Implementing Counters in Sequencer Devices AN050

INTRODUCTION

Some state machine applications require a state machine to wait for a number of clock pulses to occur before some decision point is reached. One common example of this is a state machine that needs to analyze only certain bits in a serial data stream. The state machine may have to wait for a number of serial data bits to transpire before pulsing a load signal or proceeding into states to actually check individual data bits for specific preamble or header information.

SEQUENCER ARCHITECTURE

State machine implementations using JK flip–flop based sequencer devices are generally very efficiently implemented because product terms (AND gates) are required only to force a transition from one state to the next. Product terms are not required to hold the sequencer in a state, as they are for D–type based devices. A JK based state machine can wait forever in one state for a specific parallel combination of input signals to happen, using only one product term to perform the comparison and force a jump to a new state. In addition, Philips sequencers have a PLA architecture meaning that both the AND array and the OR array have programmable connections. A single product term may be connected to the inputs of multiple state and/or output registers. This feature allows for efficient device resource utilization since any product term may be connected to any buried or output register. The product terms are not fixed in their usage to a specific register or output.

DESIGN METHODS

PLD software packages, such as Philips SNAP, provide for different methods of design entry. The easiest and usually best format for state machines is, of course, a state equation entry method. Figure 1 shows an example using state equations. For JK based sequencers, SNAP essentially translates each 'IF' statement into a product term in the device. An OR function in the input condition field of the 'IF' statement will cause an additional product term to be used.

A series of unconditional transitions to a new state may be found in some state machine designs where it is required to wait a certain number of clock cycles before performing a function. The example in Figure 1 shows a simple state machine that runs continuously through sixteen states and outputs a pulse on output 'OUT1' while in state 'F'. This state machine is not waiting for any inputs, other than the clock to occur. It is simply a counter.

COUNTER IMPLEMENTATION

For typical state machine implementations with conditional transitions between states, state equations produce efficient state machines. However for implementing counters, state equations may not produce the most efficient implementation. JK flip-flops have a feature whereby if both J and K are active, after a clock, the output will toggle or change state. This feature may be used to implement counters very efficiently. Combining the toggle feature of the flip-flops with a PLA devices ability to connect a single product term to multiple OR array inputs, produces an implementation where only one product term is needed for each bit in the counter. A four bit counter may be constructed using only four product terms!

The function described in Figure 1 is duplicated in Figure 2, except the Figure 2 design uses a counter described with Boolean equations. Only six product terms were used compared to the sixteen used for the design in Figure 1. Four product terms were used for the counter and two to control output pin OUT1. So, when a portion of a state machine design is required to unconditionally transition from one state to the next, consider implementing a counter using Boolean equations and merging it into the state machine.
The example in Figure 2 only used Boolean equations, no state equations. So, another example is shown in Figure 3. This example, using SNAP, illustrates the proper syntax for connecting the outputs of a counter to the inputs of a state machine. This example was complied for a PLUS405 device. The state machine will wait in each state until the counter reaches a specified value. It then transitions to the next state.

Complicating the design a bit more, Figure 4 shows another SNAP example. This is a listing of a design that, in addition to using the counter outputs as inputs to the state machines, connects two outputs of the state machine back to the counter. The outputs of the state machine (actually two of the state bits) can enable or disable counting, or reset the counter. In this example the state vectors were specially assigned such that state register S1 must be LOW for the counter to count. When state register S2 is HIGH, the counter will be reset. Instead of using state registers bits, additional outputs could have been defined and connected to the counter.

Figure 5 shows a counter that counts from 0 to 12 and then resets. This example may be easily modified to produce a counter that counts to any value.

