­­

****

实验三、带有保护间隔的固定优先级排队

2020年 月 日 （版本v02）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **序号** | **组长 标记** | | **教学班** | **学号** | **姓名** | **签名 （无手签无效）** |
| **1** | **\*** | |  |  | **张三** |  |
| **2** |  | |  |  | **李四** |  |
| **3** |  | |  |  | **王二** |  |
| **4** |  | |  |  | **刘五** |  |
|  |  | |  |  |  |  |
| **提交 日期** | **2020- -** | | | **签收：** | **评价日期** |  |
|  |
| **评阅1** | | **评阅2** | | **评阅3** | **评阅4** | **平均（百分制）** |
|  | |  | |  |  |  |

**XXXXXX学院**

目录

[实验三、带有保护间隔的固定优先级排队（实验指导书部分） 3](#_Toc57915702)

[1 实验指导 3](#_Toc57915703)

[1.1 背景知识 3](#_Toc57915704)

[1.1.1 混合关键性虚拟链路排队 4](#_Toc57915705)

[1.1.2 协议硬件实现的工程化设计 4](#_Toc57915706)

[1.1.3 编译开关的使用 6](#_Toc57915707)

[1.2 实验内容 7](#_Toc57915708)

[1.2.1 有限状态机 7](#_Toc57915709)

[1.2.2 握手信号的时序 8](#_Toc57915710)

[1.2.3 用户显示的设计 9](#_Toc57915711)

[1.3 基本要求 10](#_Toc57915712)

[1.4扩展要求 11](#_Toc57915713)

[1.5附加说明 11](#_Toc57915714)

[实验三、带有保护间隔的固定优先级排队（实验报告部分） 13](#_Toc57915715)

[2 实验报告 13](#_Toc57915716)

[2.1 实验背景与需求分析 13](#_Toc57915717)

[2.2 系统设计 13](#_Toc57915718)

[2.2.1 总体设计思路 13](#_Toc57915719)

[2.2.2 接口设计 13](#_Toc57915720)

[2.2.3 XXX模块 13](#_Toc57915721)

[2.2.4 YYY模块 13](#_Toc57915722)

[2.3 功能仿真测试 14](#_Toc57915723)

[2.3.1 测试程序设计 14](#_Toc57915724)

[2.3.2 功能仿真过程 14](#_Toc57915725)

[2.3.2 实验关键结果及其解释 14](#_Toc57915726)

[2.4 设计实现 14](#_Toc57915727)

[2.4.1 综合和下载过程 14](#_Toc57915728)

[2.4.2 实验关键结果及其解释 14](#_Toc57915729)

[2.5 小结 14](#_Toc57915730)

[参考文献 14](#_Toc57915731)

# 实验三、带有保护间隔的固定优先级排队（实验指导书部分）

## 实验指导

通过一个带有保护间隔的固定优先级排队案例，利用JC-FOH-V2.0实验硬件实验平台（以下简称“实验板”）通过Verilog HDL语言实现可综合代码，加深对时序逻辑设计理念的认识。

### 背景知识

数据网络通信是信息互联的基础设施，根据网络协议，交换机或转发设备对不同的数据流提供不同资源和实时性保证的服务。

对于高速网络，采用嵌入的计算机执行软件算法是不现实的，而且成本也过高，需要将网络协议“扁平化”，即采用可以高速并发执行的硬件实现。

在网络的交换或转发设备上，可以进行固定优先级排队（fixed-priority queuing，FPQ），网络的数据包往往是不可抢占的（non-preemptive），即：数据包必须完整地被传输，高优先级的数据包也无法阻止正在接受服务的低优先级的数据包。

考虑时间触发流量（TT）和事件触发流量的混合多路复用，TT流量享有固定的时隙和高优先级，多条事件触发流量具有相同的低优先级，事件触发流量之间进行轮询（round-robin）调度。

在TT流量的时隙之前，要留有保护间隔（margin），以防止低优先级数据包的不可抢占性阻塞本应在时隙内严格确定性传输的TT流量数据包。但是，如果保护间隔阻止一切低优先级数据包，又会造成很大的浪费，特别是对于变长的数据包，例如：以太网最短数据包64Byte，而最长帧的长度1518Byte（该长度又被称为MTU），如果设保护间隔长度为MTU，则可容纳很多短一些的数据包。



1. 保护间隔

理想的解决方案是根据当前剩余的保护间隔的时间长度，允许一些对TT流量无影响的低优先级数据包传输，如图 1所示。

适合数字电路实现的思路是：为保护间隔设置硬件计时器，随运行时间减计数，低优先级服务器将当前头部数据包的长度与计数器的残余值比较，如果前者较小，则设置一个标记，如bool\_ready，多路复用器根据当前的工作状态，以及各队列的标记确定转发哪个头部数据包。

#### 混合关键性虚拟链路排队

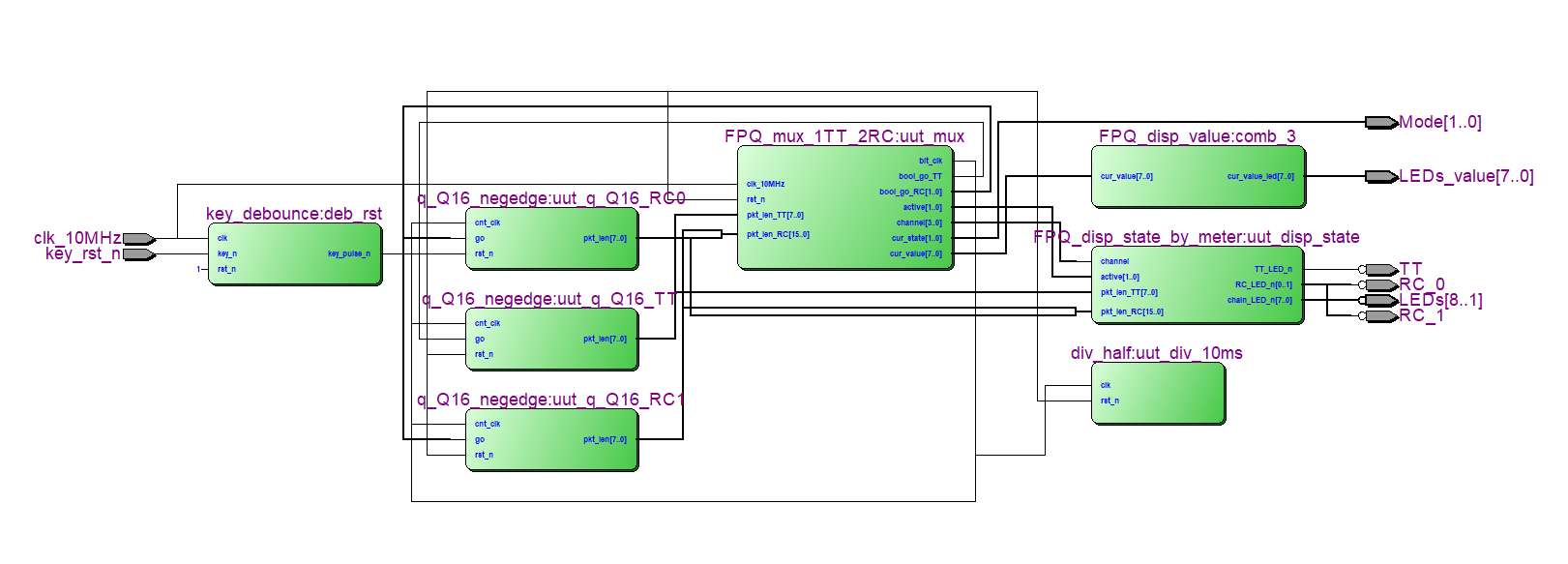
本项实验的背景来源于SAE AS6802标准定义的时间触发以太网（TTE），该网络可以容纳多条带有实时服务质量要求的流量（被称为“虚拟链路”，缩写“VL”）。其中，每个综合循环（IC）周期包含时间触发的PCF帧（网络协议开销，优先级最高，在IC起始出现）、TT流量和事件触发的速率约束（RC）虚拟链路。其中TT流量根据TDMA时间调度表组织通信。

本实验中考虑TT流量和两条RC VL，在TT时隙之间的“孔”中，两条RC VL进行轮询排队。

#### 协议硬件实现的工程化设计

硬件描述语言以“模块”为基本的实体，与普通计算机语言的顺序执行不同，合理的模块设计划分和综合是优质的工程化实现的保证，设计者要反复权衡相应的功能分配到哪些模块的实例更便于理解且更便于调试，使得这些模块实例的并发执行时信号配合关系正确且简洁。

参考设计的模块图如图 2所示。



1. 功能模块图

良好的设计习惯也是优质的工程化实现的必要条件，在编写代码之前撰写文档，是应对复杂性的过程控制方法，初步的文档可以从模块输入输出表格开始。例如：对于cur\_value\_translator模块的输入输出说明如下。

1. 模块的输入输出端口

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 端口 | 名称 | 方向 | 宽度 | 说明 |
| cur\_value | 时刻表 当前值 | input | [7:0] | 来自timetable模块的实例，是在所处时隙（或“孔”）内，减法计数器的剩余值。 |
| cur\_state | 时刻表 当前状态 | input | [1:0] | 来自timetable模块的实例，是在所处时隙（或“孔”）的类型，枚举类型，包含  T\_PCF (编码10)、T\_TT (编码11)、T\_MAR (编码01)、T\_DEF (编码00)；其中T\_MAR为保护间隔，T\_DEF是可以发送RC数据包的“默认”类型，实际是PCF、TT等时间触发时隙之间的“孔” 。 |
| cnt\_for\_TT | TT流量的计数器剩余值 | output | [7:0] | 根据cur\_value和cur\_state，面向TT流量转译的计数器剩余值；在T\_TT类型的情况，为cur\_value当时的值，除了在T\_TT类型之外，其它情况下值为0。 |
| cnt\_for\_RC | RC流量的计数器剩余值 | output | [7:0] | 根据cur\_value和cur\_state，面向TT流量转译的计数器剩余值；在T\_PCF和T\_TT类型，值为0，在T\_DEF类型值为95（对应于MTU长度），在T\_MAR类型的情况为cur\_value当时的值；值得注意的是，由于有保护间隔，所以在TT\_DEF类型的情况可以放任所有不长于MTU的数据包发送。 |

建议对于其它模块和实例，在设计阶段也辅助以相应的文档。

另外，可以采用软件工程的方法，使设计的高层需求和底层需求相互联系，并覆盖代码的功能。例如：“泳道”（lane）型的时序图（sequence diagram）可以用于描述到达队列、服务器和多路复用器之间的操作关系，如图 3所示；在这种描述方法下，代码实现有的放矢，而且真正的实际工程设计的用例（use case）往往包含这样若干的时序图，且有一些软件工程工具可以根据用例或时序图自动生成测试案例。



1. 队列、服务器、多路复用器之间的时序图

#### 编译开关的使用

为了便于观测，本实验中发送数据包的时间在几秒到几分钟量级，但在软件仿真测试中，会有很长一段均是持续稳定的数据，而实际上调试中关心的是状态变化时的逻辑。

为了提高效率，设置编译开关，具体作法是`ifndef JUST\_FOR\_SIMU (代码段1)`else（代码段2）`endif，其中JUST\_FOR\_SIMU为宏定义。在功能仿真中定义这个宏，则编译器综合“代码段2”中的代码；真正硬件实现时将宏定义行注释掉，再次编译，则综合“代码段1”中的代码。

### 实验内容

#### 有限状态机

协议的硬件实现离不开有限状态机（finite state machine，FSM），而FSM与数字电路分析设计中的状态转换图实质上是一样的。

在Verilog HDL语言中实现FSM的风格有一段式、二段式和三段式，这些内容是值得很好地自学并掌握的。

图 4给出了二状态q\_server的有限状态机的代码，尽管这种实现有缺陷（见1.2.2小节的讨论），但体现了一种“三段式”的风格。

当然，FSM的实现可以依据实际情况和设计者的经验，对于比较简单的状态机，也可以使用“一段式”描述。

1. 二状态q\_server模块的有限状态机代码（片断）

|  |  |
| --- | --- |
| 行号 | 二状态q\_server模块中有限状态机行为描述代码 |
| 1  10  20  30 | localparam S\_STOP = 1'b0,  S\_GO = 1'b1 ;    // Internal state variables  reg state;  reg next\_state;    always @( cur\_value or pkt\_len )  if ( cur\_value >= pkt\_len ) bool\_ready = 1'b1 ;  else bool\_ready = 1'b0 ;    // State changes only at pos edge of clock  always @( posedge clk or negedge rst\_n )  if ( !rst\_n ) state <= S\_STOP ;  else state <= next\_state;    //State machine using case statements  always @( state or ena\_n or pkt\_len )  case ( state )  S\_STOP :  if ( !ena\_n ) next\_state = S\_GO ;  else next\_state = S\_STOP ;  S\_GO :  if ( ena\_n ) next\_state = S\_STOP ;  else if ( pkt\_len == 0 ) next\_state = S\_STOP ;  else next\_state = S\_GO ;  endcase    //Compute values  always @( state )  case ( state )  S\_STOP : bool\_go = 1'b0 ;  S\_GO : bool\_go = 1'b1 ;  endcase |

