# Verilog HDL

## 编码风格

1 输入输出端口的声明加\_i和\_o

2 真正的寄存器声明加后缀\_r

3 端口声明不要同时加reg声明

## 设计编码注意

1 端口信号的声明不能是数组类型

2 同样generate次数的循环尽可能放到同一个generate块里面，这样VCS调试时的层次结构会更简洁一些

3 用always写的组合逻辑一定要有所有条件不满足时的默认值，否则会生成未预期的锁存器逻辑

4 generate的循环控制变量默认是整型，不能直接在表达式中进行运算操作，否则会导致数组越界

5 if条件变量的条件必须是单bit的信号，不能是一个scalar变量

6 所有信号一定要有声明，即使是单bit的wire信号，这是比较好的编码习惯，否则DC会有大量warning

模块划分

## 模块划分

模块划分输出信号尽量是寄存器输出，模块输入尽量是从寄存器来，这样综合时易于delay\_in和delay\_out的设定，模块时序分析比较容易。另外，数据的bypass通路如果可以尽量放在模块内部，不要搬到其他模块。

## 非时钟沿的信号采样

在一个寄存器输出信号的上升或者下降沿对其他信号进行采样：

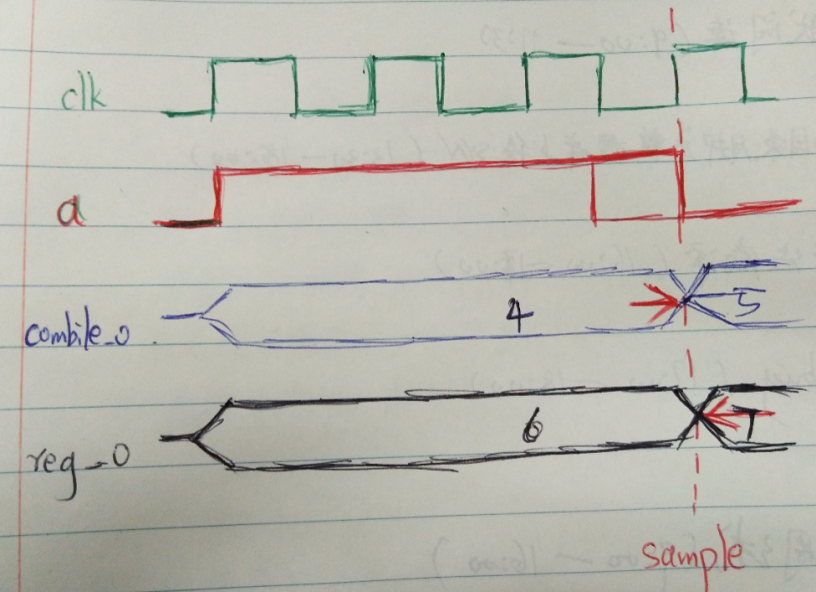


图 1.1 信号采样示意图

如上图所示，所有寄存器都使用时钟上升沿进行更新，a是一个寄存器输出信号，combile\_o是一个组合逻辑的输出信号，reg\_o是一个寄存器的输出信号，使用VCS进行仿真，在时刻“ @negedge a” 对combile\_o和reg\_o信号进行采样，combile\_o采样的是时钟延左边的值，而reg\_o采样的是时钟延右边的值。

可能原因：a的下降沿说明时钟上升沿已经过去了，所有其他寄存器的值也已经变了，因此reg\_o已经变成了7。而组合逻辑要等所有信号变化完成才变，所以采样时刻其值还未发生变化，因此combile\_o采的是未变化的左边的值。

如果想要采到reg变量时钟延左边的值，可以在环境里在利用时钟延把reg\_o的信号存起来，然后sample时刻使用保存之后的值。

## 基本块的取消逻辑

通过循环队列的头尾指针来对基本块的编号进行维护，当发生分支预测错误时如何产生要取消基本块的取消信号（block\_cancel）？

假设有8个基本块，基本块编号为3bit，最终产生8-bit的block\_cancel信号，每bit对应一个基本块。队列的尾指针为tail\_pt，队列的头指针为head\_pt，发生分支预测错误的基本块指针为error\_pt。分支指令是一个基本块结束的标志，所以当发生分支预测错误时，error\_pt本身所对应基本块的所有指令都是要保留的，即error\_pt之后到tail\_pt的所有基本块都要被cancel，而head\_pt到error\_pt所对应的基本块要被保留。

所以基本块的cancel产生逻辑分为精确取消和精确保留两种情况。精确取消就是只有有效且被取消基本块所对应的block\_cancel信号有效，而精确保留就是只有有效且被保留基本块所对应的block\_cancel信号无效。精确保留逻辑使用error\_pt和head\_pt产生，如表 1‑1所示；精确取消逻辑使用error\_pt和tail\_pt产生，如表 1‑2所示。

表 1‑1 block\_cancel的精确保留控制

|  |  |  |
| --- | --- | --- |
| 情景 | 取消基本块i的范围（block\_cancel[i]=1） | **保留基本块i的范围（block\_cancel[i]=0）** |
| head\_pt < error\_pt | <head\_pt or >error\_pt | **>=head\_pt and <=error\_pt** |
| head\_pt = error\_pt | <head\_pt or >error\_pt | **>=head\_pt and <=error\_pt** |
| head\_pt > error\_pt | <head\_pt and >error\_pt | **>=head\_pt or <=error\_pt** |

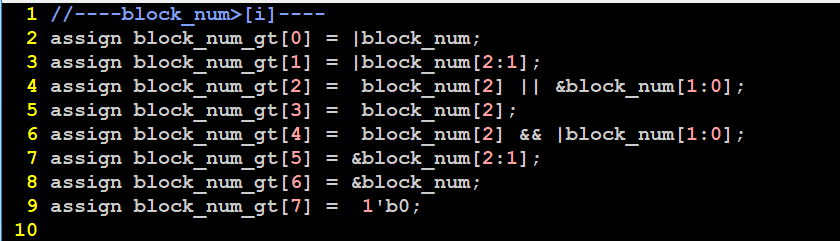
表 1‑2 block\_cancel的精确取消控制

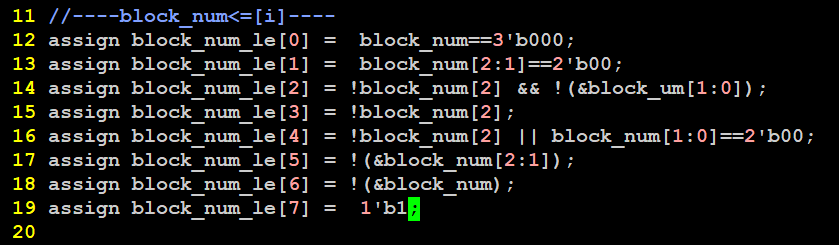
|  |  |  |
| --- | --- | --- |
| 情景 | **取消基本块i的范围（block\_cancel[i]=1）** | 保留基本块i的范围（block\_cancel[i]=0） |
| tail\_pt < error\_pt | **<=tail\_pt or >error\_pt** | >error\_pt and <=error\_pt |
| tail\_pt = error\_pt | **<=tail\_pt or >error\_pt** | >error\_pt and <=error\_pt |
| tail\_pt > error\_pt | **<=tail\_pt and >error\_pt** | >error\_pt or <=error\_pt |