SUMMARY

The toggle feature of JK flip-flops together with a product term sharing capability, found in most Philips sequencer devices, may be used to build counters using only one product term per counter bit. If a state machine design contains many unconditional transitions, it is possible to reduce the number of product terms required to implement the design by separating the design into a counter and state machine. The counter portion should be described using Boolean equations, when state equations are preferred for the state machines. The counter outputs may be used as inputs to the state machine and some state machines outputs or state bits may be used to enable or reset the counter.

```
@PINLIST
clk  i;
init i;
out1 o;

@GROUPS
@TRUTHTABLE
@LOGIC EQUATIONS

s[3..0].rst = /init; ”Use INIT function pin (19) to reset counter”
out1.rst    = /init; ”and to reset output pin.”

@INPUT VECTORS
@OUTPUT VECTORS
[out1]jkffr
o0 = 0b;
o1 = 1b;

@STATE VECTORS
[s3,s2,s1,s0]jkffr
st0 = 0000b;
st1 = 0001b;
st2 = 0010b;
st3 = 0011b;
st4 = 0100b;
st5 = 0101b;
st6 = 0110b;
st7 = 0111b;
st8 = 1000b;
st9 = 1001b;
sta = 1010b;
stb = 1011b;
stc = 1100b;
std = 1101b;
ste = 1110b;
stf = 1111b;

@TRANSITIONS

while [st0]
 if [] then [st1]
while [st1]
 if [] then [st2]
while [st2]
 if [] then [st3]
while [st3]
 if [] then [st4]
while [st4]
 if [] then [st5]
while [st5]
 if [] then [st6]
while [st6]
 if [] then [st7]
while [st7]
 if [] then [st8]
while [st8]
 if [] then [st9]
while [st9]
 if [] then [sta]
while [sta]
 if [] then [stb]
while [stb]
 if [] then [stc]
while [stc]
 if [] then [std]
while [std]
 if [] then [ste]
while [ste]
 if [] then [stf] with [o1]
while [stf]
 if [] then [st0] with [o0]
```

Figure 1. SNAP State Equations

```
@PINLIST
clk i;
init i;
out1 o;
@GROUPS
@TRUTHTABLE
@LOGIC EQUATIONS

”Simple four bit binary counter that”
”uses toggle feature of JK flip–flops.”
”Because of p–term sharing, only 4 p–terms”
”are needed to implement this counter.”

c0.j = 1;
c0.k = 1;
c1.j = c0;
c1.k = c0;
c2.j = c0 * c1;
c2.k = c0 * c1;
c3.j = c0 * c1 * c2;
c3.k = c0 * c1 * c2;

c[3..0].rst = /init; ”Use INIT function pin (19) to reset counter”
out1.rst    = /init; ” and to reset output pin”

”In this example the counter is free–running.”
”Out1 will be high when the count is 1111B and”
”will be forced low when the counter transistions”
”from 1111 to 0000 binary or reset by pin 19.”

out1.j = c3*c2*c1*/c0;
out1.k = c3*c2*c1* c0;

@INPUT VECTORS
@OUTPUT VECTORS
@STATE VECTORS
@TRANSITIONS
```

Figure 2. Counter Boolean Equations

```
@PINLIST
clk i;
init i;
out1 o;
out2 o;

@GROUPS
@TRUTHTABLE
@LOGIC EQUATIONS

"Simple four bit binary counter"

c0.j = 1;
c0.k = 1;
c1.j = c0;
c1.k = c0;
c2.j = c0 * c1;
c2.k = c0 * c1;
c3.j = c0 * c1 * c2;
c3.k = c0 * c1 * c2;

c[3..0].rst   = /init; "Use INIT function pin (19) to reset counter"
s[1..0].rst   = /init; " and state registers"
out[2..1].rst = /init; " and output pins"

@INPUT VECTORS
@OUTPUT VECTORS
[out1,out2]jkffr
 o0 = 0-b;
 o1 = 1-b;
 o2 = -0b;
 o3 = -1b;

@STATE VECTORS
[s1,s0]jkffr
st0 = 00b;
st1 = 01b;
st2 = 10b;
st3 = 11b;

@TRANSITIONS

                "In this example the counter outputs are used"
                "    as inputs to the state machine"

 while [st0]
  if [c3*c2*c1*/c0] then [st1] with [o1] "move to state 1 when counter goes"
                                         "from E hex to F hex"
 while [st1]
  if [] then [st2] with [o0]             "upon next clock go to state 2 and"
                                         "reset output"
 while [st2]
  if [/c3*c2*c1*/c0] then [st3] with [o3] "wait here until count = 6 hex"
                                          "then go to state 3 and set out2"
 while [st3]
  if [] then [st0] with [o2]              "goto state 0 and reset out2"
```