同理，关键的round\_robin\_FP功能模块可以采用如图 5所示的状态机建模。

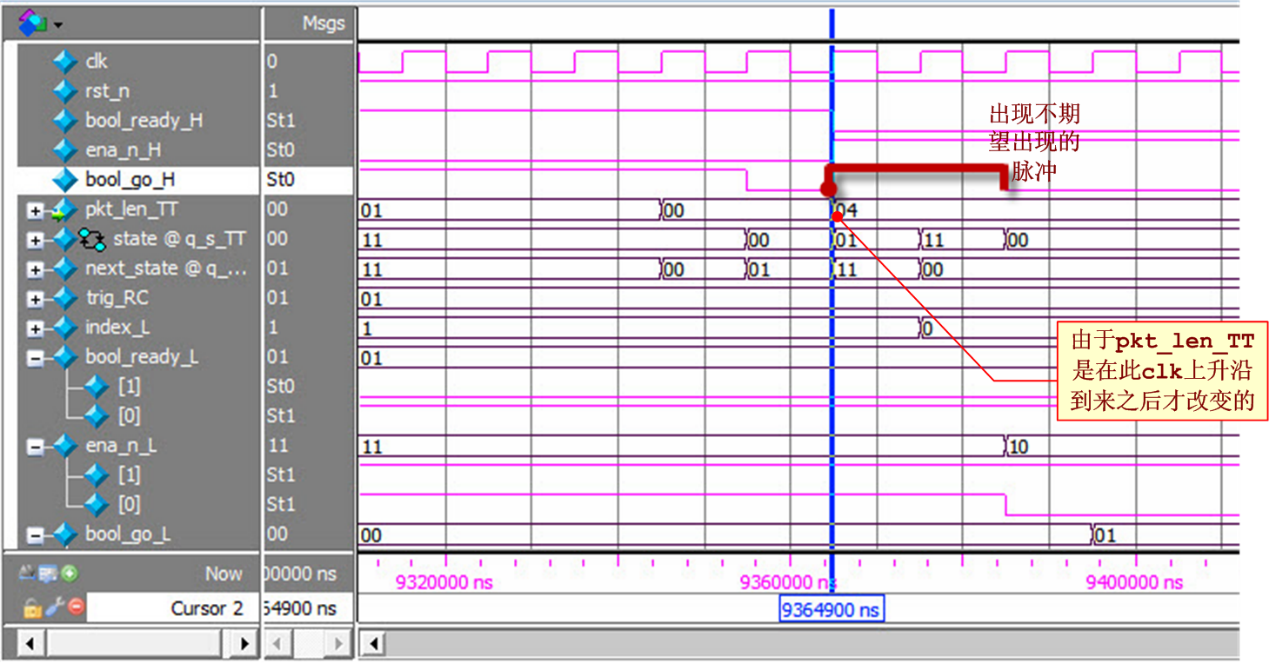


1. round\_robin\_FP模块的状态机

#### 握手信号的时序

在图 3中，最主要的信号握手关系是bool\_ready\_H（或bool\_ready\_L[i] ）🡺ena\_n\_H（或ena\_n\_L[i]）🡺bool\_go\_H（或bool\_go\_L[i]）。由于握手涉及到到达队列服务器和多路复用器，实质上还涉及到队列头部数据包长度信息的读取，时序细节上是不得不认真检查调试的部分。

当采用如图 4所示的服务器代码时，测试发现会出现类似于图 6所示波形反映出来的异常问题（对应于测试程序FPQ\_test7.v）。



1. 微小的差别导致系统工作异常

图中展示的是q\_server处理完一帧数据包后的情况，以TT流量为例，按照预期，①当pkt\_len\_TT==0时，q\_server使bool\_go\_H为FALSE；②而round\_robin\_FP观察到bool\_go\_H的变化使ena\_n\_H为无效；③同时q\_Q16观测到bool\_go\_H的变化将新的数据包放置到队列头部；④如果此时pkt\_len\_TT大于由timetable转译的cnt\_for\_TT的值，则q\_server使bool\_ready\_H为FALSE，使得暂时不会使下一轮的ena\_n\_H有效，直到获得足够的时隙剩余计数值。

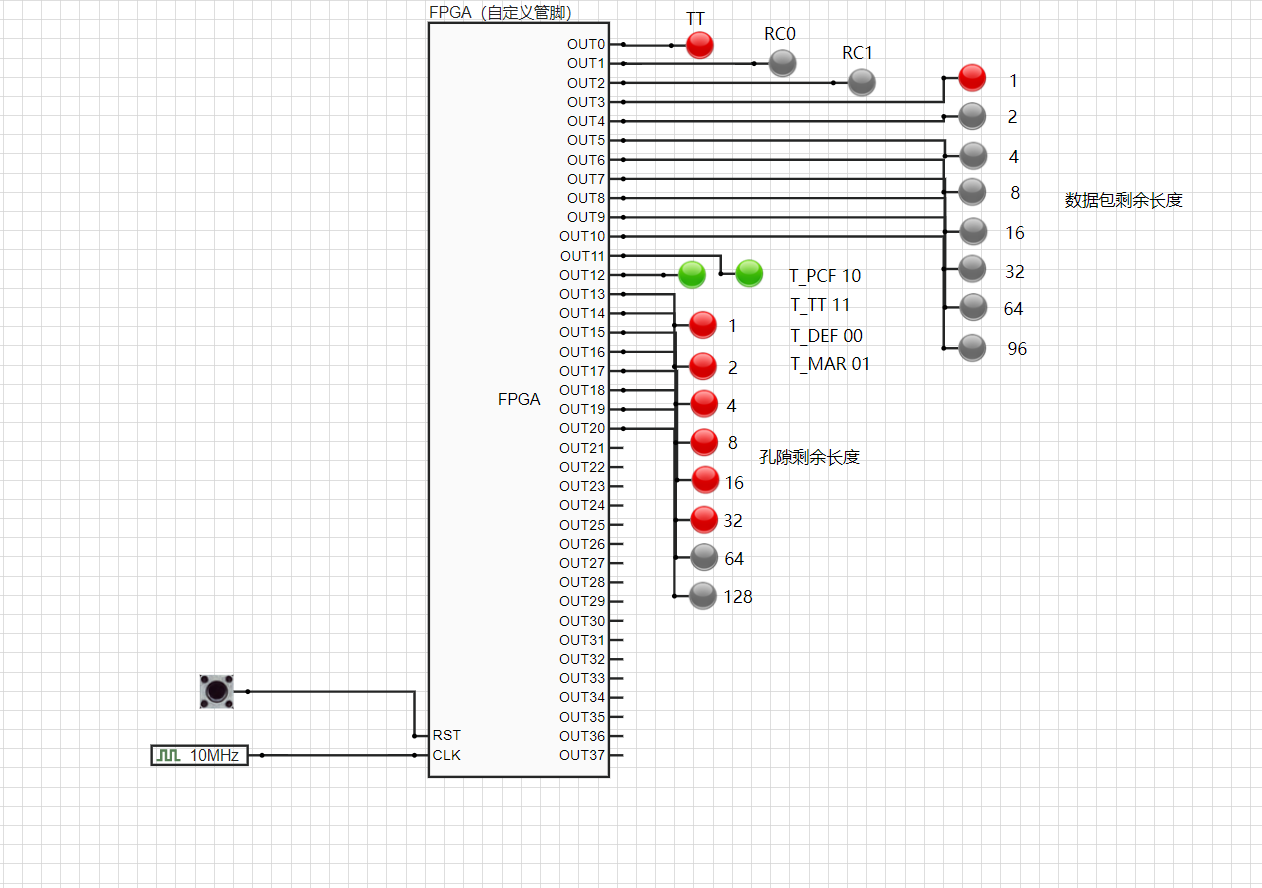
值得注意的是，设计中期望中②和③同时发生的，但是当③迟于②，更进一步则③和④存在先后的因果关系；当③随着上升沿发生改变，而④实际上在上升沿之后改变（尽管以图中的比例尺，观察不出来）。

这样，在这个关键的上升沿时刻，q\_server所作的判决仍然停留在pkt\_len\_TT（或pkt\_len\_RC[*i*]，*i*=0,1）值等于0的状态，而此时恰好上升沿之前的bool\_ready\_H（或bool\_ready\_L[*i*]，*i*=0,1）还没有变为无效，则造成错误地又生成了一个bool\_go\_H（或bool\_go\_L[*i*]，*i*=0,1）等于TRUE的多余的脉冲。

为了解决上述问题，可以使到达队列pkt\_len\_TT（或pkt\_len\_RC[*i*]，*i*=0,1）值的刷新在时钟的下降沿，避免了很多重要的状态数据均在时钟上升沿变化对系统的压力；然而，要彻底解决该问题，可以在队列服务器中加入新的状态，使得它不是立即从S\_STOP状态转移到S\_GO状态，避免信号之间由于时序因果关系的微小差别导致系统工作状况错误的情况。设修改之后的模块分别为q\_Q16\_negedge（到达队列）和q\_server\_3\_states（队列服务器）。

#### 用户显示的设计

实验装置采用三个LED灯TT（红色）、RC0（绿色）和RC1（黄色）表示正在接受服务的队列。当对应灯亮时，表示该队列正在接受服务。八个LED灯显示数据包剩余长度，当包剩余长度大于等于对应数值时，灯亮，否则灯灭。两个绿色LED灯表示时隙（或“孔”）的枚举类型，其中灯亮表示“1”，灯灭表示0。T\_PCF(编码10)为PCF帧的发送孔隙，T\_TT(编码11)为TT流量的发送孔隙，T\_MAR（编码01）为保护间隔，T\_DEF(编码00)是可以发送RC数据包的“默认”类型，实际是PCF、TT等时间触发时隙之间的“孔”。另外八个LED等表示孔隙剩余长度，当孔隙剩余长度大于等于对应数值时，灯亮，否则灯灭。具体显示设计如图所示。



1. 用户显示的设计

在输入方面，对于采用按键开关得到的key\_rst\_n信号，进行去抖，形成各个模块实例的rst\_n复位信号。

时钟方面，采用10MHz的时钟作为时钟输入。

### 基本要求

a) 理解带有保护间隔的固定优先级排队的需求，并明确相应的关键模块的功能。（除待补全的代码之外，提供其它模块的Verilog HDL源代码，并提供一些test bench测试代码）

b) 采用硬件描述语言实现round\_robin\_FP模块，注意其中状态机的定义。

c) 发现1.2.2所述的问题，并编写改进的队列服务器模块，并加以测试和调试。

d) 改写FPQ\_layout.v中的FPQ\_disp\_q\_head并在FPQ\_top\_layer.v中实例化，能够显示TT队列、RC0队列和RC1队列的头部数据帧的帧长情况。

### 扩展要求

e)将数据包剩余长度和孔隙剩余长度由二进制转化为十进制，并通过数码管显示（考虑到FPGA端口数量有限，可以通过4位数码管进行显示）。

f)在理解带有保护间隔的固定优先级排队的基础上，修改数据帧长以及孔隙长度，并解释实验现象。

g)采用更为合理与丰富的人机交互界面。

### 附加说明

(1) 为了使配置表简洁，参考设计并没有为每bit对应计数值，而是以传输16 Byte（128 bit）所需时间为“单位1”，最短帧对应数值为4，最长帧对应数值为95（最长帧未1518 Byte，约等于95所代表的1520 Byte），但计数的时候采用12位计数器，在bit\_clk的触发下对应每个比特计数；

(2) 100BASE-T以太网传输1bit时间为10ns，实验装置主频无法达到，因此设定在实验演示中传输1bit为5ms，这样即使对于最短帧，也需传输2.56s，便于演示和观测；

(3) 实用中，服务器q\_server\_3\_states的输入应该是流量的到达队列（如：同步或异步的FIFO），在本实验中，采用循环队列（内部保存有16个数据帧长度）代替FIFO；

(4) 代码文件是这样组织的，FPQ\_util.v是实现中用到的分频器等工具性模块，FPQ\_components.v是考虑保护间隔的固定优先级排队协议的关键组件，FPQ\_layout.v是具体到本实验需要用到的循环队列、用户界面显示等，而FPQ\_top.v是顶层模块；（各文件中的模块如下表所示）

1. 各文件中的模块

| 文件 | 模块 |
| --- | --- |
| FPQ\_util.v | MODULE : div\_half  MODULE : divider  MODULE : div\_16  MODULE : key\_debounce  MODULE : meter\_log2\_rank8\_max95  MOUDLE : mux4 |
| FPQ\_components.v | MODULE: timetable  MODULE: cur\_value\_translator  MODULE: q\_server // 待改进版本  MODULE: q\_server\_3\_states // 待补全  MODULE: round\_robin\_FP // 待补全 |
| FPQ\_layout.v | MODULE: q\_Q16\_negedge  MODULE: q\_Q16 // 待改进版本  MODULE: FPQ\_disp\_q\_head //待改进版本  MODULE: FPQ\_disp\_state // 作废版本  MODULE: FPQ\_disp\_state\_by\_meter  MODULE: FPQ\_mux\_1TT\_2RC  MODULE: FPQ\_disp\_value |
| FPQ\_top\_layer.v | FPQ\_top\_1TT\_2RC\_with\_ui |

# 实验三、带有保护间隔的固定优先级排队（实验报告部分）

## 实验报告

### 实验背景与需求分析

信息时代，数据网络通信是信息互联的基础设施，而显而易见的一点是，数据流之间同样是有区别的，这些区别存在于数据流传输的数据类型与数据流的优先级等属性，因此，根据网络协议，交换机或转发设备需要对不同的数据流提供不同资源和实时性保证的服务。

而在一些环境下，我们需要高速网络进行通信，在这种前提下现如今普遍应用的基于计算机的软件算法实现方案显然是不现实的，且对于“通信”，其成本也过于高昂，而基于本学期所学的数字电路相关知识，我们仅需将网络协议“扁平化”，即可以直接从硬件层面实现上述高速网络通信的应用。

数据传输的过程中有几个要点需要注意：

1.数据的传输不应当被打断，即数据包必须完整地被传输，高优先级的数据包也无法阻止正在接受服务的低优先级的数据包。

2.对数据进行类型与优先级的区分，保障多数据流信息的传输具备一定的条理性，使不同数据流穿插传输，流量之间实现轮询调度，尽量控制不同的数据流在单位时间的传输速率保持稳定。

3.为避免不同类型的数据流在传输过程中相互影响，设定保护间隔，防止低优先级数据包借由“正在传输”这一状态抢占高优先级数据包的传输时隙。

基于上述三个原则与需求，本实验模拟2种流量，即时间触发流量（TT）和事件触发流量（RC），TT流量享有固定的时隙和高优先级，事件触发流量之间进行轮询（round-robin）调度。

在TT流量的时隙之前，要留有保护间隔（margin），以防止低优先级数据包的不可抢占性阻塞本应在时隙内严格确定性传输的TT流量数据包。其中对保护间隔的设计需要我们进行考虑，既要防止保护间隔过短从而影响TT流量的正常传输，又要防止保护间隔过长从而造成数据传输的时间浪费。

因此，我们会给予保护间隔足够长的宽度，并在保护间隔中设定规则，使之可以在足够的剩余宽度种对一些短的数据包进行传输，从而规避保护间隔的存在对时间的大幅浪费。

基于现有代码（包括到达队列等模块的设计），小组成员将设计并实现三态队列服务器与多路复用器的设计，从而完成对数据流量传输这一过程的模拟，模拟实现TT流量与RC流量的混合多路复用，并通过授课组提供的test代码对我们设计的模块进行检验与针对数据传输的一些场景的实现。

### 系统设计

#### 总体设计思路

根据实验题目的相关要求，本实验的必要模块设计任务包括区别于现有两态服务器的三态队列服务器（q\_server\_3\_states）以及多路复用器（round\_robin\_FP）。

三态队列服务器的设计思路为依据两态服务器实现功能上的复现，同时在之前的两态：STOP和GO之外额外引入第三态：PAUSE，从而规避电路实际运行过程中细微的时间变化对系统整体的较大影响。

