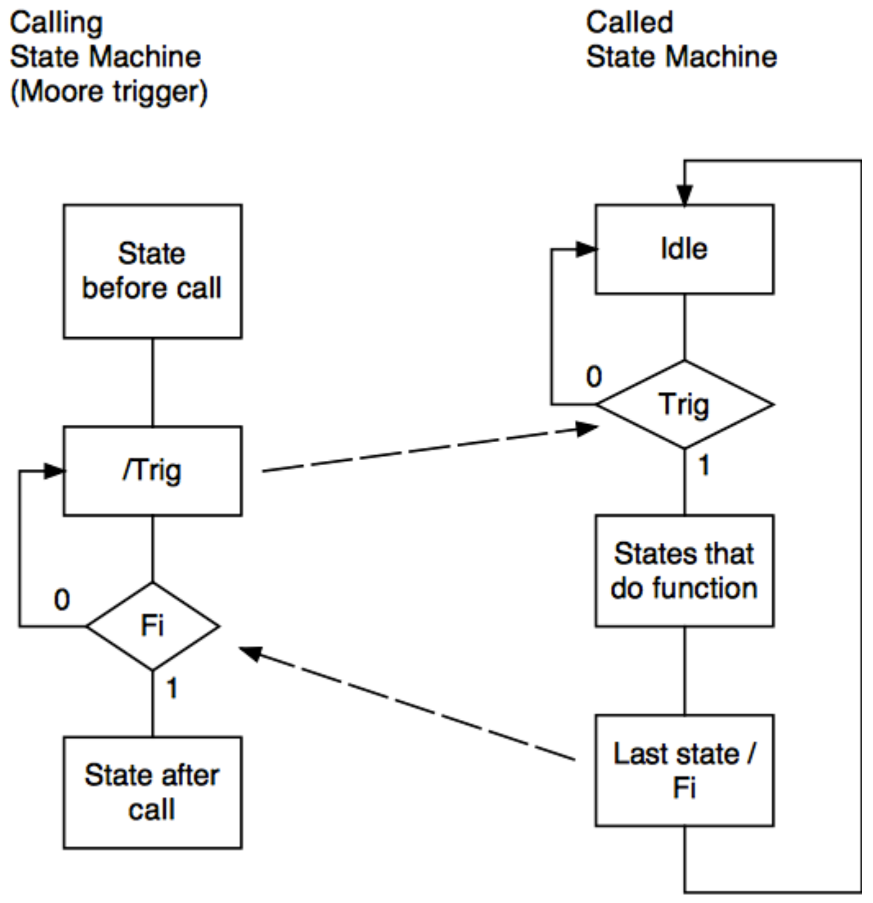
EE432 Advanced Digital Design with HDL
Spring 2013
Week 3

Lecture Topic, April 16

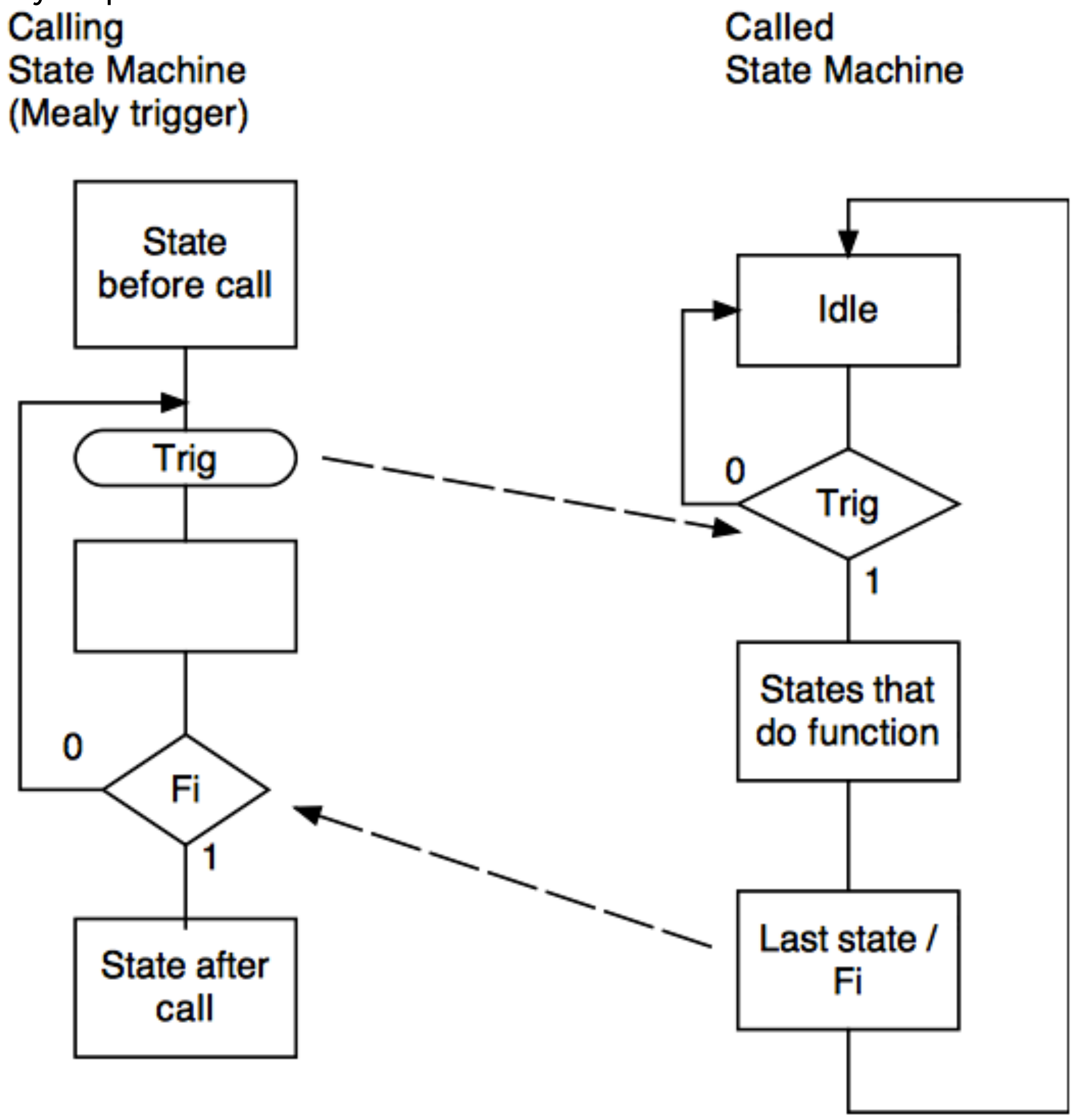
We will start with a general review of using state machines to control logic circuits. Then we will look at handling multiple state machines and alternative state machine implementations.