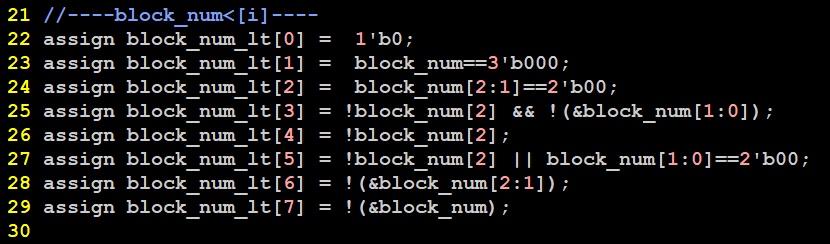
表 1‑3 小于和大于的逻辑标识

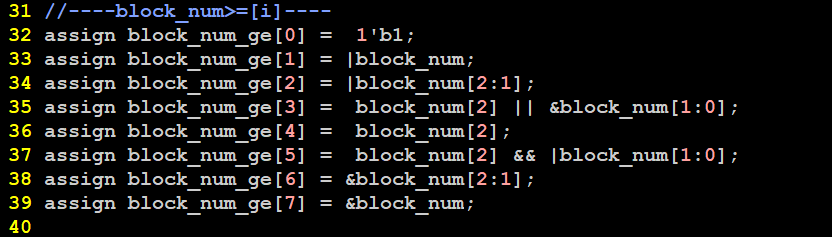


用block\_num\_lt[i]表示小于i的block\_num的值；block\_num\_ge[i]表示大于等于i的block\_num值；block\_num\_gt[i]表示大于i的block\_num值；block\_num\_le[i]表示小于等于i的block\_num值。表 1‑3是block\_num\_lt[i]和block\_num\_gt[i]的真值表。下面是四种情况的逻辑表达式：