三种状态之间的转换关系可以用如下框图进行描述。

1. 三态转换关系

多路复用器的结构见图5。根据这一结构，我们引入5个状态，并在代码中对状态与状态之间的转换关系进行设计，从而完成对多路复用器的实现。

建议：系统软硬件设计思路，模块划分。

#### 接口设计

建议：接口设计。

#### 三态队列服务器模块（q\_server\_3\_states）

这一部分我们的整体代码设计如下：

module q\_server\_3\_states

#( parameter [1:0] PRIORITY = 2'b11 )

// 2'b00 -- PCF , 2'b01 -- TT, 2'b11 -- RC, 2'b10 -- BE

(

input [7:0] cur\_value ,

input [7:0] pkt\_len ,

input clk ,

input ena\_n ,

input rst\_n ,

output reg bool\_ready ,

output reg bool\_go

);

localparam [1:0] S\_STOP = 2'b00 ;

localparam [1:0] S\_GO = 2'b11 ;

localparam [1:0] S\_PAUSE = 2'b01 ;

reg [1:0] state;

reg [1:0] next\_state;

always @( cur\_value or pkt\_len )

if ( cur\_value >= pkt\_len ) bool\_ready = 1'b1 ;

else bool\_ready = 1'b0 ;

// State changes only at pos edge of clock

always @( posedge clk or negedge rst\_n )

if ( !rst\_n ) state <= S\_STOP ;

else state <= next\_state;

//State machine using case statements

always @( state or ena\_n or pkt\_len )

case ( state )

S\_STOP:

if(!ena\_n) next\_state = S\_PAUSE;

else next\_state = S\_STOP;

S\_PAUSE :

if ( !ena\_n ) next\_state = S\_GO ;

else next\_state = S\_STOP ;

S\_GO :

if ( ena\_n ) next\_state = S\_STOP ;

else if ( pkt\_len == 0 ) next\_state = S\_STOP ;

else next\_state = S\_GO ;

endcase

//Compute values

always @( state )

case ( state )

S\_STOP : bool\_go = 1'b0 ;

S\_PAUSE : bool\_go=1'b0;

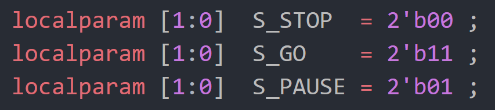
S\_GO : bool\_go = 1'b1 ;

endcase

endmodule

以下对其中的细节进行说明。

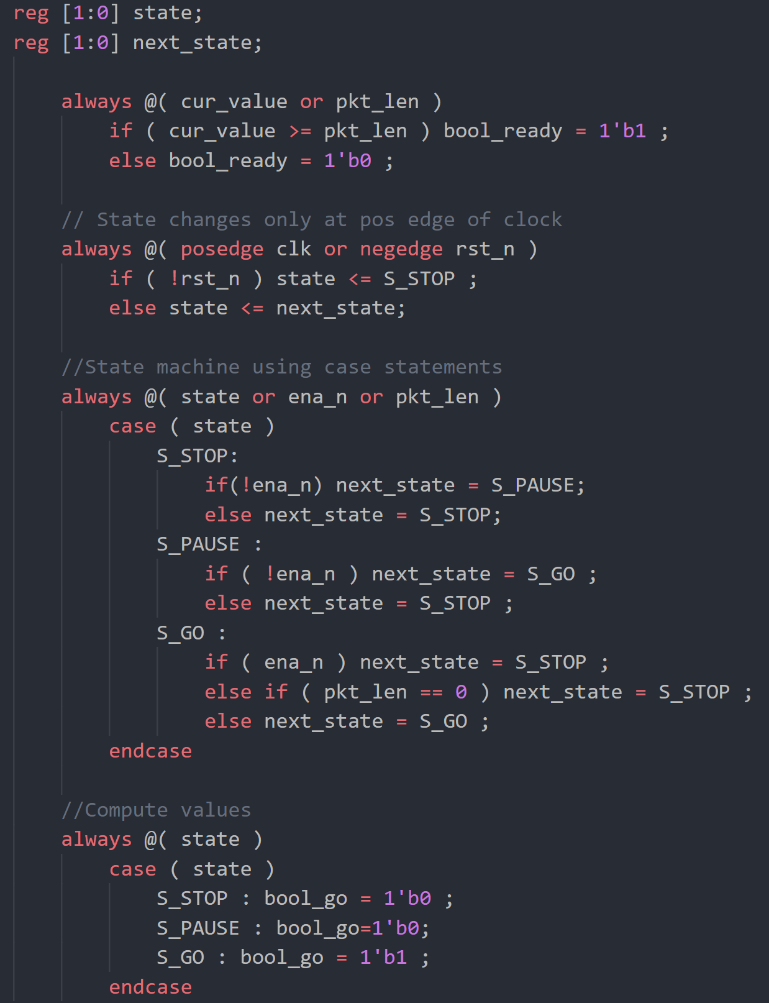
如图9所示，首先是以2位变量对三种状态进行了描述。



1. 针对三个状态的定义

而后我们针对状态转换关系进行设计，并引入了数据包的传输过程。

下图中的state变量表示当前的状态，state\_next变量用于暂存下一个状态。两个变量有且仅有三种可能的值即S\_STOP、S\_PAUSE、S\_GO。模块将通过现状态、使能信号与下一个数据包的状态得到下一个应有的状态，在此基础之上于下一个时钟周期完成状态的转换即将state\_next的值赋给state。



1. 针对状态转换关系进行的设计

需要说明的是，上述代码中bool\_go与bool\_ready是我们设计的1位标记变量，用于标记队列传输情况：当减法计数器cur\_value变量不为0或pkt\_len不为 0时即当减法计数器表明队列仍有剩余空间可用于数据传输或下一个数据包存在时，数据准备传输，bool\_ready置1，否则置0；当队列传输状态为STOP或者PAUSE，即传输队列不处于“正在传输”的状态，此时bool\_go变量为0，反之为1，用于声明队列仍处在继续传输的状态。

建议：叙述设计实现中的关键模块，根据设计复杂度，简单的设计2~3个为宜，复杂的设计酌情设置。

#### 多路复用器模块（round\_robin\_FP）

这一部分我们的整体代码设计如下：

module round\_robin\_FP

# ( parameter N = 2 )

(

input clk ,

input rst\_n ,

//

input bool\_ready\_H ,

input bool\_go\_H ,

input [ N-1 : 0 ] bool\_ready\_L ,

input [ N-1 : 0 ] bool\_go\_L ,

//

output reg ena\_n\_H ,

output reg [ N-1 : 0 ] ena\_n\_L ,

//

output reg [1:0] active , // 00 - none, 01 - H, 10 - L

output reg [3:0] channel

);

localparam [3:0] S\_ACTIVE\_H = 4'b0011 ;

localparam [3:0] S\_ACTIVE\_L = 4'b1100 ;

localparam [3:0] S\_NONE = 4'b0000 ;

localparam [3:0] S\_WAIT\_H = 4'b0001 ;

localparam [3:0] S\_WAIT\_L = 4'b1000 ;

localparam [1:0] A\_H = 2'b01 ;

localparam [1:0] A\_L = 2'b10 ;

localparam [1:0] A\_NONE = 2'b00 ;

reg [7:0] cnt\_timeout ; // count down

`ifndef BOOL\_JUST\_FOR\_SIMU

localparam [7:0] VAL\_TIMEOUT = 8'h80 ;

`else

localparam [7:0] VAL\_TIMEOUT = 8'd8 ;

`endif

reg [3:0] state ;

reg [3:0] index\_L ;

integer i ;

always @( posedge clk or negedge rst\_n )

if ( ! rst\_n ) begin

state <= S\_NONE ;

index\_L <= 0 ;

ena\_n\_H <= 1'b1 ;

for ( i=0; i<N ; i=i+1 ) ena\_n\_L[i] <= 1'b1 ;//这里为什么不整体赋值？

end

else begin

case ( state )

S\_NONE:

if( bool\_ready\_H ) begin

ena\_n\_H <= 1'b0 ;

state <= S\_WAIT\_H ;

cnt\_timeout <= VAL\_TIMEOUT ;

end

else if( bool\_ready\_L != 0 )

if( bool\_ready\_L [ index\_L ] ) begin

ena\_n\_L [ index\_L ] <= 1'b0 ;

state <= S\_WAIT\_L ;

cnt\_timeout <= VAL\_TIMEOUT ;

end

else begin

if( index\_L == N-1 ) index\_L <= 4'd0 ;

else index\_L <= index\_L + 4'd1 ;

end

else state <= S\_NONE;

S\_WAIT\_H:if(bool\_go\_H) state <= S\_ACTIVE\_H;

else if(cnt\_timeout == 0)

begin

ena\_n\_H <= 1'b1;

state <= S\_NONE;

end

else

begin

cnt\_timeout <= cnt\_timeout -1;

state <= S\_WAIT\_H;

end

S\_ACTIVE\_H:if(!bool\_go\_H)

begin

ena\_n\_H <= 1'b1;

state<=S\_NONE;

end

S\_WAIT\_L:if(bool\_go\_L[index\_L]) state <= S\_ACTIVE\_H;

else if(cnt\_timeout == 0)

begin

ena\_n\_L[index\_L] <= 1'b1;

if( index\_L == N-1 ) index\_L <= 4'd0 ;

else index\_L <= index\_L + 4'd1 ;

state<=S\_NONE;

end

else cnt\_timeout <= cnt\_timeout -1;

S\_ACTIVE\_L:if(!bool\_go\_L[index\_L])

begin

ena\_n\_L[index\_L] <= 1'b1;

if( index\_L == N-1 ) index\_L <= 4'd0 ;

else index\_L <= index\_L + 4'd1 ;

state<=S\_NONE;

end

endcase

end

always @( state )

case ( state )

S\_ACTIVE\_H: active <= A\_H ;

S\_WAIT\_H: active <= A\_H ;

S\_ACTIVE\_L: active <= A\_L ;

S\_WAIT\_L: active <= A\_L ;

S\_NONE: active <= A\_NONE ;

default: active <= A\_NONE ;

endcase

always @( active or index\_L )

if ( active == A\_H ) channel = 0 ;

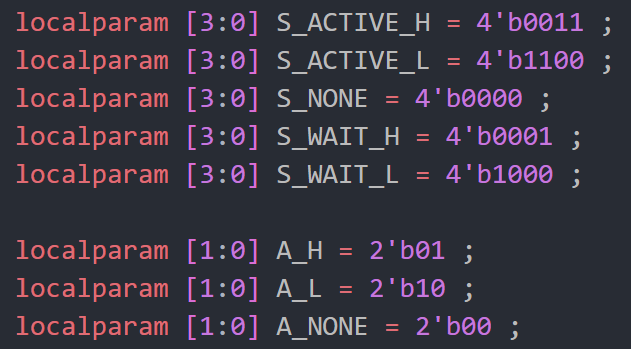
else if ( active == A\_L ) channel = index\_L ;

else channel = 0 ;

endmodule

以下对代码的整体设计思路进行说明。

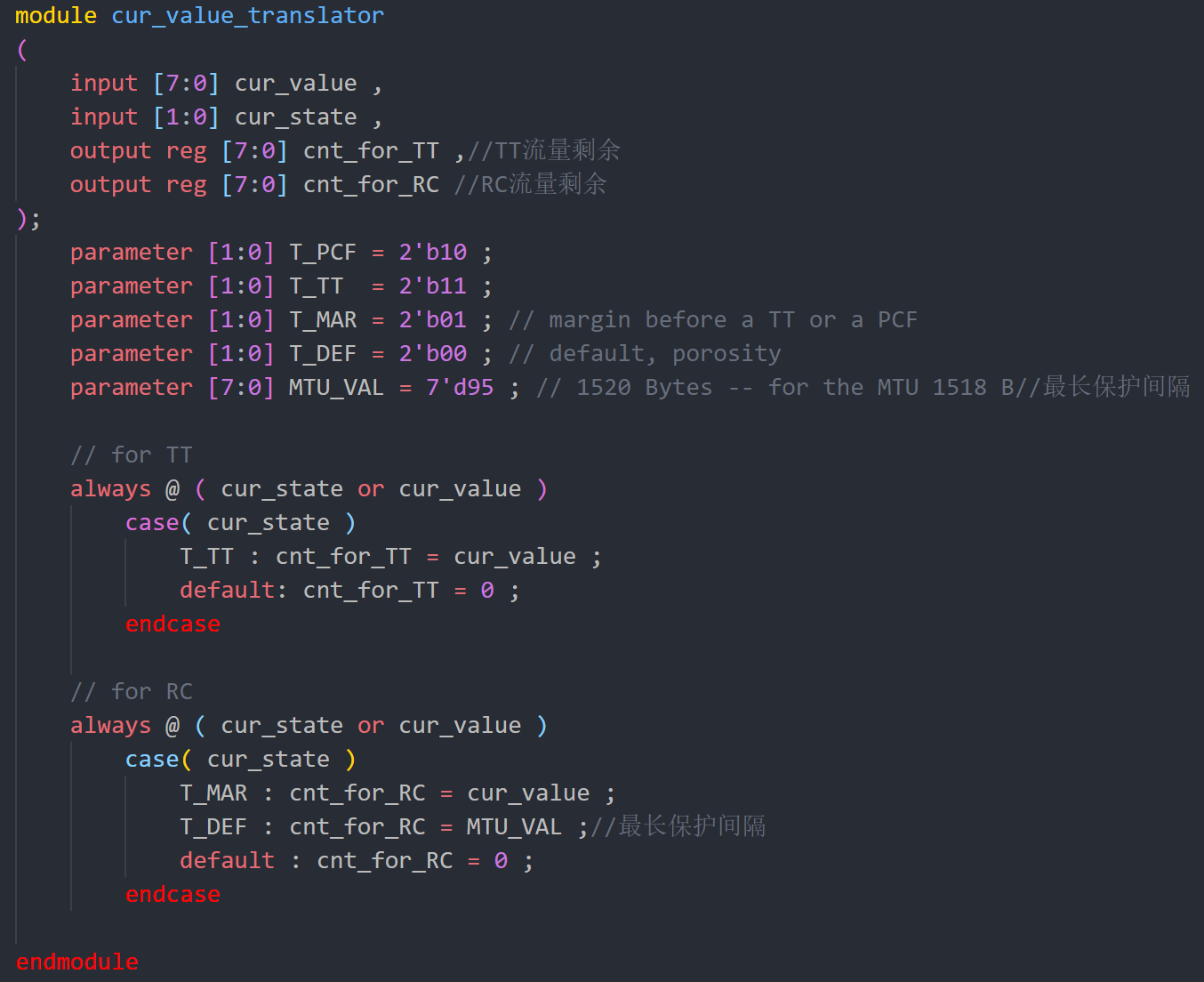
该代码中设定的“状态”一共有5种，设定的“响应”一共有三种，如图所示：



1. 多路复用器模块状态与响应的设置

程序运行的初始状态时NONE，在此情况下，使能信号无效，ready信号置为0。

在此对与该模块相关的cur\_value\_translator进行说明：



1. cur\_value\_translator模块代码

cur\_value\_translator模块的作用是将cur\_value基于状态转换成对应流量的 cnt变量，而输出的cnt变量则是在round\_robin\_FP模块中与数据包头部长度比较，进而确定ready信号置1还是置0。

