行为级建模和时间约束

简介

行为级建模在 Lab1 中被作为三个广泛使用的建模方式之一被介绍过了。在 Lab4 中更深入的介绍了关于 testbench 的额外功能。但是在这个建模方式中还有更多可用的对于复杂时序数字电路设计非常有用的结构。时序电路需要时钟，有了时钟信号，总是会有一个频率或是速度使得电路可以依此运转。通过 XDF 文件中特定的时间限制可以将所想要设定的速度传递给工具。在本次实验中你将会学到更多语言结构和时间约束概念。*请参阅 Vivado 教程来了解如何使用 Vivado 工具来创建工程并验证数字电路。*

目标

完成本次实验后，你将能：

- 使用行为级建模来使用各种语言结构
- 通过时间约束传达时钟需求

行为级建模 第一部分

如之前的实验中所说，设计的行为可以建模的主要机制是：`initial` 和 `always` 语句。`initial` 语句主要用在 testbench 中用来在需要的时候产生一个输入。而 `always` 语句主要用来秒数电路的功能。`initial` 和 `always` 语句都可能有简单的或是一块（用 begin … end 包围）的过程语句。

一个过程语句是以下之一：

procedural_assignment（阻塞的或是非阻塞的）
condition_statement
case_statement
loop_statement
wait_statement
event_trigger
sequential_block
task（用户或是系统）

当多个过程语句被 begin … end 包围，它们会串行地执行。由于 `always` 语句会不断地执行，它们通常使用延迟控制或是事件控制机制来控制。这里是一个使用延迟控制的过程语句：

```
always
   #5 CLK = ~CLK;
```

在上面的例子中，这条语句每过 5 个单位的 Verilog 代码中定义的时间就会执行一次，每次执行的时候就会把信号值翻转一次，从而产生一个 10 个单位时间的周期时钟。语句`#5 CLK = ~CLK` 可看作一个延迟控制，意即遇到该语句和实际执行该语句的时间延迟是 5 个时间单位。当延迟写在左边时（如上面的情况），也可看作是语句间延迟（inter-statement delay），

© copyright 2015 Xilinx

即该语句被阻塞那么长的时间然后才被运行并将结果赋给目标。下面的例子展示了语句间延迟的效果：

```
initial
begin
    #5 SIG1 = 3;
    #4 SIG1 = 7;
    #2 SIG1 = 4;
end
```

这里的 SIG1 会在单位时间 5 得到值 3，在单位时间 9 得到值 7，在单位时间 11 得到值 4。

```
wire test;
always @(test)
begin
    #5 CLK = ~CLK;
end
```

上面的 `always` 语句只会在线网型变量 *test* 的值发生变化（一个事件）时才会执行。值的改变可能是 0->1，1->0，0->x，x->1，1->z，z->0，0->z，z->1 或是 1->x。当事件发生时，CLK 的逻辑数值会在 5 个时间单位后翻转。

```
wire test;
always @(posedge test)
begin
    #5 CLK = ~CLK;
end
```

上面的 `always` 语句只会在线网型变量 *test* 的值有上升沿变化 (0->1, 0->x, 0->z, z->1, x->1）时执行。当事件发生时，CLK 的逻辑数值会在 5 个时间单位后翻转。这样的事件被称为边沿触发事件。与边沿触发事件相反，可能会有另一种事件，叫做电平敏感时序控制。

```
wait (SUM > 22)
    SUM = 0;

wait (DATA_READY)
    DATA = BUS;
```

在上面的例子中，只有当 SUM 大于 22 时 SUM 才会被赋值为 0，而只要 DATA_READY 被置位无论 BUS 上是何值 DATA 都会被赋给 BUS 的值。

到现在，我们已经了解了可以帮我们在代码行为中建立一个惯性延迟的语句间延迟。还有另一种延迟，叫做语句内延迟（intra-statement），它是用来在赋值语句中建立一个传输延迟。

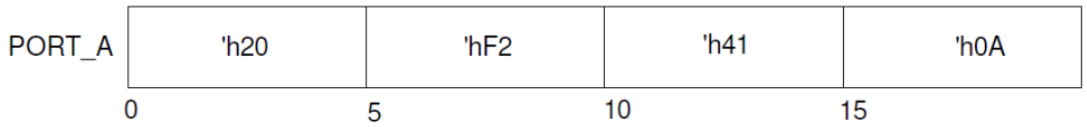
www.xilinx.com/university
xup@xilinx.com
© copyright 2015 Xilinx
XILINX