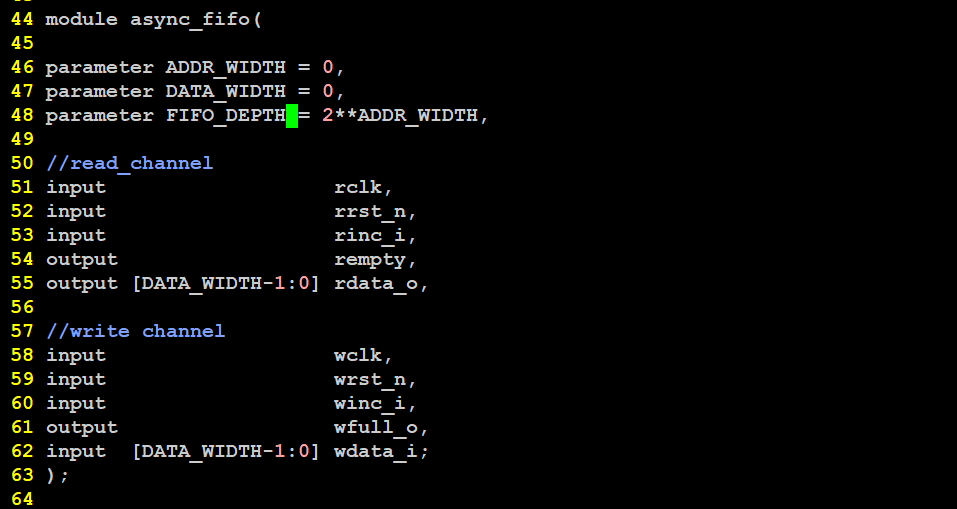


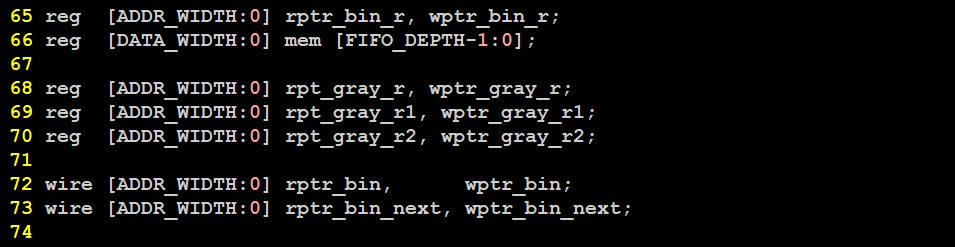


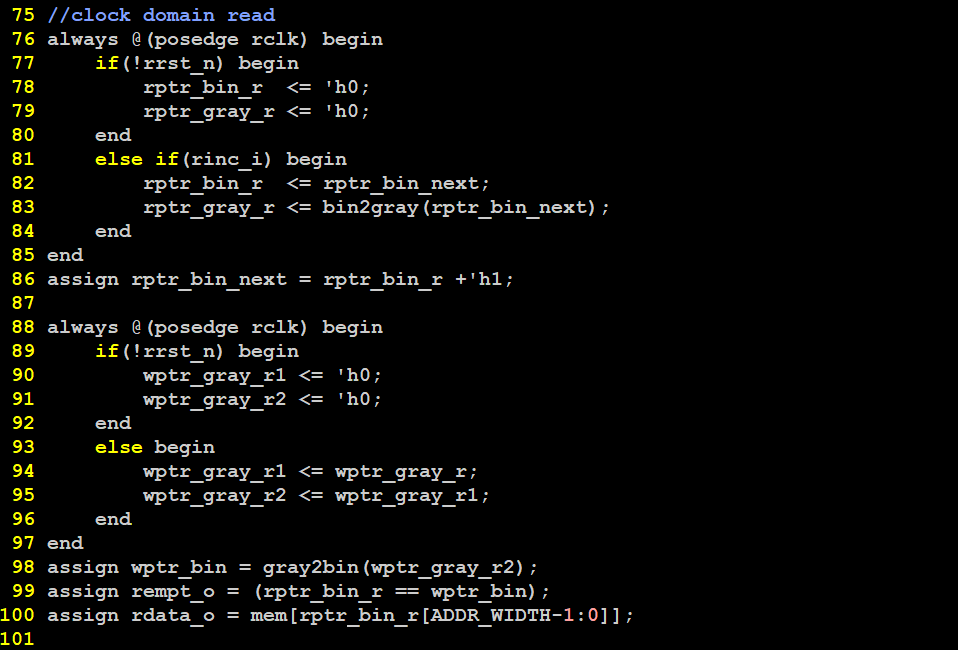


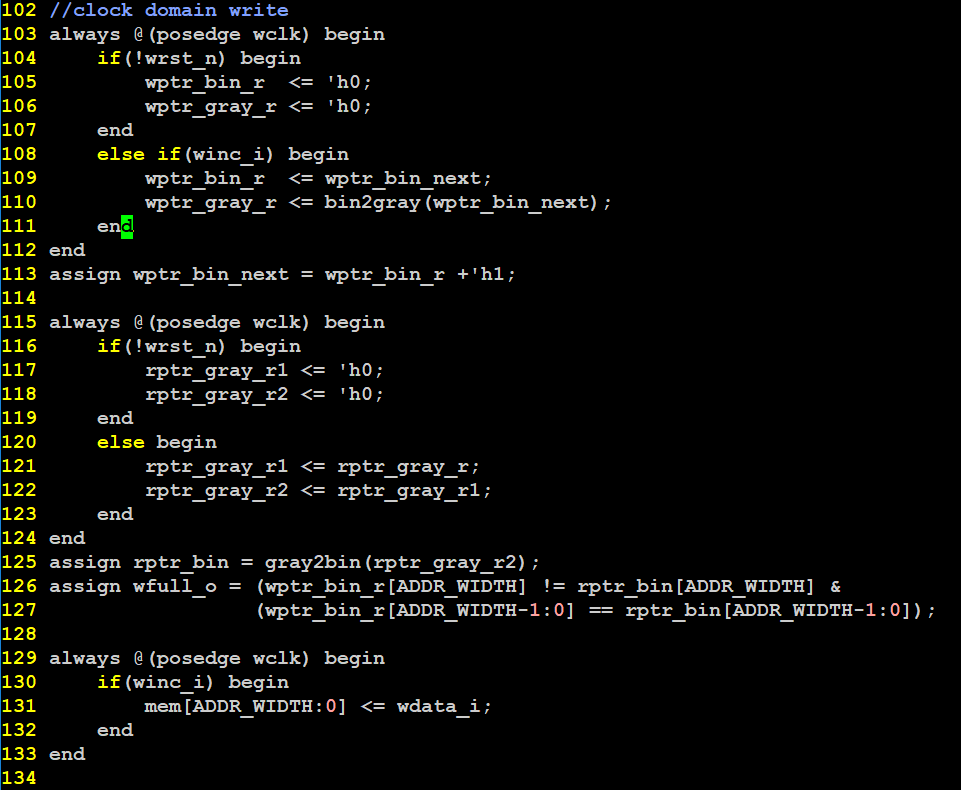


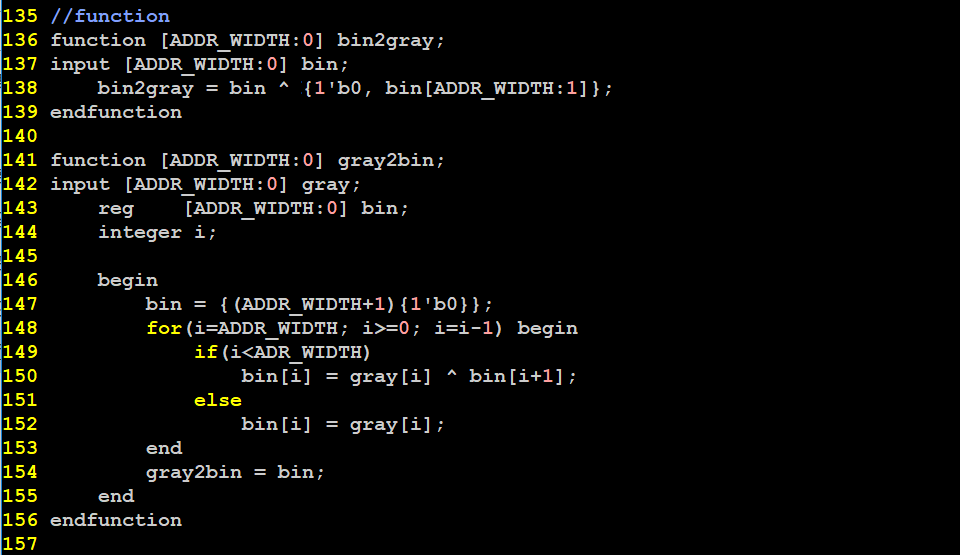
## 格雷码同步器







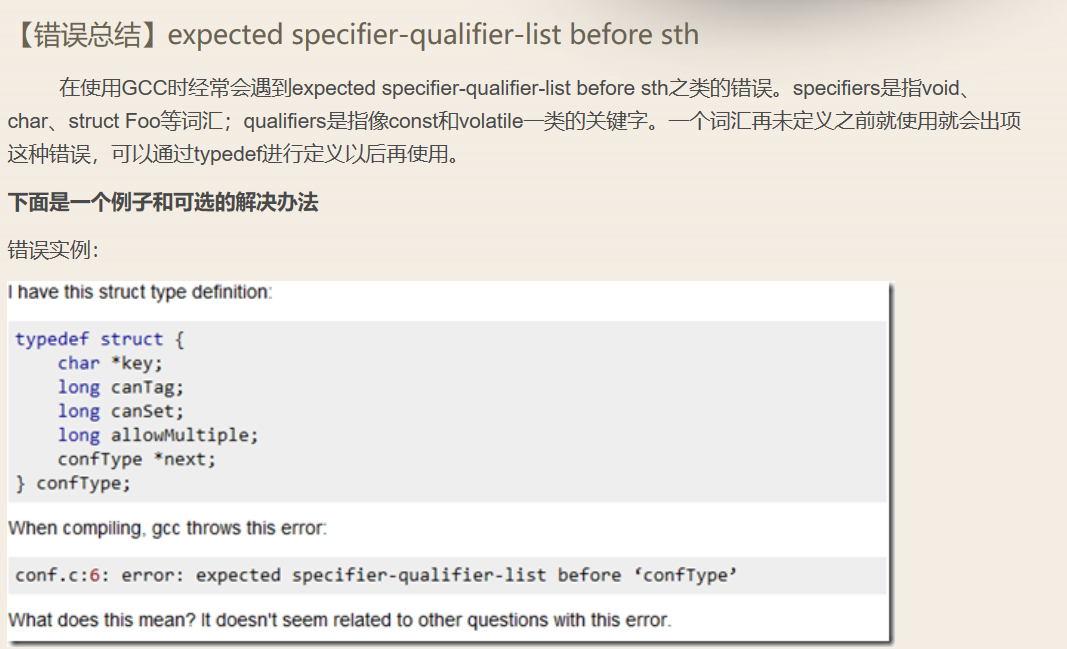


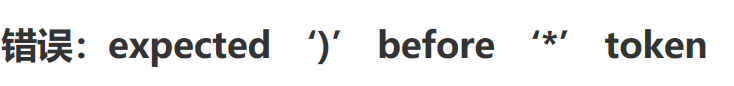




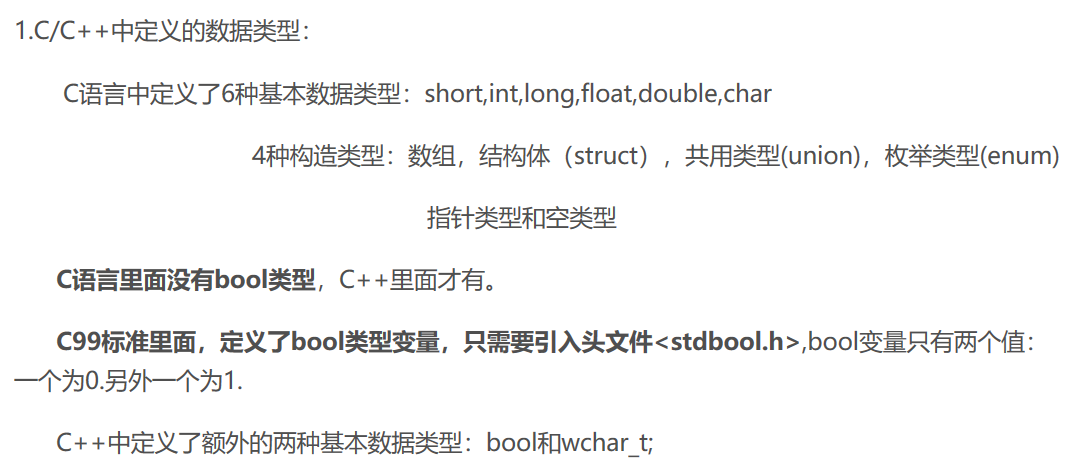
# C

## 结构体内变量类型必须定义过

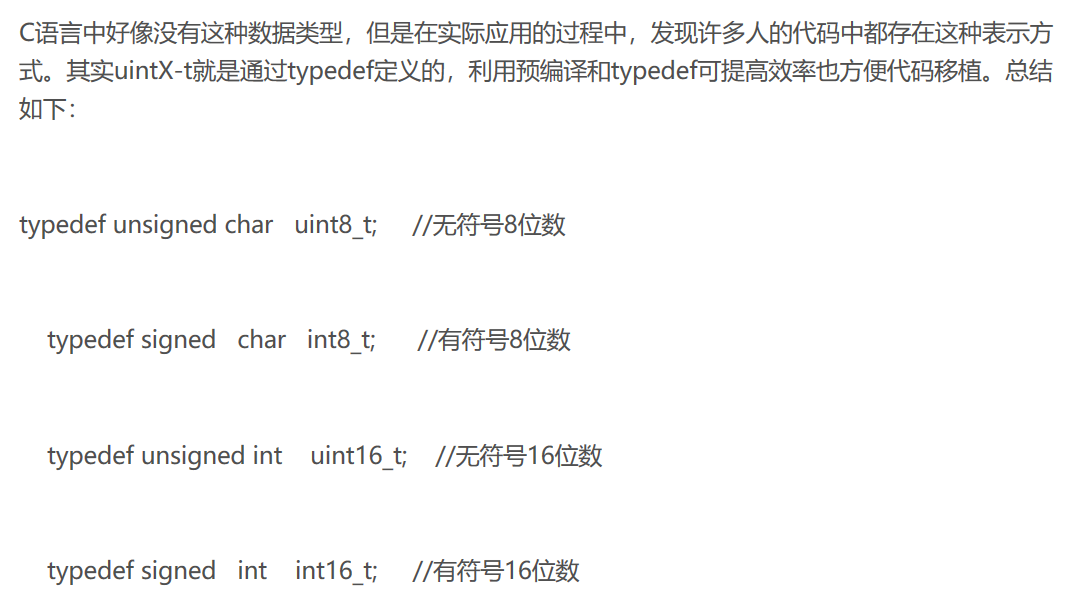


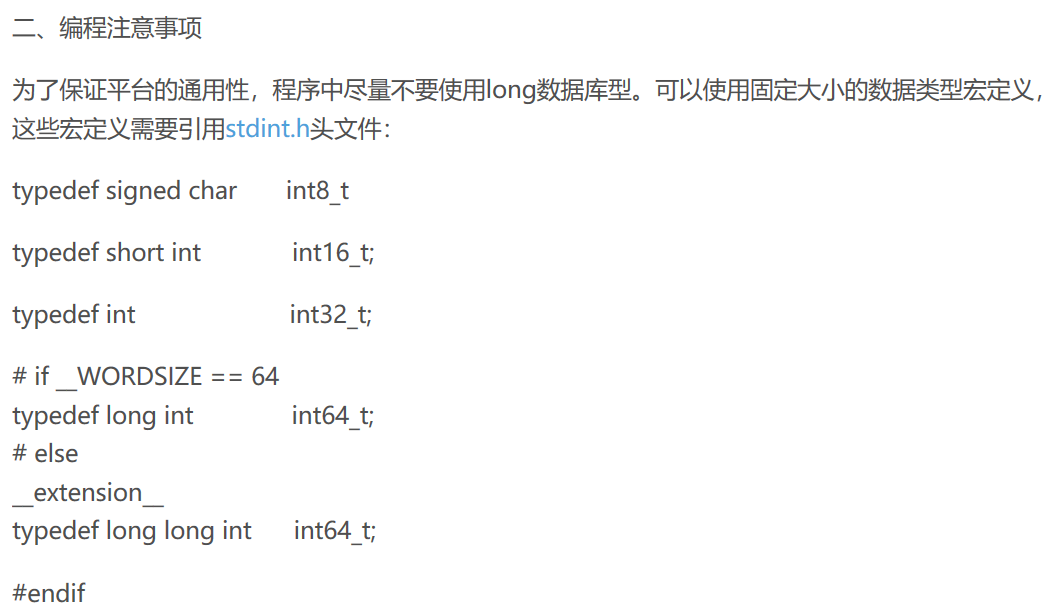


## Bool类型不是c语言的内置

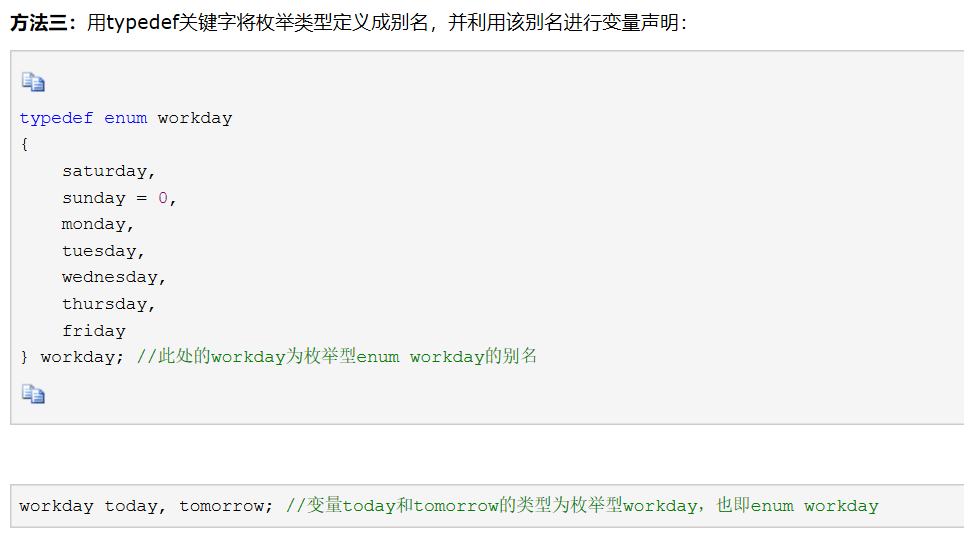


## 特殊数据类型定义

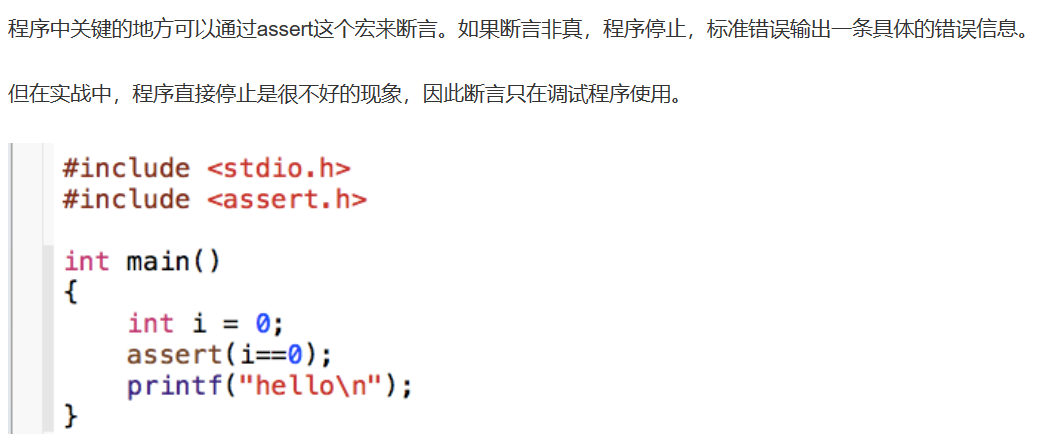