ready信号传入轮询模块，假如ready信号置１，NONE状态跳转到WAIT态，同时使能信号有效，计时器赋值。

使能信号传回服务器模块，假如使能信号有效，传回置１的go信号。

WAIT模块计时器计数，假如超时，返回NONE态。未超时，go信号置１的情况下，跳转到ACTIVE态。

其中每次先询问H，（优先级较高），不传H的情况下，对RC包进行轮询，根据ready［Ｌ］的情况确定传输哪一个RC包。

ACTIVE态：传输状态，go信号重新置０后，使能信号无效，返回NONE态。

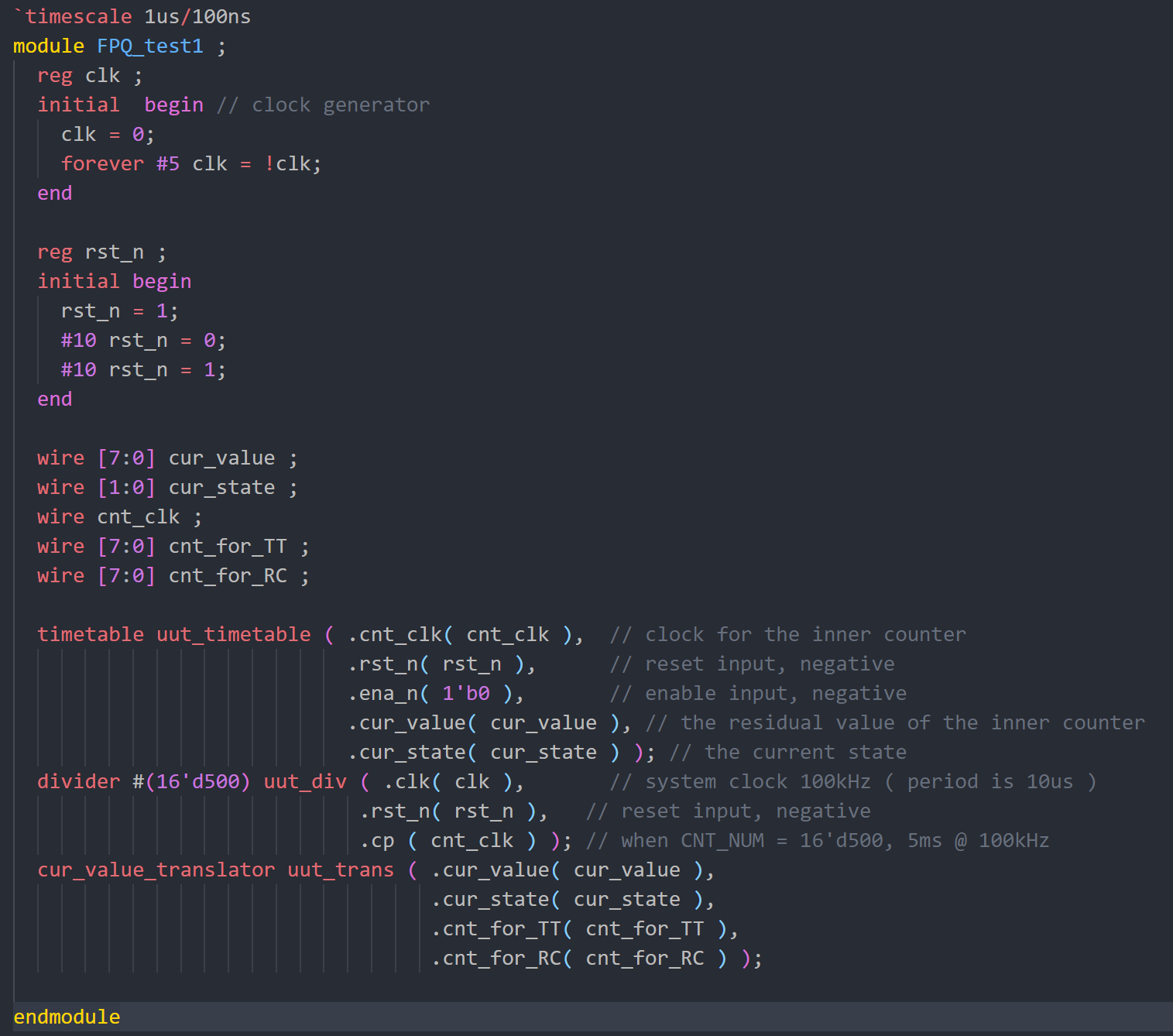
### 功能仿真测试

#### 测试程序设计

以下将针对每一个测试程序进行说明，由于部分的测试程序篇幅过长，本文将对核心代码进行重点讲解，全部的源代码将放在附录部分。

##### 测试程序test1

测试程序代码如下：

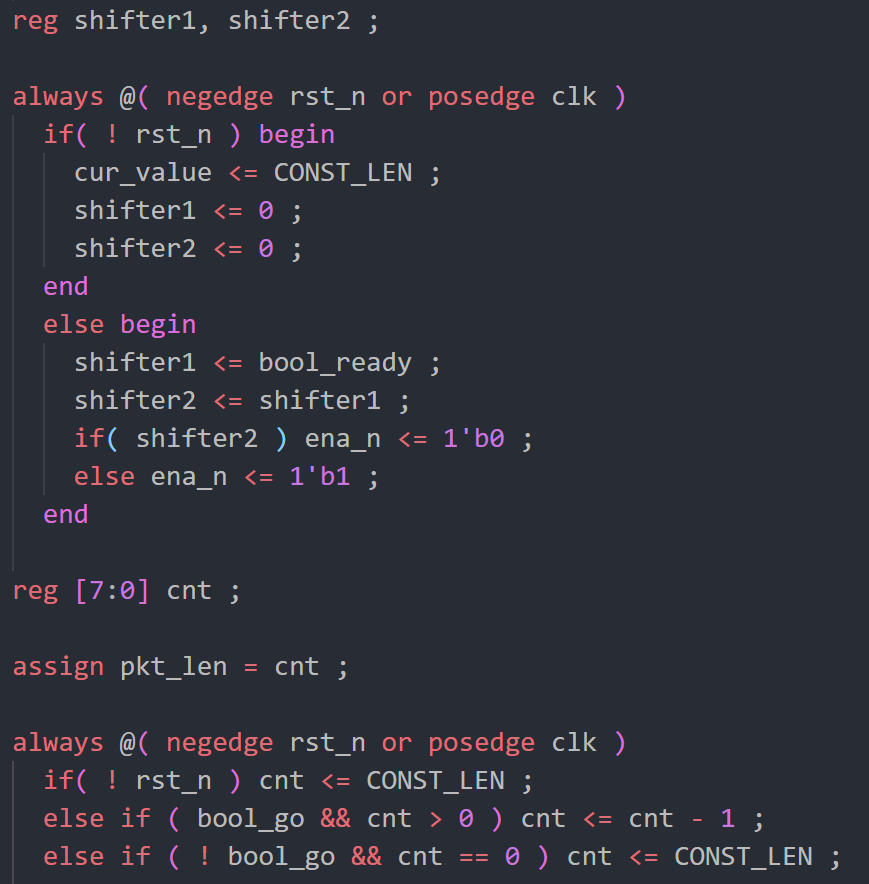


1. 测试程序test1全部代码

该段测试代码的设计是为了测试timetable中设置的流量是否能够正常传送。基于此，该测试代码未调用Q16模块，同时没有考虑队列服务器与多路复用器。测试代码中仅仅是针对cur\_value进行不断减小操作，并不停地循环与读取timetable中数据类型及数据包长度，并在仿真界面中观察cur\_value、cur\_for\_TT\RC的值，对这几个变量是否正常减少进行确认。

##### 测试程序test2

测试程序代码如下：

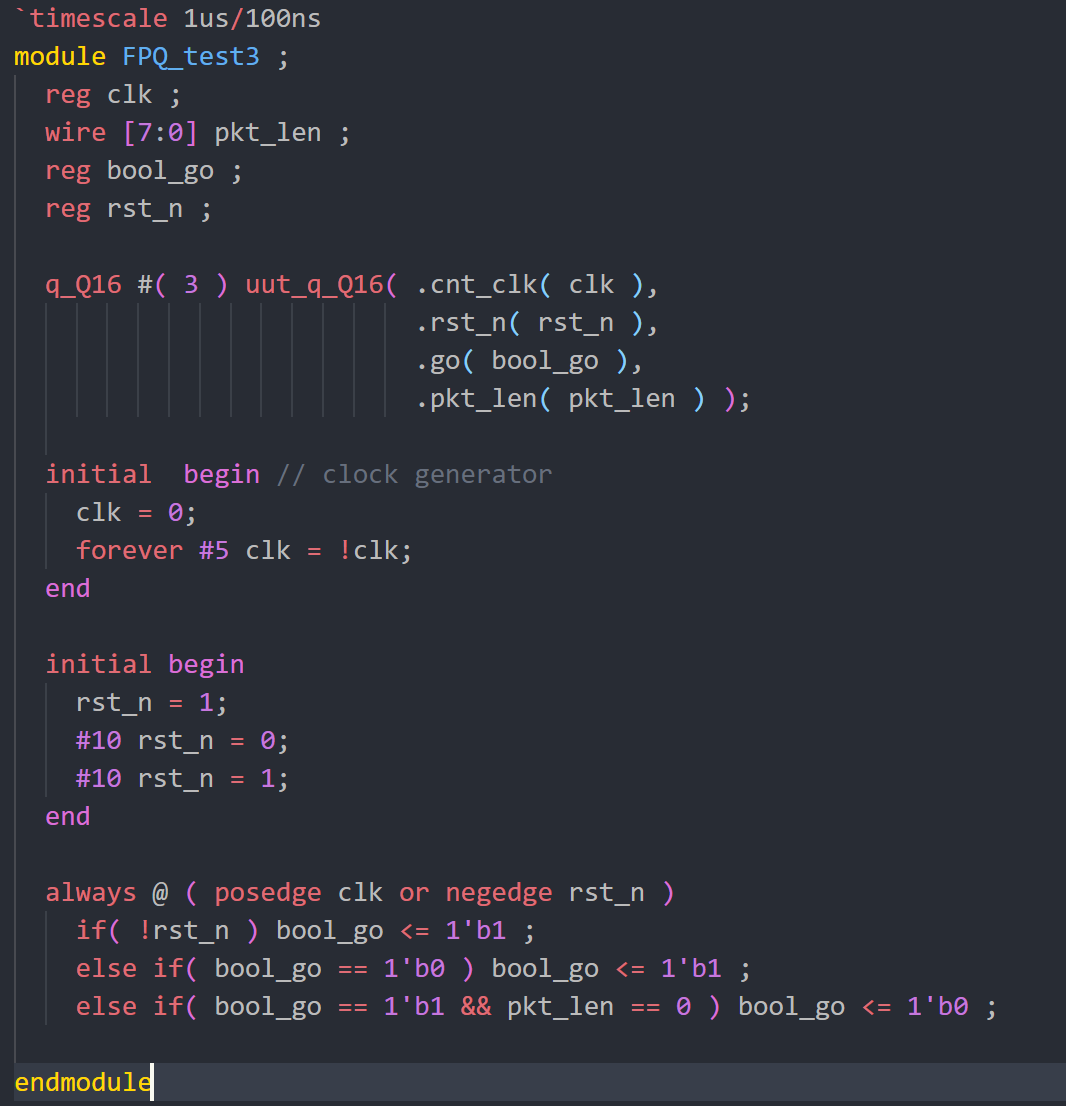


1. 测试程序test2部分代码

该段测试代码是为测试现有的q\_server模块的功能，从而让我们在q\_server的仿真结果中预先对我们要改进出的三态队列服务器具备最基本的思路。具体的代码编写是设定在一段时间内，固定cur\_value不变，并使得pkt不断减小，模拟数据包的传输与新来的数据包的内容不断减少的过程，最终用于测试二态的q\_server对bool\_go与bool\_ready的赋值，并体现在仿真界面中。且由于使用了两个shifter作为延时，所以没有出现脉冲。

##### 测试程序test3

测试程序代码如下：

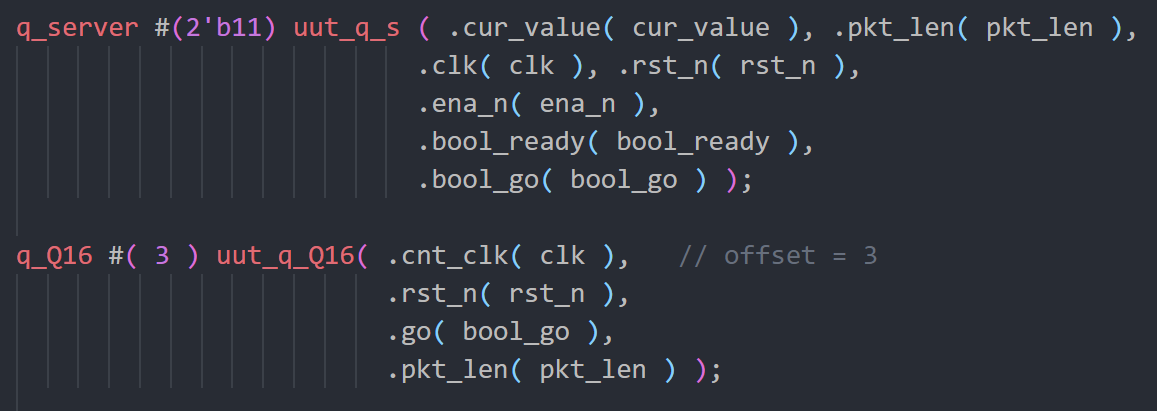


1. 测试程序test3全部代码

该段测试代码主要是用于测试Q16模块，从而让我们了解Q16模块的运行模式，进一步帮助我们开拓修改Q16\_negedge模块的思路。代码的实现上，主要是用bool\_go变量来控制pkt\_len。基于实验指导书对这一模块存在问题的表述（见1.2.2），我们也将通过这一测试程序对实验指导书指出的问题进行发现。并基于该程序进行Q16\_negedge模块进行设计。