这里是它的一个例子：

```
DONE = #5 1'b1;
```

在这个语句中，语句内延迟在赋值符号的右边被提到。右边的表达式在遇到该语句的时候就会计算，但是表达式的结果只有在语句延迟之后才被赋值。

1-1. 通过使用语句间延迟给名为 PORT_A 的端口生成如下的波形来写一个 testbench

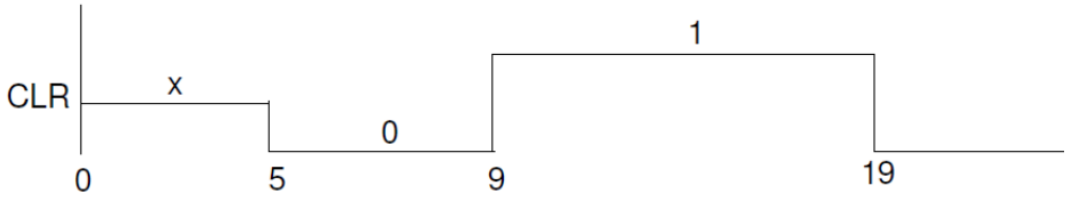


过程赋值语句中的“=”被用作阻塞过程语句（blocking procedural assignment）。正如其名，随后的语句在当前的赋值完成之前都是被阻塞的。这里是一个解释此概念的例子：

```
always @(A or B or CIN)
begin
    reg T1, T2, T3;
    T1 = A & B;
    T2 = B & CIN;
    T3 = A & CIN;
end
```

T1 的赋值最先发生，T1 被计算出来，然后第二条语句被执行，T2 被赋值，然后第三条语句被执行，然后 T3 被赋值。这里是另一个例子。

```
initial
begin
    CLR = #5 0;
    CLR = #4 1;
    CLR = #10 0;
end
```



另一种赋值符号，该赋值符号“<=”被称为非阻塞的（non-blocking）。使用非阻塞符号的语句不会阻塞执行。它的赋值是计划在将来发生的。当非阻塞赋值执行时，右边的表达式在当

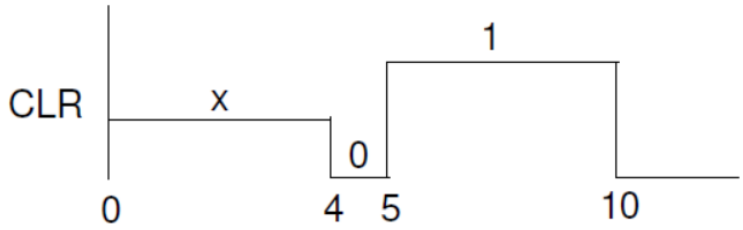
© copyright 2015 Xilinx

时被计算出来，它的值被计划赋给左手边的目标，而下一条语句会紧接着执行。非阻塞语句被广泛使用于在一个期望的时钟事件发生时在多个寄存器间的内容传输（经常是并行的）。

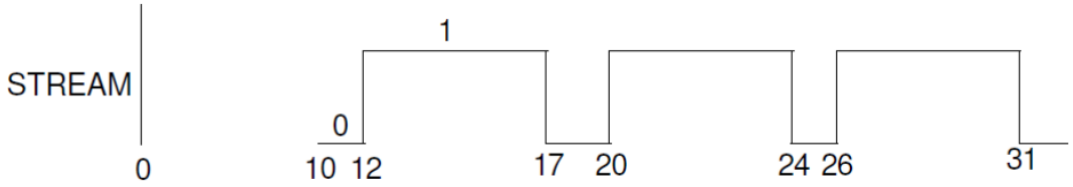
```
always @(posedge CLK)
begin
    T1 <= #5 A & B;
    T2 <= #8 B & CIN;
    T3 <= #2 A & CIN;
end
```

在这个例子中，当一个时钟上升沿事件发生时，A，B，和 CIN 的值被记下来（被捕获），然后 T1 在 5 个时间单位的延迟后值更新，T2 在 8 个时间单位的延迟之后值更新，T3 在 2 个时间单位延迟之后更新，这些都是在时钟信号的同一个上升沿之后。这里是另一个生成波形图输出的例子。

```
initial
begin
    CLR <= #5 1;
    CLR <= #4 0;
    CLR <= #10 0;
end
```



1-2. 通过使用语句内延迟生成如下的波形来写一个 testbench



到此我们已经了解了一些能无条件的产生刺激信号的结构。然而，很多时候我们也想能对不同的条件有不同的刺激信号。Verilog HDL 提供了控制更改语句例如，**if, if ... else** 和 **if ... else if**。一个 **if** 语句的大体结构如下：

```
if (conditon-1)
    procedural_statement
```


www.xilinx.com/university
xup@xilinx.com
© copyright 2015 Xilinx

```
[else if (condition-2)
   procedural_statement]
[else
   procedural_statement]
```