Figure 3. Counter Connected to State Machine

```
@PINLIST
clk  i;    out1 o;
in1  i;    out2 o;
init i;

@GROUPS
@TRUTHTABLE
@LOGIC EQUATIONS

"Four bit binary counter"
"controlled by state machine state register bits"

c0.j = /s1*/s2;                "Counter will be forced to 0000 upon"
c0.k = /s1;                    "clock and state bit s2 high."
c1.j = /s1 * c0 * /s2;         "Counter won't count unless state"
c1.k = /s1 * c0;               "register s1 is low. (won't count"
c2.j = /s1 * c0 * c1 * /s2;    "in state st1)"
c2.k = /s1 * c0 * c1;
c3.j = /s1 * c0 * c1 * c2 * /s2;
c3.k = /s1 * c0 * c1 * c2;

c[3..0].rst   = /init; "Use INIT function pin (19) to reset counter"
s[2..0].rst   = /init; " and state registers"
out[2..1].rst = /init; " and output pins"

@INPUT VECTORS
@OUTPUT VECTORS
[out1,out2]jkffr
 o0 = 0-b;
 o1 = 1-b;
 o2 = -0b;
 o3 = -1b;

@STATE VECTORS
[s2,s1,s0]jkffr
st0 = 000b;       "Note the special state assignments to"
st1 = -10b;       "simplify one state bit connections to"
st2 = -01b;       "the counter."
st3 = 111b;

@TRANSITIONS

               "In this example the counter outputs are used"
               "as inputs to the state machine and some of the"
               "state register bits S2 and S1 control the operation"
               "of the counter."

 while [st0]
  if [/c3*c2*/c1*c0] then [st1] with [o1] "move to state 1 when counter goes"
                                          "from 5 hex to 6 hex"
 while [st1]
  if [in1] then [st2] with [o0]    "when input 'in1 = high' go to state 2 but"
                                   "hold counter at 6 while waiting for in1"
 while [st2]
  if [c3*/c2*/c1*/c0] then [st3] with [o3] "wait here until count = 8 hex"
                                           "then go to state 3 and set out2"
 while [st3]
  if [] then [st0] with [o2]               "goto state 0 and reset out2"
                                           "and counter"
```

Figure 4. Counter Enable and Reset Functions Controlled

```
@PINLIST
clk i;
ignd i;
init i;
out1 o;

@GROUPS
@TRUTHTABLE
@LOGIC EQUATIONS

"Four bit binary counter"
"modified to count from"
"0 to 11 and then reset."

c0.j = nor;
c0.k = nor;
c1.j = nor * c0;
c1.k = nor * c0;
c2.j = nor * c0 * c1;
c2.k = (nor * c0 * c1) + count12;
c3.j = nor * c0 * c1 * c2;
c3.k = (nor * c0 * c1 * c2) + count12;

count12 = c3*c2*/c1*/c0;
nor = /(count12+ignd);

"When count=11, then output NOR is LOW, disabling the product terms"
"that cause the counter to count. Another product term (count12)"
"connects to the registers of the counter that are HIGH K-inputs,"
"forcing it to all zeros upon the next clock. These connections may"
"be modified to alter the upper count limit."

c[3..0].rst = /init; "Use INIT function pin (19) to reset counter"
out1.rst    = /init; " and to reset output pin"

"In this example the counter is free-running."
"Out1 will be high when the count is 1100B and"
"will be forced low when the counter transistions"
"from 1100 to 0000 binary or reset by pin 19."

out1.j = c3*/c2*c1*c0;
out1.k = c3*c2*/c1*/c0;

"For SNAP 1.90 to implement this design in a minimum number of product
 terms, two passes through the merger are necessary. First, generate a
 netlist normally - running NETCONV and MERGER. Then, highlight equations
 in the MERGER box to extract the equations from the netlist. Run the
 extracted equations through the minimizer (EQNGEN). Run through NETCONV
 (Minimized) and MERGER again to produce a minimized netlist. The design
 may then be compiled for the device."
@INPUT VECTORS
@OUTPUT VECTORS
@STATE VECTORS
@TRANSITIONS
```

Figure 5. Modulo-n Counter