##### 测试程序test4

测试程序代码如下：



1. test4程序模块调用情况

该代码整体结合了test2与test3的设计思路，在此之上进行测试的升级：测试Q16与q\_server的联合使用。在该代码中，bool\_go变量是由q\_server模块进行提供的。bool\_go最后得到的bool\_go变量也会继续用于控制Q16中pkt\_len的变化。

##### 测试程序test5

测试程序代码如下：

1. 测试程序test5部分代码（补充：test5代码太多了我不知道哪里的匹配下面的话）

该测试程序的设计显著要比之前的测试程序更为复杂，测试内容是多态转换测试，将四态的队列服务器与多路复用器联合测试，测试了当所有流量的keep、op、rdy\_change变量都为1时的情况，其中转化关系为：

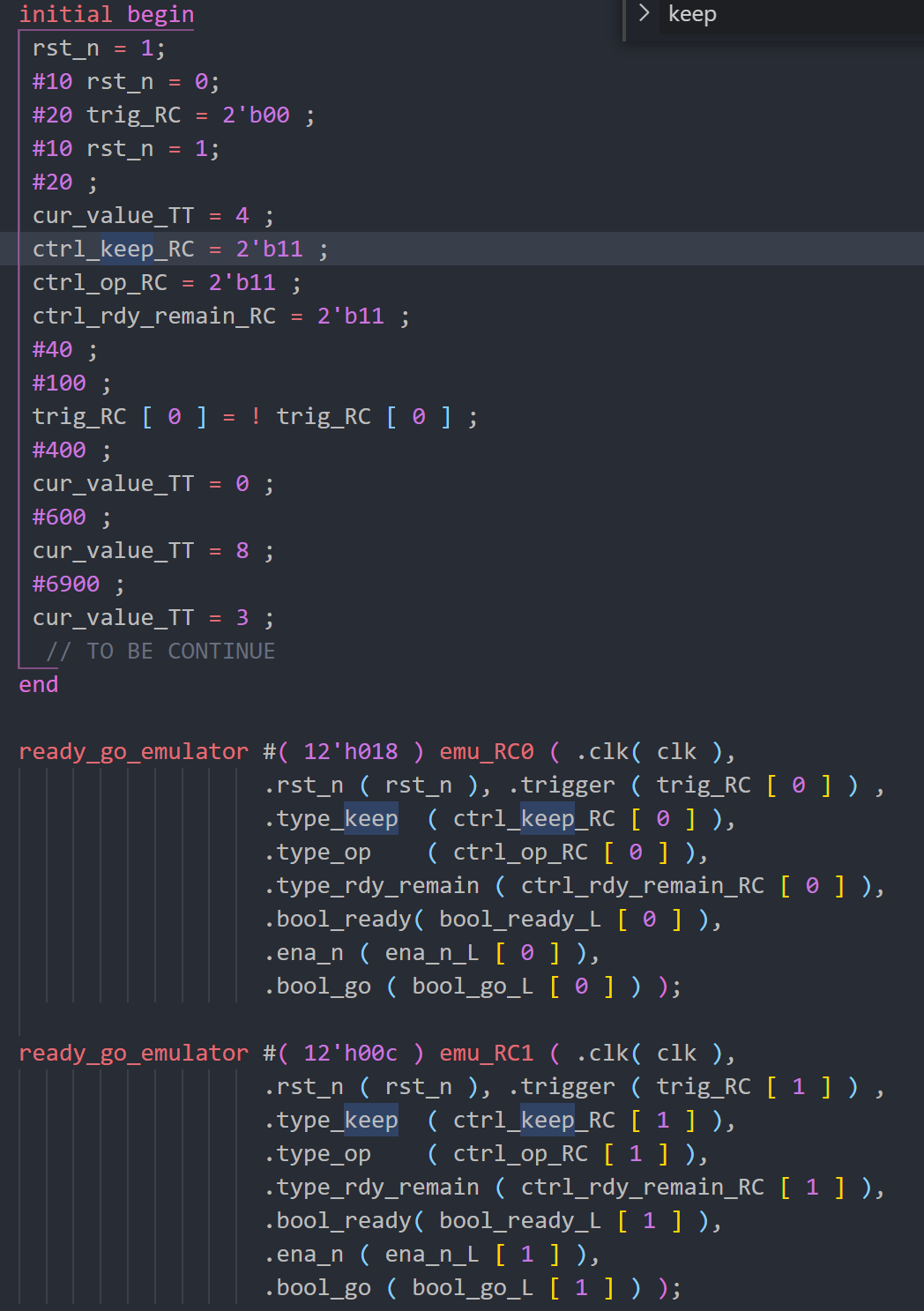
当keep为TRUE时，若trigger有变化，则状态会从WAIT->READY，之后若不变，则会再READY->GO->OVER->GO中不断循环；当keep变为false时，则over不会转化为READY进行下一轮循环，而是变为wait，等待下一次trigger变化。

在此这里以TT为例子。

TT流量：WAIT在接收到pulse（trigger变化）后变为READY态，在ena\_n==0时转化为GO态，等待16个时钟周期转化为OVER态，再等待2个周期后变为为READY态，再同之前一样循环，从ready到go，再到over，此时由于再initial中将keep置为false，所以转化为WAIT等待下一次脉冲。随后trigger再次取反后，由进行一次WAIT->READY->GO->OVER->WAIT的循环。

##### 测试程序test6

测试程序代码如下：

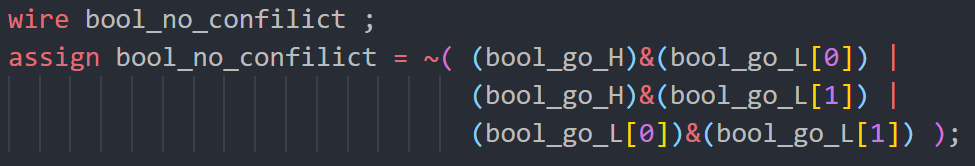


1. 测试程序test6部分代码

该代码对不同的流量采取了不同的调用方案：TT流量用三态的队列服务器、到达队列和多路复用器；RC1与RC0用四态的队列服务器和多路复用器。基于此，本测试代码的实质是对于对TEST4与TEST5的联合。其中RC0流量在传输过程中由于KEEP一直为1，故RC0在trigger变化后不断循环在READY->OVER->GO->READY四个状态之间，从GO可以看出。TT流量与TEST4是一致的。

##### 测试程序test6a

测试程序代码如下：



1. 测试程序test6a部分代码

相比于test6,增加了变量bool\_no\_confilict，用于判断是否存在两种及以上流量同时传输的情况。当两种及以上流量同时传输的情况存在时，bool\_no\_confilict为0，反之则为1。

##### 测试程序test6b

测试程序代码如下：



1. 测试程序test6b部分代码

相比于TEST6，该测试代码对四种流量以不同的方式进行了模拟传输。

TT流量的传输测试调用了三态的队列服务器、到达队列和多路复用器。

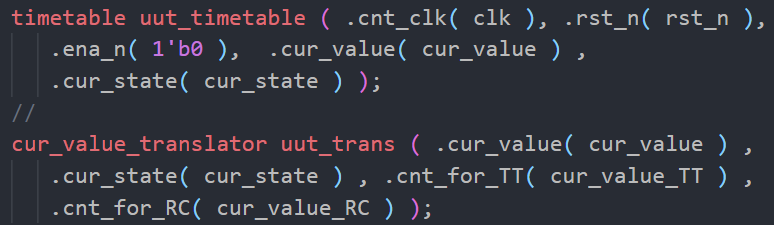
RC0、1流量的传输测试调用了四态的服务器、到达队列和多路复用器。

RC2流量是在该测试文件中单独生成的全新RC序列，用以测试不轮询状态下的RC流量传输，具体的RC2序列见下图。该流量的传输测试调用了三态的服务器、到达队列和多路复用器。

最后要将三种RC流量与TT流量都用于多路复用器，所以多路复用器的N设置为3。

##### 测试程序test7

测试程序代码如下：

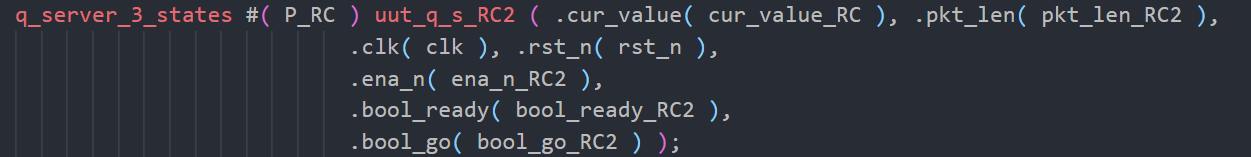


1. 测试程序test7部分代码

相比于test6b，test7添加了timetable模块，用来循环读取cur\_value，并用cur\_value\_translator模块将cur\_value分配给cur\_value\_TT和cur\_value\_RC，最终用于TT流量与RC流量中，对于测试程序自主设定的RC2流量的剩余计数器剩余值则仍是在initial中进行赋值。

##### 测试程序test7a

测试程序代码如下：

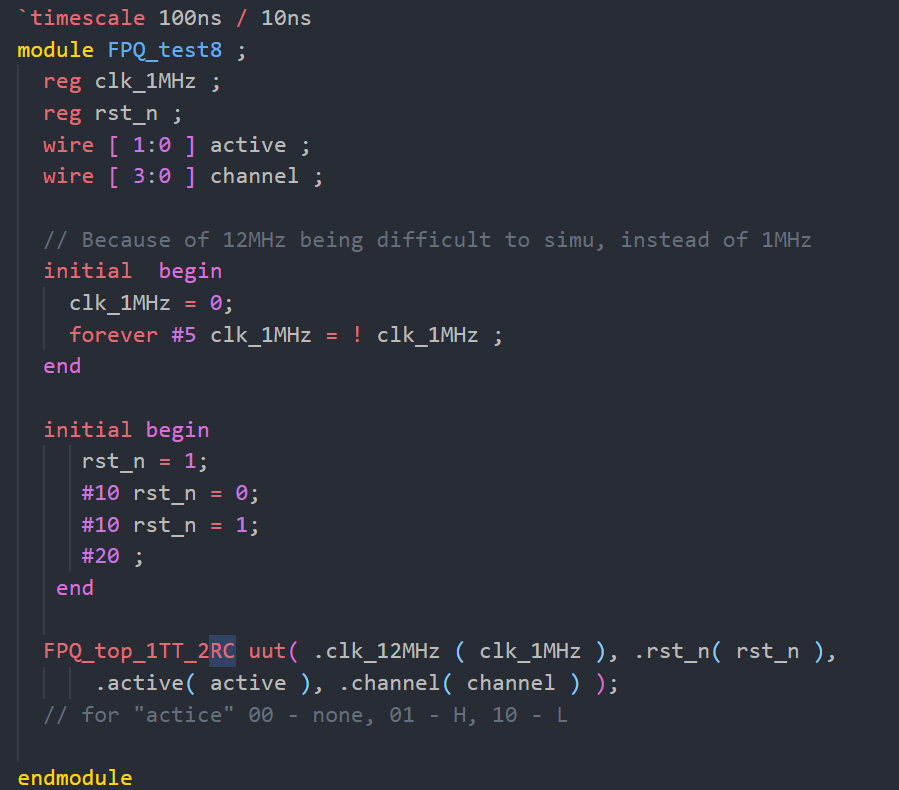


1. 测试程序test7a部分代码

相比于test7，该测试代码中的 RC2流量剩余值计数器调用了timetable模块输出的cur\_value\_RC，而不是用initial赋值。

##### 测试程序test8

测试程序代码如下：



1. 测试程序test8全部代码

该测试代码综合使用了Q16\_negdge与FPQ\_mux\_1TT\_2RC模块，将TT与RC0、RC1三种流量进行传输。在仿真方面完成了对整个带有保护间隔的固定优先级排队系统的测试。

##### 测试程序test9

测试程序代码如下：

相比于TEST8多加了显示模块。

#### 功能仿真过程

#### 实验关键结果及其解释

### 设计实现

#### 综合和下载过程

#### 实验关键结果及其解释

建议：如果调试过程较复杂，可分小节撰写。

### 小结

## 参考文献

1. （如果有）

## 附录

### 测试程序源代码

#### 测试程序test1

`timescale 1us/100ns

module FPQ\_test1 ;

reg clk ;

initial begin // clock generator

clk = 0;

forever #5 clk = !clk;

end

reg rst\_n ;

initial begin

rst\_n = 1;

#10 rst\_n = 0;

#10 rst\_n = 1;

end

wire [7:0] cur\_value ;

wire [1:0] cur\_state ;

wire cnt\_clk ;

wire [7:0] cnt\_for\_TT ;

wire [7:0] cnt\_for\_RC ;

timetable uut\_timetable ( .cnt\_clk( cnt\_clk ), // clock for the inner counter

.rst\_n( rst\_n ), // reset input, negative

.ena\_n( 1'b0 ), // enable input, negative

.cur\_value( cur\_value ), // the residual value of the inner counter

.cur\_state( cur\_state ) ); // the current state

divider #(16'd500) uut\_div ( .clk( clk ), // system clock 100kHz ( period is 10us )

.rst\_n( rst\_n ), // reset input, negative

.cp ( cnt\_clk ) ); // when CNT\_NUM = 16'd500, 5ms @ 100kHz

cur\_value\_translator uut\_trans ( .cur\_value( cur\_value ),

.cur\_state( cur\_state ),

.cnt\_for\_TT( cnt\_for\_TT ),

.cnt\_for\_RC( cnt\_for\_RC ) );

Endmodule

#### 测试程序test2

`timescale 1us/100ns

module FPQ\_test2 ;

localparam CONST\_LEN = 8'd4 ;

reg clk ;

reg [7:0] cur\_value ;

wire [7:0] pkt\_len ;

reg ena\_n ;

wire bool\_go ;

initial begin // clock generator

clk = 0;

forever #5 clk = !clk;

end

reg rst\_n ;

initial begin

rst\_n = 1;

#10 rst\_n = 0;

#10 rst\_n = 1;

#100

cur\_value = 4 ;

#200

cur\_value = 16 ;

#200

cur\_value = 4 ;

#200

cur\_value = 2 ;