如果上面的 procedural_statement 由多于一个语句组成，那么它们就会被一个 `begin...end` 包围。

有可能会有嵌套 **if** 语句。在这种情况下，**else** 部分会被与最近的 **if** 部分配对。例如下面的，这里的 **else** 部分与 **if (**RESET**)**条件配对，

```
if (CLK)
   if (RESET)
      Q <= 0;
   else
      Q <= D;
```

if 语句通常用来创建一个优先结构，它会给在条件中列为第一个的最高优先级。

1-3. 使用 if-else-if 语句写一个设计一个 1 比特 4-1 选择器的行为级模型。编写一个 testbench 来验证设计。将四个输入管道赋给 SW3-SW0（SW3 被赋给最高位管道，有最低的优先级，其余同理），将选择线路赋给 SW5-SW4，将输出赋给 LED0。使用 Basys3 或者 Nexys4 DDR 开发板来在硬件中验证你的设计。查看 Project Summary report 来确保没有使用或提及任何锁存器或寄存器资源。

另一个广泛使用的语句是 case 语句。case 语句经常被用在我们想要创建一个并行的结构（没有优先级）时。Case 语句经常被用于创建有限状态机。case 语句的语法是：

```
case [case_expression]
   case_item_expr [, case_item_expression]: procedural_statement
   ...
   ...
   [default: procedural_statement]
endcase
```

case_expression 会被最先计算（只要是有事件的时候），其值会被按照 case_item_expr 列的顺序与它们匹配。当匹配成功时，对应的 procedural_statement 被执行。procedural_statement

© copyright 2015 Xilinx

如果由多条语句组成就会被一个 begin...end 块包围。default 情况包含了所有没被其他的任何一个 case_item_expr 情况包含的值。在 case_expression 中，x 和 z（如果存在的话）的确会与 case_item_expr 比较。也就是说它们并没有被不当作一种情况考虑。如果你想把它们视作不用考虑的，你可以使用 casez 或是 casez 来代替 case。在 casez 语句中，case_expression 和 case_item_expr 中出现的值 z 不用考虑。在 casex 语句中，值 x 和值 z 都可以不用考虑。

1-4. 使用 case 语句设计一个格雷码生成器。该设计会需要通过 SW3-SW0 输入一个 4 比特 BCD 码，在 LED3-LED0 这四个 LED 灯上输出对应的格雷码，SW4 提供的使能输入应该是 TRUE。如果使能输入为 FALSE 或是输入不是 BCD 码那么 LED3-LED0 应该都被点亮 LED4 也应该被点亮。查看 Project Summary report 来确保没有使用或提及任何锁存器或寄存器资源。

Verilog HDL 还支持多种 loop 语句来多次执行同一个功能。支持的 Loop 语句如下：

```
forever loop
repeat loop
while loop
for loop
```

forever loop 语句被用在一个过程语句需要被连续地执行时。如果需要一个周期性的输出那么过程语句中就必须有某种时钟控制。举个例子，为了生成一个 20 个单位时间周期的时钟，可使用如下的代码。

```
initial
begin
    CLK = 0;
    forever
        #10 CLK = ~CLK;
end
```

repeat loop 语句用在当过程语句需要被连续地执行特定的次数时。提示：如果循环次数表达式是一个 x 值或是一个 z 值，那么这个循环次数会被当作 0。

```
repeat (COUNT)
    SUM = SUM + 5
```

While loop 语句中的过程语句会一直执行直到某个条件变成 false 值。

www.xilinx.com/university
xup@xilinx.com
© copyright 2015 Xilinx

```
while (COUNT < COUNT_LIMIT)
    SUM = SUM + 5
```

for loop 语句在当过程语句需要被连续地执行特定的次数时。与 repeat 语句不一样，它会使用一个索引变量。这个索引变量可以被初始化为任何一个想初始化的值，更进一步地，它可以被更新为任何一个要更新为的值，并且可以给出一个条件来中止 loop 语句。Loop 的索引变量一般会被定义为一个整型的。这里是一个 loop 语句的例子。

```
integer K;
for (K = 0; k < COUNT_LIMIT; K = K + 1)
    SUM = SUM + K;
```

1-5. 写一个能按下所述的序列计数的计数器模型。计数器需要使用行为级建模和 case 语句。编写一个 testbench 来对它做测试。testbench 需要将计数器的输出显示在仿真的控制台输出中。仿真 200ns 使用每周期 10 个时间单位的时钟。（计数序列为）000，001， 011，101，111，（重复 000）。计数器要有一个使能信号（SW1），一个重置信号（SW0）,和一个时钟信号（SW15）。计数器输出到 LED2-LED0。

```
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets { clk }];
```