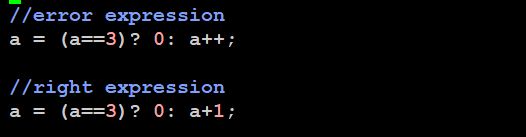
## 枚举类型定义



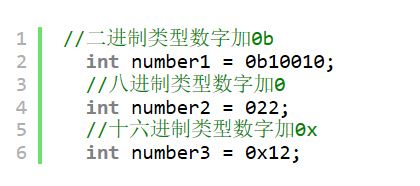
## 使用assert



## 条件表达式



## 不同进制书写方式

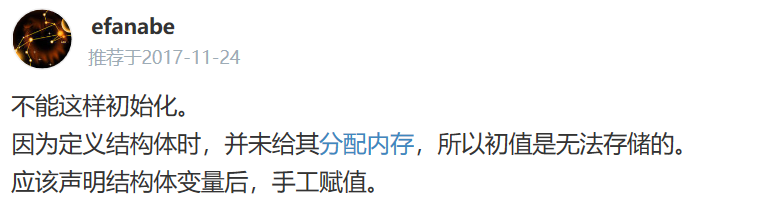


## 长整型打印



## 结构体成员变量初始化

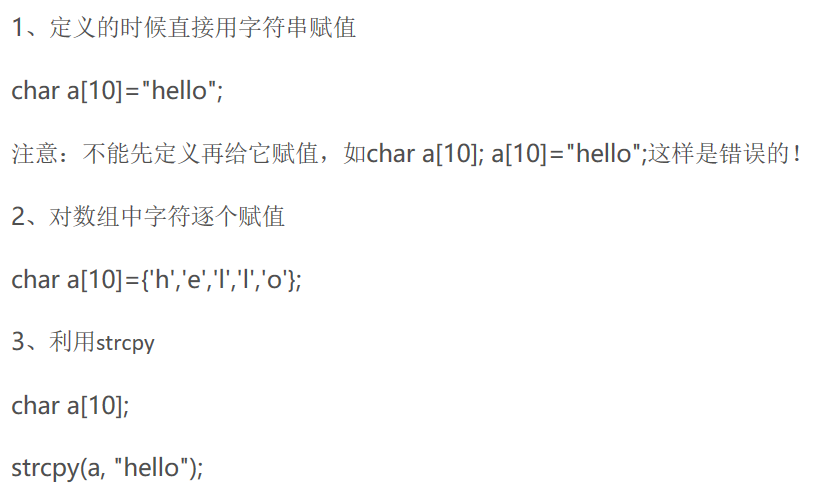
结构体定义的时候可以同时对变量初始化吗？比如没例化一个结构体变量都要将里面的一个成员变量赋0值，可不可以在结构体里直接给该成员变量一个默认值，每次例化时便自动赋值，不用人为初始化。



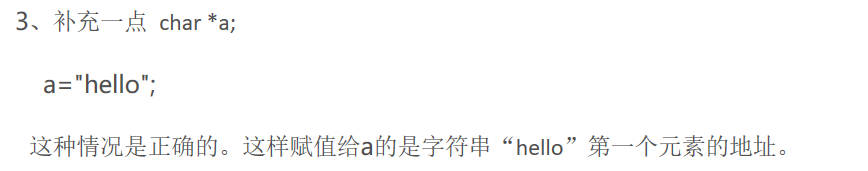
## 字符数组和字符指针赋值

@必须包含头文件 <string.h>

**字符数组**



**字符指针**



# VHDL

## 模块划分

模块划分输出信号尽量是寄存器输出，模块输入尽量是从寄存器来，这样综合时易于delay\_in和delay\_out的设定，模块时序分析比较容易。另外，数据的bypass通路如果可以尽量放在模块内部，不要搬到其他模块。

在没有timing风险的时候信号命名和划分怎么舒服怎么来。