#200

cur\_value = 0 ;

end

q\_server #(2'b11) uut\_q\_s ( .cur\_value( cur\_value ), .pkt\_len( pkt\_len ),

.clk( clk ), .rst\_n( rst\_n ),

.ena\_n( ena\_n ),

.bool\_ready( bool\_ready ),

.bool\_go( bool\_go ) );

reg shifter1, shifter2 ;

always @( negedge rst\_n or posedge clk )

if( ! rst\_n ) begin

cur\_value <= CONST\_LEN ;

shifter1 <= 0 ;

shifter2 <= 0 ;

end

else begin

shifter1 <= bool\_ready ;

shifter2 <= shifter1 ;

if( shifter2 ) ena\_n <= 1'b0 ;

else ena\_n <= 1'b1 ;

end

reg [7:0] cnt ;

assign pkt\_len = cnt ;

always @( negedge rst\_n or posedge clk )

if( ! rst\_n ) cnt <= CONST\_LEN ;

else if ( bool\_go && cnt > 0 ) cnt <= cnt - 1 ;

else if ( ! bool\_go && cnt == 0 ) cnt <= CONST\_LEN ;

endmodule

#### 测试程序test3

`timescale 1us/100ns

module FPQ\_test3 ;

reg clk ;

wire [7:0] pkt\_len ;

reg bool\_go ;

reg rst\_n ;

q\_Q16 #( 3 ) uut\_q\_Q16( .cnt\_clk( clk ),

.rst\_n( rst\_n ),

.go( bool\_go ),

.pkt\_len( pkt\_len ) );

initial begin // clock generator

clk = 0;

forever #5 clk = !clk;

end

initial begin

rst\_n = 1;

#10 rst\_n = 0;

#10 rst\_n = 1;

end

always @ ( posedge clk or negedge rst\_n )

if( !rst\_n ) bool\_go <= 1'b1 ;

else if( bool\_go == 1'b0 ) bool\_go <= 1'b1 ;

else if( bool\_go == 1'b1 && pkt\_len == 0 ) bool\_go <= 1'b0 ;

endmodule

#### 测试程序test4

`timescale 1us/100ns

module FPQ\_test4 ;

localparam CONST\_LEN = 8'd4 ;

reg clk ;

reg [7:0] cur\_value ;

wire [7:0] pkt\_len ;

reg ena\_n ;

wire bool\_go ;

initial begin // clock generator

clk = 0;

forever #5 clk = !clk;

end

reg rst\_n ;

initial begin

rst\_n = 1;

#10 rst\_n = 0;

#10 rst\_n = 1;

#100

cur\_value = 4 ;

#2000

cur\_value = 16 ;

#2000

cur\_value = 4 ;

#2000

cur\_value = 2 ;

#2000

cur\_value = 0 ;

end

q\_server #(2'b11) uut\_q\_s ( .cur\_value( cur\_value ), .pkt\_len( pkt\_len ),

.clk( clk ), .rst\_n( rst\_n ),

.ena\_n( ena\_n ),

.bool\_ready( bool\_ready ),

.bool\_go( bool\_go ) );

q\_Q16 #( 3 ) uut\_q\_Q16( .cnt\_clk( clk ), // offset = 3

.rst\_n( rst\_n ),

.go( bool\_go ),

.pkt\_len( pkt\_len ) );

reg shifter1, shifter2 ;

always @( negedge rst\_n or posedge clk )

if( ! rst\_n ) begin

cur\_value <= CONST\_LEN ;

shifter1 <= 0 ;

shifter2 <= 0 ;

end

else begin

shifter1 <= bool\_ready ;

shifter2 <= shifter1 ;

if( shifter2 ) ena\_n <= 1'b0 ;

else ena\_n <= 1'b1 ;

end

endmodule

#### 测试程序test5

`timescale 1us/100ns

module FPQ\_test5 ;

reg clk ;

reg rst\_n ;

//

wire bool\_ready\_H ;

wire bool\_go\_H ;

wire [ 1 : 0 ] bool\_ready\_L ;

wire [ 1 : 0 ] bool\_go\_L ;

//

wire ena\_n\_H ;

wire [ 1 : 0 ] ena\_n\_L ;

//

reg trig\_TT ;

reg [ 1 : 0 ] trig\_RC ;

reg ctrl\_keep\_TT ;

reg [ 1 : 0 ] ctrl\_keep\_RC ;

reg [ 1 : 0 ] ctrl\_op\_RC ;

reg [ 1 : 0 ] ctrl\_rdy\_remain\_RC ;

//

wire [1:0] active ; // 00 - none, 01 - H, 10 - L

wire [3:0] channel ;

//

round\_robin\_FP #(2) uut\_rr\_FP ( .clk( clk ), .rst\_n( rst\_n ),

.bool\_ready\_H ( bool\_ready\_H ),

.bool\_go\_H ( bool\_go\_H ),

.bool\_ready\_L ( bool\_ready\_L ),

.bool\_go\_L ( bool\_go\_L ),

.ena\_n\_H ( ena\_n\_H ),

.ena\_n\_L ( ena\_n\_L ),

.active ( active ),

.channel ( channel ) );

initial begin // clock generator

clk = 0;

forever #5 clk = !clk;

end

initial begin

rst\_n = 1;

#10 rst\_n = 0;

#20 trig\_TT = 0 ; trig\_RC = 2'b00 ;

#10 rst\_n = 1;

#20 ;

ctrl\_keep\_TT = 1'b1 ;

ctrl\_keep\_RC = 2'b11 ;

ctrl\_op\_RC = 2'b11 ;

ctrl\_rdy\_remain\_RC = 2'b11 ;

#40 ;

// test case by setting bool\_ready\_H/\_L

trig\_TT = ! trig\_TT ;

#300 ;

ctrl\_keep\_TT = 1'b0 ;

#10 ;

trig\_RC[ 0 ] = ! trig\_RC [ 0 ];

#90 ;

#900 ;

trig\_RC[ 1 ] = ! trig\_RC [ 1 ] ;

#400 ;

trig\_TT = ! trig\_TT ;

#200 ;

ctrl\_keep\_RC = 2'b10 ;

// TO BE CONTINUE

end

/\* INTERFACE OF MODULE ready\_go\_emulator

( input clk , rst\_n ,

input trigger ,

input type\_keep ,

input type\_op ,

input type\_rdy\_remain ,

output reg bool\_ready ,

input ena\_n ,

output reg bool\_go );

\*/

ready\_go\_emulator #( 12'h010 ) emu\_TT ( .clk( clk ),

.rst\_n ( rst\_n ), .trigger ( trig\_TT ) ,

.type\_keep( ctrl\_keep\_TT ),

.type\_op ( 1'b1 ),

.type\_rdy\_remain ( 1'b1 ),

.bool\_ready( bool\_ready\_H ),

.ena\_n ( ena\_n\_H ),

.bool\_go ( bool\_go\_H ) );

ready\_go\_emulator #( 12'h018 ) emu\_RC0 ( .clk( clk ),

.rst\_n ( rst\_n ), .trigger ( trig\_RC [ 0 ] ) ,

.type\_keep ( ctrl\_keep\_RC [ 0 ] ),

.type\_op ( ctrl\_op\_RC [ 0 ] ),

.type\_rdy\_remain ( ctrl\_rdy\_remain\_RC [ 0 ] ),

.bool\_ready( bool\_ready\_L [ 0 ] ),

.ena\_n ( ena\_n\_L [ 0 ] ),

.bool\_go ( bool\_go\_L [ 0 ] ) );

ready\_go\_emulator #( 12'h00c ) emu\_RC1 ( .clk( clk ),

.rst\_n ( rst\_n ), .trigger ( trig\_RC [ 1 ] ) ,

.type\_keep ( ctrl\_keep\_RC [ 1 ] ),

.type\_op ( ctrl\_op\_RC [ 1 ] ),

.type\_rdy\_remain ( ctrl\_rdy\_remain\_RC [ 1 ] ),

.bool\_ready( bool\_ready\_L [ 1 ] ),

.ena\_n ( ena\_n\_L [ 1 ] ),

.bool\_go ( bool\_go\_L [ 1 ] ) );

endmodule

module shifter2\_n

(

input clk, rst\_n ,

input in ,

output reg out\_n

);

reg d\_ff1, d\_ff2 ;

always @ ( negedge rst\_n or posedge clk )

if( ! rst\_n ) begin

d\_ff1 <= 0 ;

d\_ff2 <= 0 ;

end

else begin

d\_ff1 <= in ;

d\_ff2 <= d\_ff1 ;

if( d\_ff2 ) out\_n <= 1'b0 ;

else out\_n <= 1'b1 ;

end

endmodule

module ready\_go\_emulator

#( parameter [11:0] VAL\_DELAY = 12'h010 )

(

input clk , rst\_n ,

input trigger ,

input type\_keep ,

input type\_op ,

input type\_rdy\_remain ,

output reg bool\_ready ,

input ena\_n ,

output reg bool\_go

);

localparam TYPE\_KEEP\_FALSE = 0;

localparam TYPE\_KEEP\_TRUE = 1 ;

localparam TYPE\_OP\_NORMAL = 1 ;

localparam TYPE\_OP\_TIMEOUT = 0 ;

localparam TYPE\_READY\_REMAIN = 1 ;

localparam TYPE\_READY\_CHANGE\_EARLY = 0 ;

localparam [1:0] S\_WAIT = 2'b00 ;

localparam [1:0] S\_READY = 2'b01 ;

localparam [1:0] S\_GO = 2'b11 ;

localparam [1:0] S\_OVER = 2'b10;

localparam [11:0] VAL\_GO\_DOWN = 12'h002 ;

localparam [11:0] VAL\_READY\_CHANGE = 12'h008 ;

reg [1:0] state ;

reg [11:0] cnt ;

reg reg1\_trigger ;

reg reg2\_trigger ;

wire pulse ;

always @ ( negedge clk ) begin

reg1\_trigger <= trigger ;

reg2\_trigger <= reg1\_trigger ;

end

assign pulse = reg2\_trigger ^ trigger;

always @( posedge clk or negedge rst\_n )

if( !rst\_n ) begin

state <= S\_WAIT ;

bool\_ready <= 1'b0 ;

bool\_go <= 1'b0 ;

end

else

case( state )

S\_WAIT: begin

if( pulse ) begin

bool\_ready <= 1'b1 ;

state <= S\_READY ;

end

end

S\_READY: begin

if( ! ena\_n )

if ( type\_op == TYPE\_OP\_NORMAL ) begin

cnt <= VAL\_DELAY ;

bool\_go <= 1'b1 ;

state <= S\_GO ;

end

else // type\_op == TYPE\_OP\_TIMEOUT

state <= S\_WAIT ;

end

S\_GO: begin

if ( cnt == 0 ) begin

cnt <= VAL\_GO\_DOWN ;

bool\_go <= 1'b0 ;

state <= S\_OVER ;

end

else if ( type\_rdy\_remain == TYPE\_READY\_CHANGE\_EARLY &&

cnt == VAL\_READY\_CHANGE ) begin

bool\_ready <= 1'b0 ;

cnt <= cnt -1 ;

end

else cnt <= cnt - 1 ;

end

S\_OVER: begin

if( cnt == 0 ) begin

if( type\_keep == TYPE\_KEEP\_TRUE ) begin

bool\_ready <= 1'b1 ;

state <= S\_READY ;

end

else begin // type\_keep == TYPE\_KEEP\_FALSE

bool\_ready <= 1'b0 ;

state <= S\_WAIT ;

end

end

else cnt <= cnt - 1 ;

end

endcase

endmodule

#### 测试程序test6

`timescale 1us/100ns

module FPQ\_test6 ;

reg clk ;

reg rst\_n ;

//

wire bool\_ready\_H ;

wire bool\_go\_H ;

wire [ 1 : 0 ] bool\_ready\_L ;

wire [ 1 : 0 ] bool\_go\_L ;

//

wire ena\_n\_H ;

wire [ 1 : 0 ] ena\_n\_L ;

//

wire [1:0] active ; // 00 - none, 01 - H, 10 - L

wire [3:0] channel ;

//

reg [ 1 : 0 ] trig\_RC ;

reg [ 1 : 0 ] ctrl\_keep\_RC ;

reg [ 1 : 0 ] ctrl\_op\_RC ;

reg [ 1 : 0 ] ctrl\_rdy\_remain\_RC ;

//

round\_robin\_FP #(2) uut\_rr\_FP ( .clk( clk ), .rst\_n( rst\_n ),

.bool\_ready\_H ( bool\_ready\_H ),

.bool\_go\_H ( bool\_go\_H ),

.bool\_ready\_L ( bool\_ready\_L ),

.bool\_go\_L ( bool\_go\_L ),

.ena\_n\_H ( ena\_n\_H ),

.ena\_n\_L ( ena\_n\_L ),

.active ( active ),

.channel ( channel ) );

localparam [1:0] P\_PCF = 2'b00 ;

localparam [1:0] P\_TT = 2'b01 ;

localparam [1:0] P\_RC = 2'b11 ;

localparam [1:0] P\_BE = 2'b10 ;

reg [7:0] cur\_value\_TT ;

wire [7:0] pkt\_len\_TT ;

/\*

q\_server #( P\_TT ) uut\_q\_s\_TT ( .cur\_value( cur\_value\_TT ), .pkt\_len( pkt\_len\_TT ),

.clk( clk ), .rst\_n( rst\_n ),

.ena\_n( ena\_n\_H ),

.bool\_ready( bool\_ready\_H ),

.bool\_go( bool\_go\_H ) );

\*/

q\_server\_3\_states #( P\_TT ) uut\_q\_s\_TT ( .cur\_value( cur\_value\_TT ), .pkt\_len( pkt\_len\_TT ),

.clk( clk ), .rst\_n( rst\_n ),

.ena\_n( ena\_n\_H ),

.bool\_ready( bool\_ready\_H ),

.bool\_go( bool\_go\_H ) );

q\_Q16 #( 0 ) uut\_q\_Q16\_TT ( .cnt\_clk( clk ), // offset = 0

.rst\_n( rst\_n ),

.go( bool\_go\_H ),

.pkt\_len( pkt\_len\_TT ) );

initial begin // clock generator

clk = 0;

forever #5 clk = !clk;

end

initial begin

rst\_n = 1;

#10 rst\_n = 0;

#20 trig\_RC = 2'b00 ;

#10 rst\_n = 1;

#20 ;

cur\_value\_TT = 4 ;

ctrl\_keep\_RC = 2'b11 ;

ctrl\_op\_RC = 2'b11 ;

ctrl\_rdy\_remain\_RC = 2'b11 ;

#40 ;

#100 ;

trig\_RC [ 0 ] = ! trig\_RC [ 0 ] ;

#400 ;

cur\_value\_TT = 0 ;

#600 ;

cur\_value\_TT = 8 ;

#6900 ;

cur\_value\_TT = 3 ;