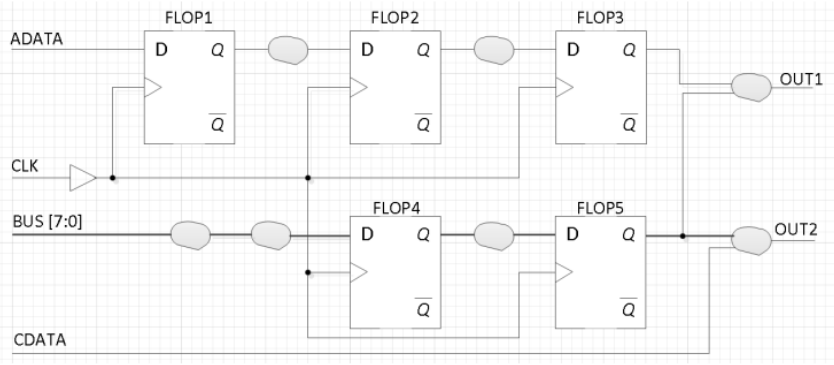
将上述代码添加到 XDC 文件中并将时钟设置为 SW15。

时间约束 第二部分

在组合逻辑设计中，电路的延迟取决于逻辑层的数目，每个线网的扇出（一个线网驱动的门输入数），输出线网加载的电容。当这种电路被放置在触发器或是寄存器之间时，它们会影响操作时序电路的时钟的速度。如果预期的性能通过时间约束传达给它们，那么综合（synthesis）和（implementation）工具会将这些设计打包成查找表（LUT），触发器，和寄存器，同时将它们放在合适的位置。时间约束可以被分为全局时间（global timing constraints）和特定线路约束（path specific constraints）。特定线路约束相比全局时间约束有着更高的优先级，在特定线路约束中使用的元件也会被最先放置连线。

全局时间约束使用非常少的几行指令就能覆盖设计的大部分。在任何一个纯组合逻辑电路中线路到线路的约束是用来描述电路能承受的延迟的。在时序电路中会使用周期，输入延迟和输出延迟约束。这四种时间约束见下图。

© copyright 2015 Xilinx



在上图中，覆盖 ADATA 输入和 FLOP1 的 D 端口的线路，BUS 输入和 FLOP4 的 D 端口之前的线路能被一个叫做 SET_INPUT_DELAY 命令的约束所约束。set_input_delay 命令显示了在上游设备中一个触发器的 Q 端口中会花费多长时间，上游设备的线路延迟和开发板延迟。（综合和实现）工具会从命令中列出的时钟信号的时钟周期中减去那些延迟，并使用计算的结果延迟来在输入和触发器的 D 输入之间放置布线。它还会考虑时钟信号到达目标触发器（比如上图中的 FLPO1）的时钟端口时经历的延迟。max 和 min 限定符用于建立和保持检查。

FLOP3 的 Q 端口和输出 OUT1 之间的线路，FLOP5 的 Q 端口和 OUT1 之间的线路，FLOP5 的 Q 端口和 OUT2 之间的线路能被 SET_OUTPUT_DELAY 命令约束。同样，提到的该延迟指开发板延迟中有多大的延迟，线路延迟和下游设备中触发器的启动延迟。

CDTATA 和 OUT2 之间的线路可以被 SET_MAX_DELAY 约束所约束。

FLOP1 的 Q 端口和 FLOP2 的 D 端口之间的线路，FLOP2 的 Q 端口和 FLOP3 的 D 端口之间的线路，FLOP4 的 Q 端口和 FLOP5 的 D 端口之间的线路能被周期约束所约束。周期约束使用 create_clock 命令创建的。create_clock 命令可能会提及 FPGA 设计的一个引脚或者不提及任何引脚。如果没有提及时钟引脚，就会创建一个虚拟时钟。如果引脚被提及，周期参数（period parameter）指出上升到上升沿延迟，而 waveform 选项则指出上升沿何时发生，第二个数字指出下降沿何时发生。Waveform 选项可用来创建占空比不是 50%的时钟和/或相位延迟的时钟信号。

```
create_clock -name CLK -period 10.0 -waveform (0 5.0) [get_ports CLK]
set_input_delay -clock CLK -max 3.0 [all_inputs]
set_input_delay -clock CLK -min 1.0 [all_inputs]
set_output_delay -clock CLK 2.0 [all_outputs]
set_max_delay 5.0 -from [get_ports CDATA] -to [get_ports OUT2]
```

注意这里的时钟周期被定义为 10ns。为了一致性这在整个例子中都适用。更多各个约束类型的语法细节可在 the Vivado Using Constraints Guide 中的 UG903 内找到。

结论

www.xilinx.com/university
xup@xilinx.com
© copyright 2015 Xilinx

在本次实验中你学到了行为级建模中各种可用的结构。你还学到了阻塞和非阻塞操作以及时间约束的概念和需求。给实现（implementation）工具提供时间约束，生成的输出就能符合设计的时间需求。

© copyright 2015 Xilinx