## Compile Error

### Error: Poorly formed operand of type conversion



如何将generate循环控制变量转化为bit5\_type（std\_logic\_vector(4 downto 0）信号？

先用to\_unsigned转化为无符号整数，然后再转化为std\_logic\_vector.



### Error: Array index mismatch

数组的index都是integer类型，所以std\_logic\_vector信号要先转化成unsign， 然后转换成integer。



### Warning: component is not fully bound

Makefile lack file path declaration

### Error-Illegal prefix

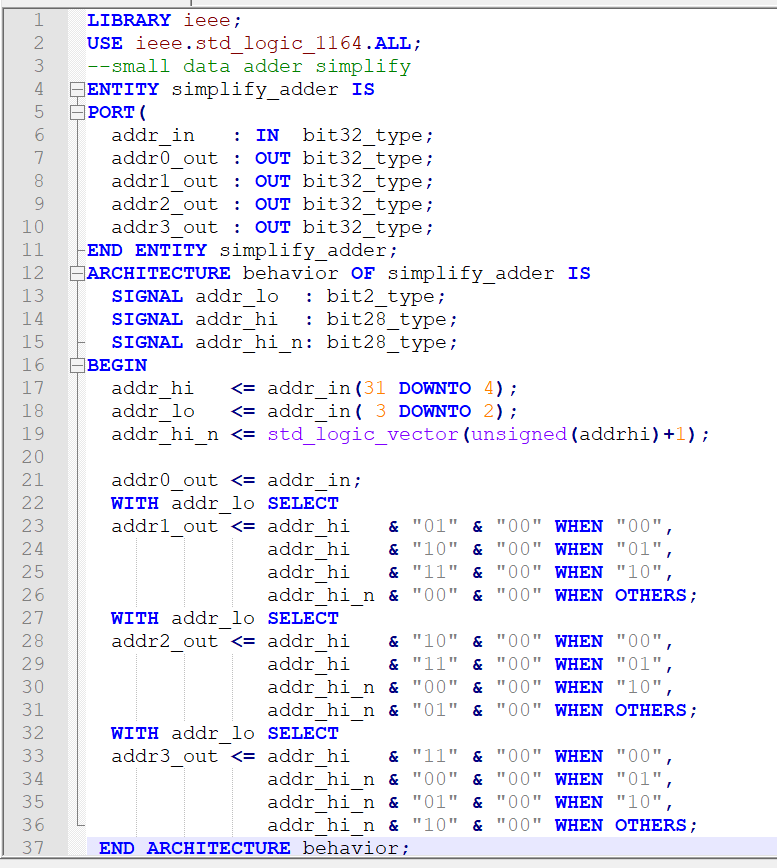
respq\_i.data(i) <= is data\_i OR is\_sync\_i;

Expand names are not aloowed for the named prefix. Please refer to section 6.3 of the VHDL LRM for rules regarding the expanded name.