// TO BE CONTINUE

end

ready\_go\_emulator #( 12'h018 ) emu\_RC0 ( .clk( clk ),

.rst\_n ( rst\_n ), .trigger ( trig\_RC [ 0 ] ) ,

.type\_keep ( ctrl\_keep\_RC [ 0 ] ),

.type\_op ( ctrl\_op\_RC [ 0 ] ),

.type\_rdy\_remain ( ctrl\_rdy\_remain\_RC [ 0 ] ),

.bool\_ready( bool\_ready\_L [ 0 ] ),

.ena\_n ( ena\_n\_L [ 0 ] ),

.bool\_go ( bool\_go\_L [ 0 ] ) );

ready\_go\_emulator #( 12'h00c ) emu\_RC1 ( .clk( clk ),

.rst\_n ( rst\_n ), .trigger ( trig\_RC [ 1 ] ) ,

.type\_keep ( ctrl\_keep\_RC [ 1 ] ),

.type\_op ( ctrl\_op\_RC [ 1 ] ),

.type\_rdy\_remain ( ctrl\_rdy\_remain\_RC [ 1 ] ),

.bool\_ready( bool\_ready\_L [ 1 ] ),

.ena\_n ( ena\_n\_L [ 1 ] ),

.bool\_go ( bool\_go\_L [ 1 ] ) );

endmodule

#### 测试程序test6a

`timescale 1us/100ns

module FPQ\_test6a ;

reg clk ;

reg rst\_n ;

//

wire bool\_ready\_H ;

wire bool\_go\_H ;

wire [ 1 : 0 ] bool\_ready\_L ;

wire [ 1 : 0 ] bool\_go\_L ;

//

wire ena\_n\_H ;

wire [ 1 : 0 ] ena\_n\_L ;

//

wire [1:0] active ; // 00 - none, 01 - H, 10 - L

wire [3:0] channel ;

//

reg [ 1 : 0 ] trig\_RC ;

reg [ 1 : 0 ] ctrl\_keep\_RC ;

reg [ 1 : 0 ] ctrl\_op\_RC ;

reg [ 1 : 0 ] ctrl\_rdy\_remain\_RC ;

//

round\_robin\_FP #(2) uut\_rr\_FP ( .clk( clk ), .rst\_n( rst\_n ),

.bool\_ready\_H ( bool\_ready\_H ),

.bool\_go\_H ( bool\_go\_H ),

.bool\_ready\_L ( bool\_ready\_L ),

.bool\_go\_L ( bool\_go\_L ),

.ena\_n\_H ( ena\_n\_H ),

.ena\_n\_L ( ena\_n\_L ),

.active ( active ),

.channel ( channel ) );

localparam [1:0] P\_PCF = 2'b00 ;

localparam [1:0] P\_TT = 2'b01 ;

localparam [1:0] P\_RC = 2'b11 ;

localparam [1:0] P\_BE = 2'b10 ;

reg [7:0] cur\_value\_TT ;

wire [7:0] pkt\_len\_TT ;

q\_server\_3\_states #( P\_TT ) uut\_q\_s\_TT ( .cur\_value( cur\_value\_TT ), .pkt\_len( pkt\_len\_TT ),

.clk( clk ), .rst\_n( rst\_n ),

.ena\_n( ena\_n\_H ),

.bool\_ready( bool\_ready\_H ),

.bool\_go( bool\_go\_H ) );

q\_Q16\_negedge #( 0 ) uut\_q\_Q16\_TT ( .cnt\_clk( clk ), // offset = 0

.rst\_n( rst\_n ),

.go( bool\_go\_H ),

.pkt\_len( pkt\_len\_TT ) );

initial begin // clock generator

clk = 0;

forever #5 clk = !clk;

end

initial begin

rst\_n = 1;

#10 rst\_n = 0;

#20 trig\_RC = 2'b00 ;

#10 rst\_n = 1;

#20 ;

cur\_value\_TT = 4 ;

ctrl\_keep\_RC = 2'b11 ;

ctrl\_op\_RC = 2'b11 ;

ctrl\_rdy\_remain\_RC = 2'b11 ;

#40 ;

#100 ;

trig\_RC [ 0 ] = ! trig\_RC [ 0 ] ;

#400 ;

cur\_value\_TT = 0 ;

#600 ;

cur\_value\_TT = 8 ;

#6900 ;

cur\_value\_TT = 3 ;

#5000

cur\_value\_TT = 5 ;

// TO BE CONTINUE

end

ready\_go\_emulator #( 12'h018 ) emu\_RC0 ( .clk( clk ),

.rst\_n ( rst\_n ), .trigger ( trig\_RC [ 0 ] ) ,

.type\_keep ( ctrl\_keep\_RC [ 0 ] ),

.type\_op ( ctrl\_op\_RC [ 0 ] ),

.type\_rdy\_remain ( ctrl\_rdy\_remain\_RC [ 0 ] ),

.bool\_ready( bool\_ready\_L [ 0 ] ),

.ena\_n ( ena\_n\_L [ 0 ] ),

.bool\_go ( bool\_go\_L [ 0 ] ) );

ready\_go\_emulator #( 12'h00c ) emu\_RC1 ( .clk( clk ),

.rst\_n ( rst\_n ), .trigger ( trig\_RC [ 1 ] ) ,

.type\_keep ( ctrl\_keep\_RC [ 1 ] ),

.type\_op ( ctrl\_op\_RC [ 1 ] ),

.type\_rdy\_remain ( ctrl\_rdy\_remain\_RC [ 1 ] ),

.bool\_ready( bool\_ready\_L [ 1 ] ),

.ena\_n ( ena\_n\_L [ 1 ] ),

.bool\_go ( bool\_go\_L [ 1 ] ) );

wire bool\_no\_confilict ;

assign bool\_no\_confilict = ~( (bool\_go\_H)&(bool\_go\_L[0]) |

(bool\_go\_H)&(bool\_go\_L[1]) |

(bool\_go\_L[0])&(bool\_go\_L[1]) );

endmodule

#### 测试程序test6b

`timescale 1us/100ns

module FPQ\_test6b ;

reg clk ;

reg rst\_n ;

//

wire bool\_ready\_H ;

wire bool\_go\_H ;

wire [ 1 : 0 ] bool\_ready\_L ;

wire [ 1 : 0 ] bool\_go\_L ;

wire bool\_ready\_RC2 ;

wire bool\_go\_RC2 ;

//

wire ena\_n\_H ;

wire [ 1 : 0 ] ena\_n\_L ;

wire ena\_n\_RC2 ;

//

wire [1:0] active ; // 00 - none, 01 - H, 10 - L

wire [3:0] channel ;

//

reg [ 1 : 0 ] trig\_RC ;

reg [ 1 : 0 ] ctrl\_keep\_RC ;

reg [ 1 : 0 ] ctrl\_op\_RC ;

reg [ 1 : 0 ] ctrl\_rdy\_remain\_RC ;

//

round\_robin\_FP #(3) uut\_rr\_FP ( .clk( clk ), .rst\_n( rst\_n ),

.bool\_ready\_H ( bool\_ready\_H ),

.bool\_go\_H ( bool\_go\_H ),

.bool\_ready\_L ( { bool\_ready\_RC2 , bool\_ready\_L } ),

.bool\_go\_L ( { bool\_go\_RC2 , bool\_go\_L } ),

.ena\_n\_H ( ena\_n\_H ),

.ena\_n\_L ( { ena\_n\_RC2 , ena\_n\_L } ),

.active ( active ),

.channel ( channel ) );

localparam [1:0] P\_PCF = 2'b00 ;

localparam [1:0] P\_TT = 2'b01 ;

localparam [1:0] P\_RC = 2'b11 ;

localparam [1:0] P\_BE = 2'b10 ;

reg [7:0] cur\_value\_TT ;

wire [7:0] pkt\_len\_TT ;

reg [7:0 ] cur\_value\_RC2 ;

wire [7:0] pkt\_len\_RC2 ;

q\_server\_3\_states #( P\_TT ) uut\_q\_s\_TT ( .cur\_value( cur\_value\_TT ), .pkt\_len( pkt\_len\_TT ),

.clk( clk ), .rst\_n( rst\_n ),

.ena\_n( ena\_n\_H ),

.bool\_ready( bool\_ready\_H ),

.bool\_go( bool\_go\_H ) );

q\_Q16\_negedge #( 0 ) uut\_q\_Q16\_TT ( .cnt\_clk( clk ), // offset = 0

.rst\_n( rst\_n ),

.go( bool\_go\_H ),

.pkt\_len( pkt\_len\_TT ) );

q\_server\_3\_states #( P\_RC ) uut\_q\_s\_RC2 ( .cur\_value( cur\_value\_RC2 ), .pkt\_len( pkt\_len\_RC2 ),

.clk( clk ), .rst\_n( rst\_n ),

.ena\_n( ena\_n\_RC2 ),

.bool\_ready( bool\_ready\_RC2 ),

.bool\_go( bool\_go\_RC2 ) );

q\_Q16\_negedge #( 2 ) uut\_q\_Q16\_RC2 ( .cnt\_clk( clk ), // offset = 2

.rst\_n( rst\_n ),

.go( bool\_go\_RC2 ),

.pkt\_len( pkt\_len\_RC2 ) );

initial begin // clock generator

clk = 0;

forever #5 clk = !clk;

end

initial begin

rst\_n = 1;

#10 rst\_n = 0;

#20 trig\_RC = 2'b00 ;

#10 rst\_n = 1;

#20 ;

cur\_value\_TT = 4 ;

cur\_value\_RC2 = 0 ;

ctrl\_keep\_RC = 2'b11 ;

ctrl\_op\_RC = 2'b11 ;

ctrl\_rdy\_remain\_RC = 2'b11 ;

#40 ;

#100 ;

trig\_RC [ 0 ] = ! trig\_RC [ 0 ] ;

#400 ;

cur\_value\_TT = 0 ;

#10 ;

cur\_value\_RC2 = 8 ;

#10 ;

trig\_RC [ 1] = ! trig\_RC [ 1 ] ;

#180 ;

cur\_value\_RC2 = 4 ;

#150 ;

cur\_value\_RC2 = 2 ;

#150 ;

cur\_value\_RC2 = 0 ;

cur\_value\_TT = 8 ;

#2900 ;

cur\_value\_TT = 3 ;

#1000 ;

cur\_value\_TT = 5 ;

#1000 ;

cur\_value\_TT = 0 ;

#1010 ;

cur\_value\_RC2 = 8 ;

#490 ; cur\_value\_RC2 = 7 ;

#500 ; cur\_value\_RC2 = 6 ;

#500 ; cur\_value\_RC2 = 5 ;

#500 ; cur\_value\_RC2 = 4 ;

#500 ; cur\_value\_RC2 = 3 ;

#500 ; cur\_value\_RC2 = 2 ;

#500 ; cur\_value\_RC2 = 1 ;

#500 ; cur\_value\_RC2 = 0 ;

// TO BE CONTINUE

end

ready\_go\_emulator #( 12'h018 ) emu\_RC2 ( .clk( clk ),

.rst\_n ( rst\_n ), .trigger ( trig\_RC [ 0 ] ) ,

.type\_keep ( ctrl\_keep\_RC [ 0 ] ),

.type\_op ( ctrl\_op\_RC [ 0 ] ),

.type\_rdy\_remain ( ctrl\_rdy\_remain\_RC [ 0 ] ),

.bool\_ready( bool\_ready\_L [ 0 ] ),

.ena\_n ( ena\_n\_L [ 0 ] ),

.bool\_go ( bool\_go\_L [ 0 ] ) );

ready\_go\_emulator #( 12'h00c ) emu\_RC1 ( .clk( clk ),

.rst\_n ( rst\_n ), .trigger ( trig\_RC [ 1 ] ) ,

.type\_keep ( ctrl\_keep\_RC [ 1 ] ),

.type\_op ( ctrl\_op\_RC [ 1 ] ),

.type\_rdy\_remain ( ctrl\_rdy\_remain\_RC [ 1 ] ),

.bool\_ready( bool\_ready\_L [ 1 ] ),

.ena\_n ( ena\_n\_L [ 1 ] ),

.bool\_go ( bool\_go\_L [ 1 ] ) );

wire bool\_no\_confilict ;

assign bool\_no\_confilict = ~( (bool\_go\_H)&(bool\_go\_L[0]) |

(bool\_go\_H)&(bool\_go\_L[1]) |

(bool\_go\_L[0])&(bool\_go\_L[1]) );

endmodule

#### 测试程序test7

`timescale 1us/100ns

module FPQ\_test7 ;

reg clk ;

reg rst\_n ;

//

wire bool\_ready\_H ;

wire bool\_go\_H ;

wire [ 1 : 0 ] bool\_ready\_L ;

wire [ 1 : 0 ] bool\_go\_L ;

wire bool\_ready\_RC2 ;

wire bool\_go\_RC2 ;

//

wire ena\_n\_H ;

wire [ 1 : 0 ] ena\_n\_L ;

wire ena\_n\_RC2 ;

//

wire [1:0] active ; // 00 - none, 01 - H, 10 - L

wire [3:0] channel ;

//

reg [ 1 : 0 ] trig\_RC ;

reg [ 1 : 0 ] ctrl\_keep\_RC ;

reg [ 1 : 0 ] ctrl\_op\_RC ;

reg [ 1 : 0 ] ctrl\_rdy\_remain\_RC ;

//

wire [ 1 : 0 ] cur\_state ;

wire [ 7 : 0 ] cur\_value ;

wire [ 7 : 0 ] cur\_value\_RC ;

wire [7:0] cur\_value\_TT ;

wire [7:0] pkt\_len\_TT ;

//

timetable uut\_timetable ( .cnt\_clk( clk ), .rst\_n( rst\_n ),

.ena\_n( 1'b0 ), .cur\_value( cur\_value ) ,

.cur\_state( cur\_state ) );

//

cur\_value\_translator uut\_trans ( .cur\_value( cur\_value ) ,

.cur\_state( cur\_state ) , .cnt\_for\_TT( cur\_value\_TT ) ,

.cnt\_for\_RC( cur\_value\_RC ) );

