制在单一寄存器类型的要求。



不允许Wait 声明和# delay声明

- Wait 声明语句，不论是清楚还是含糊，都不能用于可综合设计。
- 原因：从RTL级转换到gate级的综合工具一般都不支持Wait 声明和# delay声明，为了有效的综合，这些语句应该避免。
- 例外：在不需要进行综合的行为模块中，如测试模块、表示行为的虚拟器件模块中可以使用。

在时序电路中必须使用非阻塞赋值(<=)
组合逻辑电路必须使用阻塞赋值(=)






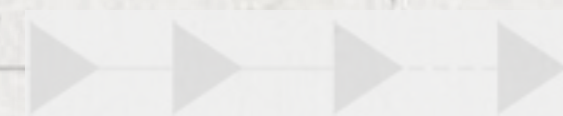
四、模块划分

分开异步逻辑与同步逻辑

- 建议分开异步逻辑与同步逻辑
- 原因：避免综合时的问题，简化约束和编码难度。
- 例外：不可应用于非综合模块中（例如：总线模块，总线监视器或是模拟模块）除非他们被设计来综合仿真。

分开控制逻辑和存储器

- 建议控制逻辑和存储器逻辑分成独立的模块
 - 原因：便于高层的存储器模块的使用和便于重新描述为不同的存储器类型。
- 



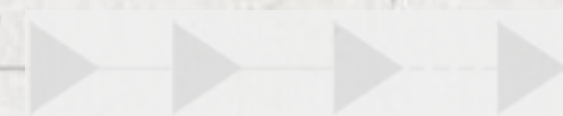


总结

初学者的误区： - verilog代码当做了程序，把电路设计当成了编程。

如何书写高性能verilog HDL：

- 牢记并理解可综合“四大法宝”所对应的硬件结构
- 写前，确认电路指标是什么：性能？面积？
- 硬件思维方式，代码不再是一行行的代码而是一块一块的硬件模块；
- 对所需实现的硬件电路“胸有成竹”；





The quality of optimization results depends on **how the HDL description is written.**

Don't rely on DC for timing !

—— 《Design Compiler® Reference Manual》