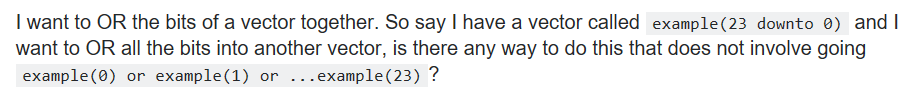
Respq是一个数组变量，每个元素都是一个record，record中包含data。选择第i个元素的data，应该用respq\_i(i).data而不是respq\_i.data(i).

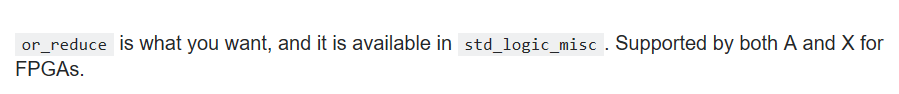
## 加法器简化

对于指令PC的顺序加，可以使用简化的加法器，实际只有一个PC加1的操作，其他通过位的拼接和多路选择器来做。

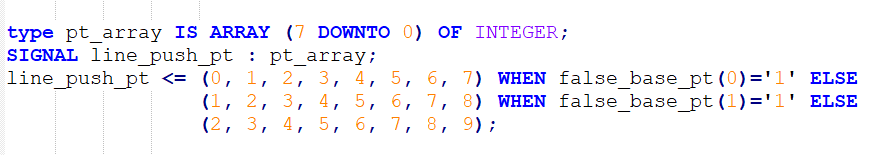


## 缩位运算符号





## 数组变量赋值



## 加减运算

* Std\_vector\_logic信号运算数必须确定是signed（符号位拓展）还是unsigned（0拓展），signed和unsigned只有拓展功能，没有截位功能，所以等式左边的信号位宽一定要大于等于右边任何一个信号的位宽。
* 只有相同类型的变量之间才能进行运算

## 移位运算

VHDL没有移位运算符（<<和>>），所以使用的是位拼接 & 来取代。

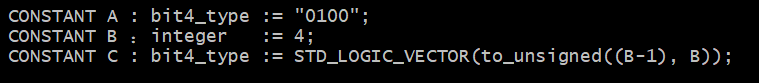
比如

SIGNAL a : bit8\_type;

Sl2\_a <= a(5 DOWNTO 0) & “00”; --a左移两位

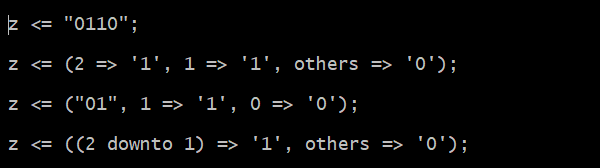
Rsl2\_a <= a(5 DOWNTO 0) & a(7 DOWNTO 6); --a循环左移两位

## 常量声明



注：to\_unsigned（data, width），两个参数都是integer类型，它用于将integer数转化为无符号整数，位宽由width指定；STD\_LOGIC\_VECTOR(unsign\_data)可以将一个无符号整数转化为Bit向量。

## 静态变量赋值



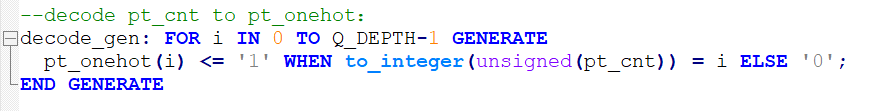
注：表示常量bit的index的数值必须是常量，不能是变量

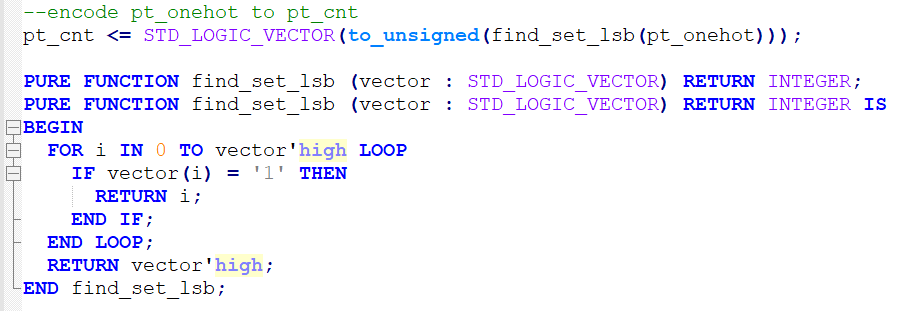
不能给某几位赋值一个数值串，只能赋值’0’或者‘1’, 比如下面的写法就是错误的：