//

round\_robin\_FP #(3) uut\_rr\_FP ( .clk( clk ), .rst\_n( rst\_n ),

.bool\_ready\_H ( bool\_ready\_H ),

.bool\_go\_H ( bool\_go\_H ),

.bool\_ready\_L ( { bool\_ready\_RC2 , bool\_ready\_L } ),

.bool\_go\_L ( { bool\_go\_RC2 , bool\_go\_L } ),

.ena\_n\_H ( ena\_n\_H ),

.ena\_n\_L ( { ena\_n\_RC2 , ena\_n\_L } ),

.active ( active ),

.channel ( channel ) );

localparam [1:0] P\_PCF = 2'b00 ;

localparam [1:0] P\_TT = 2'b01 ;

localparam [1:0] P\_RC = 2'b11 ;

localparam [1:0] P\_BE = 2'b10 ;

reg [ 7:0] cur\_value\_RC2 ;

wire [7:0] pkt\_len\_RC2 ;

q\_server\_3\_states #( P\_TT ) uut\_q\_s\_TT ( .cur\_value( cur\_value\_TT ), .pkt\_len( pkt\_len\_TT ),

.clk( clk ), .rst\_n( rst\_n ),

.ena\_n( ena\_n\_H ),

.bool\_ready( bool\_ready\_H ),

.bool\_go( bool\_go\_H ) );

q\_Q16\_negedge #( 0 ) uut\_q\_Q16\_TT ( .cnt\_clk( clk ), // offset = 0

.rst\_n( rst\_n ),

.go( bool\_go\_H ),

.pkt\_len( pkt\_len\_TT ) );

q\_server\_3\_states #( P\_RC ) uut\_q\_s\_RC2 ( .cur\_value( cur\_value\_RC2 ), .pkt\_len( pkt\_len\_RC2 ),

.clk( clk ), .rst\_n( rst\_n ),

.ena\_n( ena\_n\_RC2 ),

.bool\_ready( bool\_ready\_RC2 ),

.bool\_go( bool\_go\_RC2 ) );

q\_Q16\_negedge #( 2 ) uut\_q\_Q16\_RC2 ( .cnt\_clk( clk ), // offset = 2

.rst\_n( rst\_n ),

.go( bool\_go\_RC2 ),

.pkt\_len( pkt\_len\_RC2 ) );

initial begin // clock generator

clk = 0;

forever #5 clk = !clk;

end

initial begin

rst\_n = 1;

#10 rst\_n = 0;

#20 trig\_RC = 2'b00 ;

#10 rst\_n = 1;

#20 ;

cur\_value\_RC2 = 0 ;

ctrl\_keep\_RC = 2'b11 ;

ctrl\_op\_RC = 2'b11 ;

ctrl\_rdy\_remain\_RC = 2'b11 ;

#40 ;

#100 ;

trig\_RC [ 0 ] = ! trig\_RC [ 0 ] ;

#400 ;

#10 ;

cur\_value\_RC2 = 8 ;

#10 ;

trig\_RC [ 1] = ! trig\_RC [ 1 ] ;

#180 ;

cur\_value\_RC2 = 4 ;

#150 ;

cur\_value\_RC2 = 2 ;

#150 ;

cur\_value\_RC2 = 0 ;

#2900 ;

#1000 ;

#1000 ;

#1010 ;

cur\_value\_RC2 = 8 ;

#490 ; cur\_value\_RC2 = 7 ;

#500 ; cur\_value\_RC2 = 6 ;

#500 ; cur\_value\_RC2 = 5 ;

#500 ; cur\_value\_RC2 = 4 ;

#500 ; cur\_value\_RC2 = 3 ;

#500 ; cur\_value\_RC2 = 2 ;

#500 ; cur\_value\_RC2 = 1 ;

#500 ; cur\_value\_RC2 = 0 ;

// TO BE CONTINUE

end

ready\_go\_emulator #( 12'h018 ) emu\_RC2 ( .clk( clk ),

.rst\_n ( rst\_n ), .trigger ( trig\_RC [ 0 ] ) ,

.type\_keep ( ctrl\_keep\_RC [ 0 ] ),

.type\_op ( ctrl\_op\_RC [ 0 ] ),

.type\_rdy\_remain ( ctrl\_rdy\_remain\_RC [ 0 ] ),

.bool\_ready( bool\_ready\_L [ 0 ] ),

.ena\_n ( ena\_n\_L [ 0 ] ),

.bool\_go ( bool\_go\_L [ 0 ] ) );

ready\_go\_emulator #( 12'h00c ) emu\_RC1 ( .clk( clk ),

.rst\_n ( rst\_n ), .trigger ( trig\_RC [ 1 ] ) ,

.type\_keep ( ctrl\_keep\_RC [ 1 ] ),

.type\_op ( ctrl\_op\_RC [ 1 ] ),

.type\_rdy\_remain ( ctrl\_rdy\_remain\_RC [ 1 ] ),

.bool\_ready( bool\_ready\_L [ 1 ] ),

.ena\_n ( ena\_n\_L [ 1 ] ),

.bool\_go ( bool\_go\_L [ 1 ] ) );

wire bool\_no\_confilict ;

assign bool\_no\_confilict = ~( (bool\_go\_H)&(bool\_go\_L[0]) |

(bool\_go\_H)&(bool\_go\_L[1]) |

(bool\_go\_L[0])&(bool\_go\_L[1]) );

endmodule

#### 测试程序test7a

`timescale 1us/100ns

module FPQ\_test7a ;

reg clk ;

reg rst\_n ;

//

wire bool\_ready\_H ;

wire bool\_go\_H ;

wire [ 1 : 0 ] bool\_ready\_L ;

wire [ 1 : 0 ] bool\_go\_L ;

wire bool\_ready\_RC2 ;

wire bool\_go\_RC2 ;

//

wire ena\_n\_H ;

wire [ 1 : 0 ] ena\_n\_L ;

wire ena\_n\_RC2 ;

//

wire [1:0] active ; // 00 - none, 01 - H, 10 - L

wire [3:0] channel ;

//

reg [ 1 : 0 ] trig\_RC ;

reg [ 1 : 0 ] ctrl\_keep\_RC ;

reg [ 1 : 0 ] ctrl\_op\_RC ;

reg [ 1 : 0 ] ctrl\_rdy\_remain\_RC ;

//

wire [ 1 : 0 ] cur\_state ;

wire [ 7 : 0 ] cur\_value ;

wire [ 7 : 0 ] cur\_value\_RC ;

wire [7:0] cur\_value\_TT ;

wire [7:0] pkt\_len\_TT ;

//

timetable uut\_timetable ( .cnt\_clk( clk ), .rst\_n( rst\_n ),

.ena\_n( 1'b0 ), .cur\_value( cur\_value ) ,

.cur\_state( cur\_state ) );

//

cur\_value\_translator uut\_trans ( .cur\_value( cur\_value ) ,

.cur\_state( cur\_state ) , .cnt\_for\_TT( cur\_value\_TT ) ,

.cnt\_for\_RC( cur\_value\_RC ) );

//

round\_robin\_FP #(3) uut\_rr\_FP ( .clk( clk ), .rst\_n( rst\_n ),

.bool\_ready\_H ( bool\_ready\_H ),

.bool\_go\_H ( bool\_go\_H ),

.bool\_ready\_L ( { bool\_ready\_RC2 , bool\_ready\_L } ),

.bool\_go\_L ( { bool\_go\_RC2 , bool\_go\_L } ),

.ena\_n\_H ( ena\_n\_H ),

.ena\_n\_L ( { ena\_n\_RC2 , ena\_n\_L } ),

.active ( active ),

.channel ( channel ) );

localparam [1:0] P\_PCF = 2'b00 ;

localparam [1:0] P\_TT = 2'b01 ;

localparam [1:0] P\_RC = 2'b11 ;

localparam [1:0] P\_BE = 2'b10 ;

// reg [ 7:0] cur\_value\_RC2 ;

wire [7:0] pkt\_len\_RC2 ;

q\_server\_3\_states #( P\_TT ) uut\_q\_s\_TT ( .cur\_value( cur\_value\_TT ), .pkt\_len( pkt\_len\_TT ),

.clk( clk ), .rst\_n( rst\_n ),

.ena\_n( ena\_n\_H ),

.bool\_ready( bool\_ready\_H ),

.bool\_go( bool\_go\_H ) );

q\_Q16\_negedge #( 0 ) uut\_q\_Q16\_TT ( .cnt\_clk( clk ), // offset = 0

.rst\_n( rst\_n ),

.go( bool\_go\_H ),

.pkt\_len( pkt\_len\_TT ) );

q\_server\_3\_states #( P\_RC ) uut\_q\_s\_RC2 ( .cur\_value( cur\_value\_RC ), .pkt\_len( pkt\_len\_RC2 ),

.clk( clk ), .rst\_n( rst\_n ),

.ena\_n( ena\_n\_RC2 ),

.bool\_ready( bool\_ready\_RC2 ),

.bool\_go( bool\_go\_RC2 ) );

q\_Q16\_negedge #( 2 ) uut\_q\_Q16\_RC2 ( .cnt\_clk( clk ), // offset = 2

.rst\_n( rst\_n ),

.go( bool\_go\_RC2 ),

.pkt\_len( pkt\_len\_RC2 ) );

initial begin // clock generator

clk = 0;

forever #5 clk = !clk;

end

initial begin

rst\_n = 1;

#10 rst\_n = 0;

#20 trig\_RC = 2'b00 ;

#10 rst\_n = 1;

#20 ;

// cur\_value\_RC2 = 0 ;

ctrl\_keep\_RC = 2'b11 ;

ctrl\_op\_RC = 2'b11 ;

ctrl\_rdy\_remain\_RC = 2'b11 ;

#40 ;

#100 ;

trig\_RC [ 0 ] = ! trig\_RC [ 0 ] ;

#400 ;

#10 ;

// cur\_value\_RC2 = 8 ;

#10 ;

trig\_RC [ 1] = ! trig\_RC [ 1 ] ;

#180 ;

// cur\_value\_RC2 = 4 ;

#150 ;

// cur\_value\_RC2 = 2 ;

#150 ;

// cur\_value\_RC2 = 0 ;

#2900 ;

#1000 ;

#1000 ;

#1010 ;

// cur\_value\_RC2 = 8 ;

#490 ; // cur\_value\_RC2 = 7 ;

#500 ; // cur\_value\_RC2 = 6 ;

#500 ; // cur\_value\_RC2 = 5 ;

#500 ; // cur\_value\_RC2 = 4 ;

#500 ; // cur\_value\_RC2 = 3 ;

#500 ; // cur\_value\_RC2 = 2 ;

#500 ; // cur\_value\_RC2 = 1 ;

#500 ; // cur\_value\_RC2 = 0 ;

// TO BE CONTINUE

end

ready\_go\_emulator #( 12'h018 ) emu\_RC2 ( .clk( clk ),

.rst\_n ( rst\_n ), .trigger ( trig\_RC [ 0 ] ) ,

.type\_keep ( ctrl\_keep\_RC [ 0 ] ),

.type\_op ( ctrl\_op\_RC [ 0 ] ),

.type\_rdy\_remain ( ctrl\_rdy\_remain\_RC [ 0 ] ),

.bool\_ready( bool\_ready\_L [ 0 ] ),

.ena\_n ( ena\_n\_L [ 0 ] ),

.bool\_go ( bool\_go\_L [ 0 ] ) );

ready\_go\_emulator #( 12'h00c ) emu\_RC1 ( .clk( clk ),

.rst\_n ( rst\_n ), .trigger ( trig\_RC [ 1 ] ) ,

.type\_keep ( ctrl\_keep\_RC [ 1 ] ),

.type\_op ( ctrl\_op\_RC [ 1 ] ),

.type\_rdy\_remain ( ctrl\_rdy\_remain\_RC [ 1 ] ),

.bool\_ready( bool\_ready\_L [ 1 ] ),

.ena\_n ( ena\_n\_L [ 1 ] ),

.bool\_go ( bool\_go\_L [ 1 ] ) );

wire bool\_no\_confilict ;

assign bool\_no\_confilict = ~( (bool\_go\_H)&(bool\_go\_L[0]) |

(bool\_go\_H)&(bool\_go\_L[1]) |

(bool\_go\_L[0])&(bool\_go\_L[1]) );

endmodule

#### 测试程序test8

`timescale 100ns / 10ns

module FPQ\_test8 ;

reg clk\_1MHz ;

reg rst\_n ;

wire [ 1:0 ] active ;

wire [ 3:0 ] channel ;

// Because of 12MHz being difficult to simu, instead of 1MHz

initial begin

clk\_1MHz = 0;

forever #5 clk\_1MHz = ! clk\_1MHz ;

end

initial begin

rst\_n = 1;

#10 rst\_n = 0;

#10 rst\_n = 1;

#20 ;

end

FPQ\_top\_1TT\_2RC uut( .clk\_12MHz ( clk\_1MHz ), .rst\_n( rst\_n ),

.active( active ), .channel( channel ) );

// for "actice" 00 - none, 01 - H, 10 - L

endmodule

#### 测试程序test9

`timescale 100ns / 10ns

module FPQ\_test9 ;

reg clk\_1MHz ;

reg rst\_n ;

wire [ 1:0 ] active ;

wire [ 3:0 ] channel ;

wire [ 8:0 ] seg\_LED\_1 ;

wire [ 8:0 ] seg\_LED\_2 ;

wire [ 8:1 ] LEDs\_n ;

// Because of 12MHz being difficult to simu, instead of 1MHz

initial begin

clk\_1MHz = 0;

forever #5 clk\_1MHz = ! clk\_1MHz ;

end

initial begin

rst\_n = 1;

#10 rst\_n = 0;

#10 rst\_n = 1;

#20 ;

end

/\*

FPQ\_top\_1TT\_2RC uut( .clk\_12MHz ( clk\_1MHz ), .rst\_n( rst\_n ),

.active( active ), .channel( channel ) );

// for "actice" 00 - none, 01 - H, 10 - L

\*/

FPQ\_top\_1TT\_2RC\_with\_ui uut( .clk\_12MHz ( clk\_1MHz ), .key\_rst\_n( rst\_n ),

.seg\_LED\_1 ( seg\_LED\_1 ) , .seg\_LED\_2 ( seg\_LED\_2 ) , .LEDs\_n ( LEDs\_n ) );

reg test\_key\_deb\_pulse ;

wire test\_rst\_n ;

initial begin

test\_key\_deb\_pulse = 1 ;

#2000 ;

test\_key\_deb\_pulse = 0 ;

#200 ;

test\_key\_deb\_pulse = 1 ;

#200 ;

test\_key\_deb\_pulse = 0 ;

#3600 ;

test\_key\_deb\_pulse = 1 ;

#200 ;

test\_key\_deb\_pulse = 0 ;

#200 ;

test\_key\_deb\_pulse = 1 ;

end

key\_debounce # ( 16'd100 ) // 1MHz, 100us

uut\_key\_deb ( .clk( clk\_1MHz ), .rst\_n( rst\_n ),

.key\_n( test\_key\_deb\_pulse ),

.key\_pulse\_n( test\_rst\_n ) );

endmodule