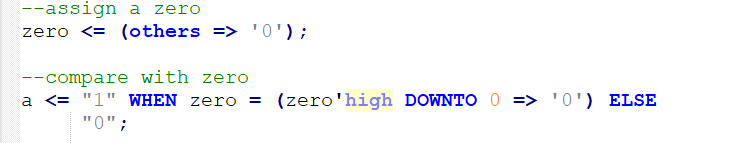
## 指针编码解码

队列的访问指针有两种形式，cnt和onehot模式，对于深度为Q\_DEPTH的queue, 不同形式指针之间的相互转化。

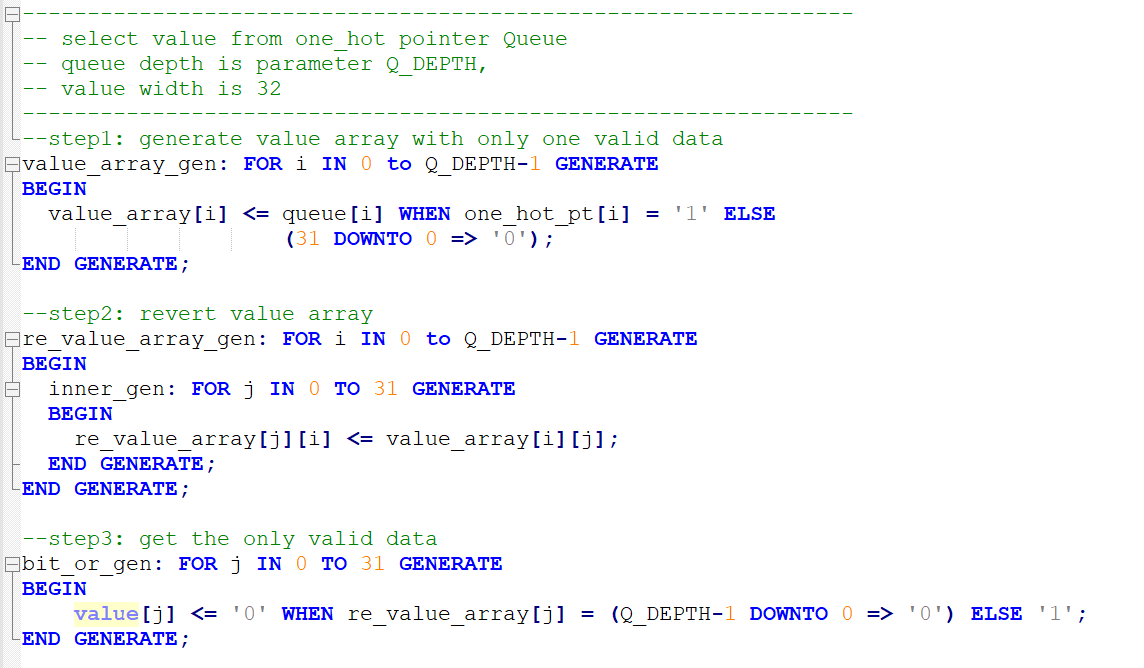




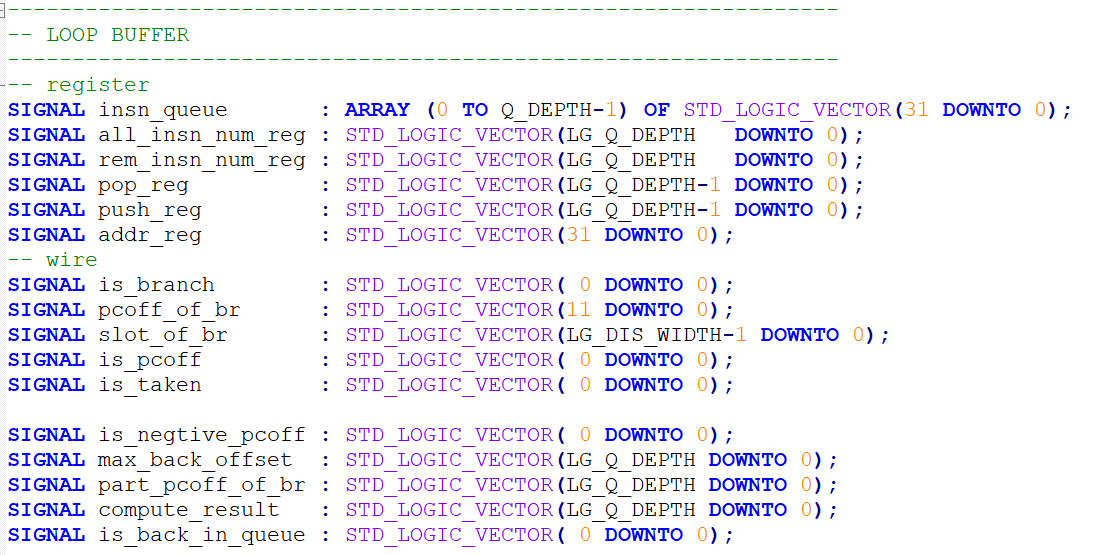
## 变量的零值

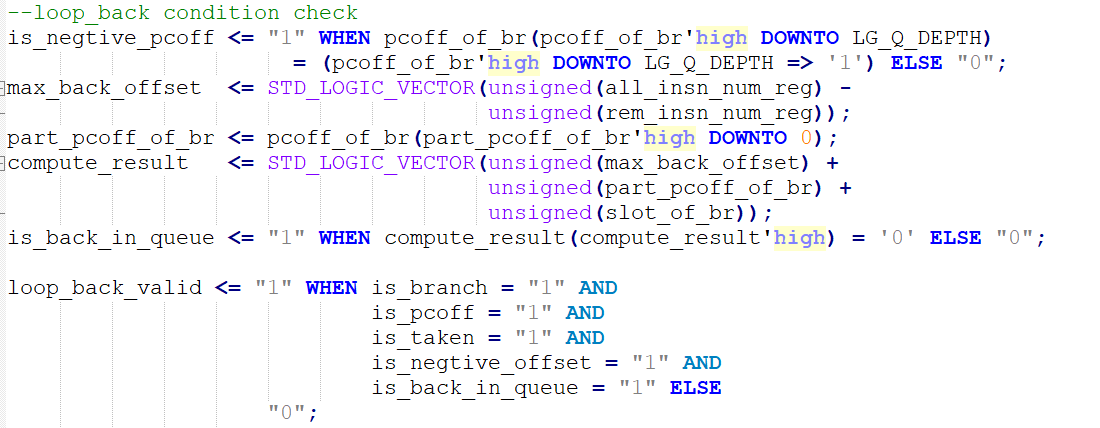


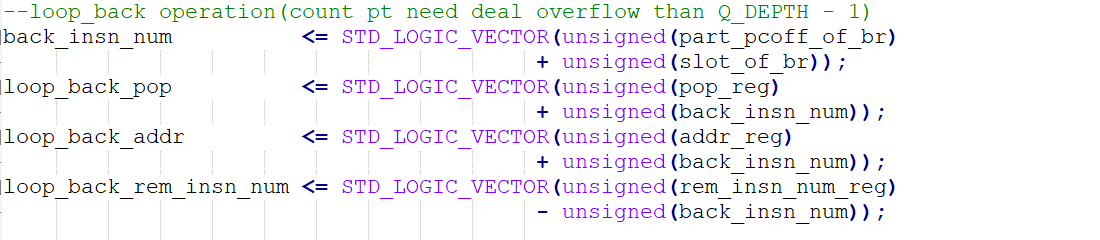
## 移位指针队列深度参数化



## Loop buffer







Unsign：0拓展; sign：符号位拓展

## Repeat instruction



**Repeat dispatch拍的检查**

* $counter = 0：不执行repeat\_sequence，repeat当作一条jump来执行
* $counter = 1：repeat\_sequence只执行一次， repeat当作一条nop来执行
* $insn\_num = 0：repeat\_sequnce内的指令数为0， 没有要repeat的指令，repeat当作一条nop来执行
* 其他情况走正常的repeat处理逻辑

**Repeat的处理逻辑**

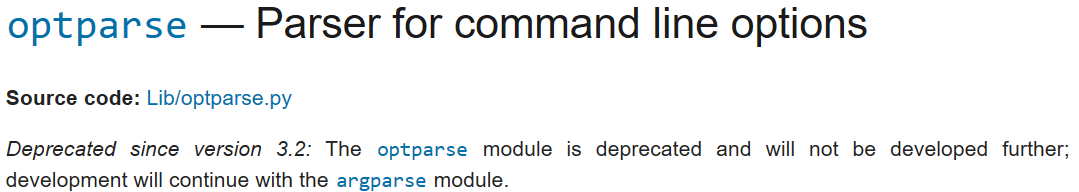
$insn\_num < rem\_insn\_num\_reg



* Rem\_insn\_num\_reg 设置为$insn\_num
* Push\_reg相应前移
* Stop\_fetch\_reg置位，暂停发送取指请求，且不接收取指响应
* Repeat\_mode\_reg置位
* Repeat\_addr\_reg设置为repeat下一条指令的PC
* Repeat\_all\_insn\_num\_reg设置为$insn\_num的值
* Repeat\_counter\_reg设置为$counter的值
* Repeat\_rem\_insn\_num\_reg设置为$insn\_num的值
* 之后指令顺序派遣，直到pop\_insn = rem\_insn, 且repeat\_counter\_reg>1, 一轮repeat指令派遣结束，repeat\_counter\_reg--；repeat\_rem\_insn\_reg = repeat\_all\_insn\_num\_reg; rem\_insn\_num\_reg = repeat\_all\_insn\_num\_reg
* 当repeat\_counter=1，rem\_insn\_num\_reg=0, alive\_load\_reg = 0, 将stop\_fetch\_reg置为0(repeat\_sequence全部发完，且不会回退)
* 当repeat\_counter=1，rem\_insn\_num\_reg=0,alive\_load\_reg = 0, 将repeat\_mode\_reg置0. (repeat\_sequence全部发完，且不会回退)

# Python

## optparse



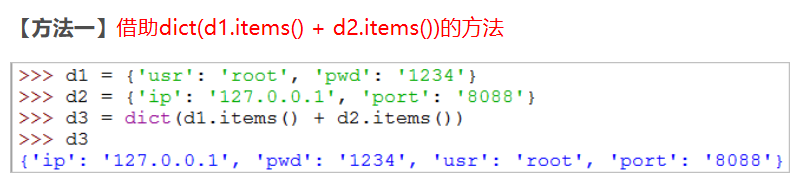
<https://docs.python.org/3/library/optparse.html>

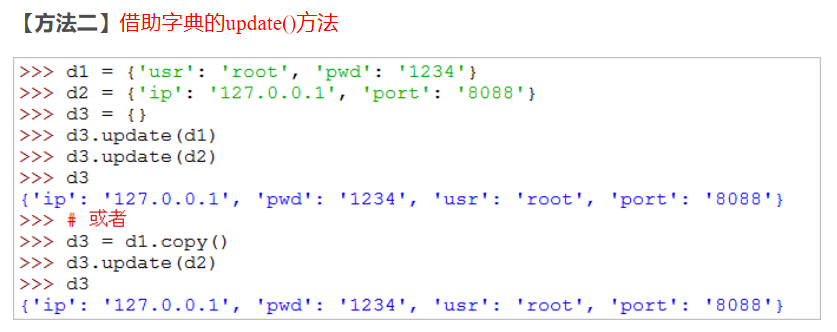
## 列表

* 列表可以包含任何种类的对象：数字，字符串甚至其他列表。
* 由于列表的每一项都是有序的，你可以执行诸如分片和拼接之类的任务。
* 列表拼接：L = L1 + L2

## 字典

### 字典合并



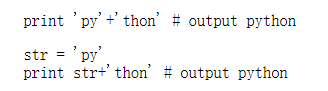




## 传参



## 字符串变量和常量的拼接

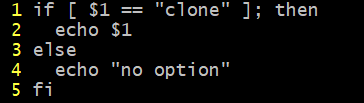


# Shell

## Error

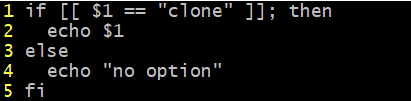


原来写法：

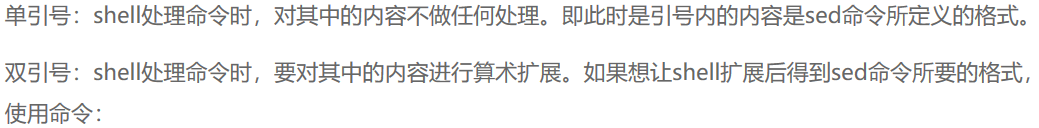


当 执行脚本没有传参的时候， $1为空， 相当于 if [$1 == ]， 所以报错

改为：



## 引号作用



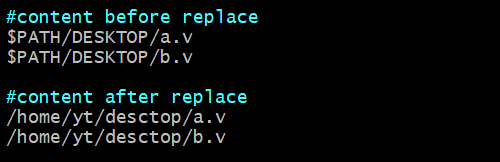




## 文件内容变量替换



文件中内容的变化



# Common

## ASSIC字符表



## Unaligned Q pointer

Overflow need subtract Q number;

Underflow need subtract (2n-Q number);

## 循环移位逻辑

Q：data =8’b11100000, 将之循环左移n位，求c?

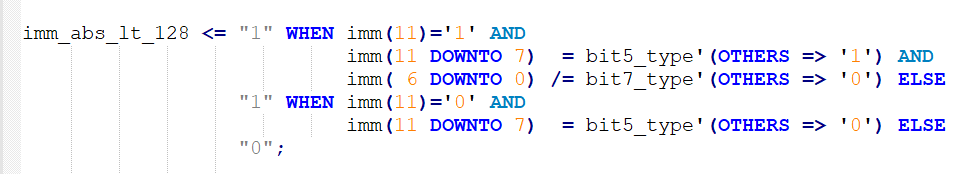
A: a = data << n; //将data逻辑左移n位， 低位补0，高位丢弃

b = data >> (8-n) //将data逻辑右移（8-n）位， 高位补0，低位丢弃

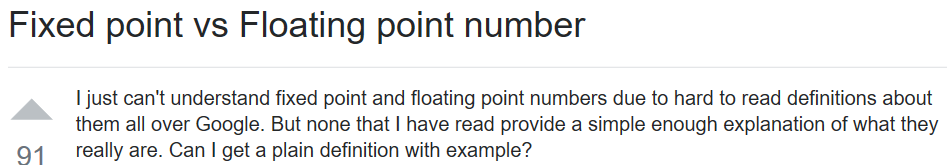
c = a | b; //将 a 和 b 按位或，得到data循环左移n位的值。

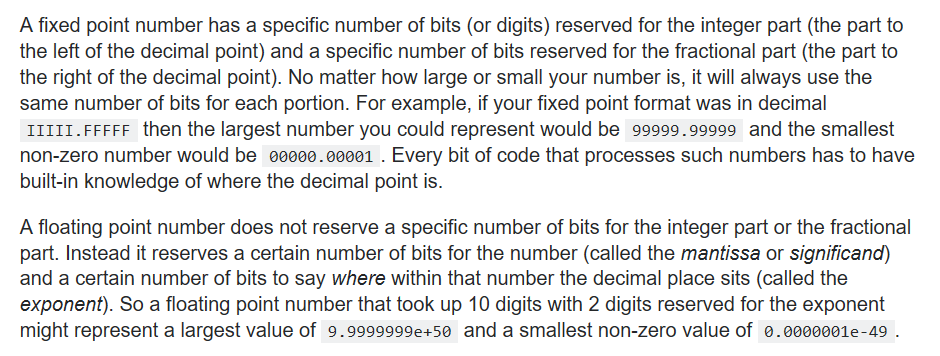
## 立即数范围判断

判断一个2的补码表示的12-bit立即数的绝对值是否小于128：



## 定点小数





## Queue的顺序选择逻辑

**14-entry instruction queue，choose 3 sequential instruction**

对所有元素进行分组重新排序，要顺序选N个元素就分成N组，这样一次可以选出所有元素，从而可以节约MUX资源。然后对已经选出的元素进行顺序调整，就可以获取我们需要的顺序N个元素。

因此将14条指令分成3组，每组5条指令（地址越界后就顺序回绕），这样选出我们需要的3条指令需要5级18个mux。这样子虽然节约了mux资源，但是比直接的14选1逻辑多了一级mux，时序恶化了。



换个方式，我们可以从14条指令中选择4条指令出来，但是只使用3条。将14条指令分成4组，每组4条指令（地址越界后就顺序回绕），这样选出我们需要的3条指令需要4级21个MUX。



# AMBA协议





AHB主要是针对高效率、高频宽及快速系统模块所设计的总线，它可以连接如微处理器、芯片上或芯片外的内存模块和DMA等高效率模块



APB主要用在低速且低功率的外围，可针对外围设备作功率消耗及复杂接口的最佳化。APB在AHB和低带宽的外围设备之间提供了通信的桥梁，所以APB是AHB或ASB的二级拓展总线。



AXI：高速度、高带宽，管道化互联，单向通道，只需要首地址，读写并行，支持乱序，支持非对齐操作，有效支持初始延迟较高的外设，连线非常多。