Reasons for more than one state machine in system:
- Single state machine too complicated to implement
- Division of labor – tasks run in parallel.
- Multiple clock domains

Communication between cooperating state machines uses a handshake with start signal, *Trig*, to "called" SM and finished signal, *Fi*, going back to calling SM. If clock source for both is the same then we don't need the "full handshake" used in asynchronous systems or in cases where we don't want to wait for the other state machine to finish.

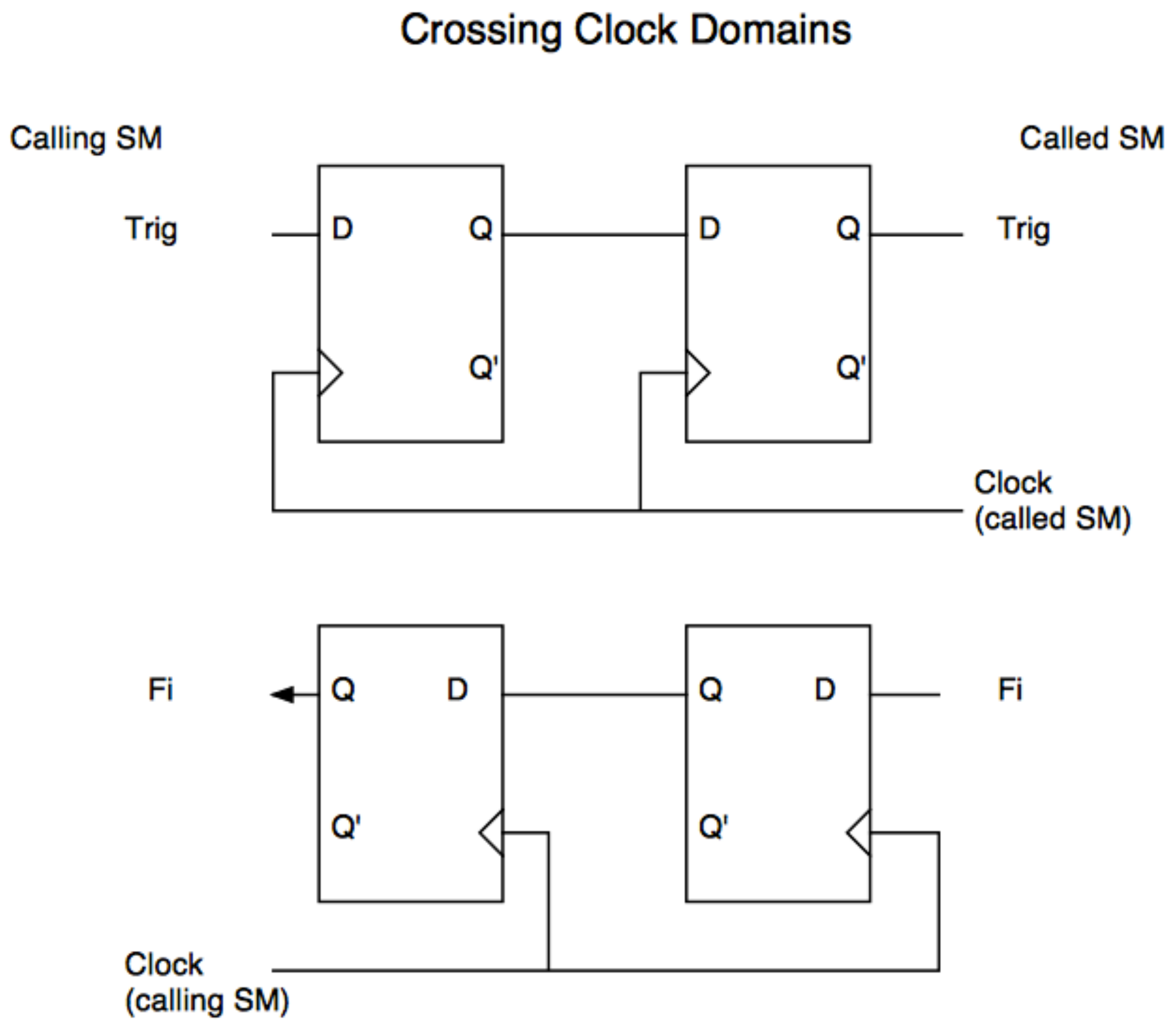


If we are careful, we can use a Mealy output to trigger . It is very important in this case the two state machines be on the same clock domain because glitches in the Mealy output could cause false state transitions.

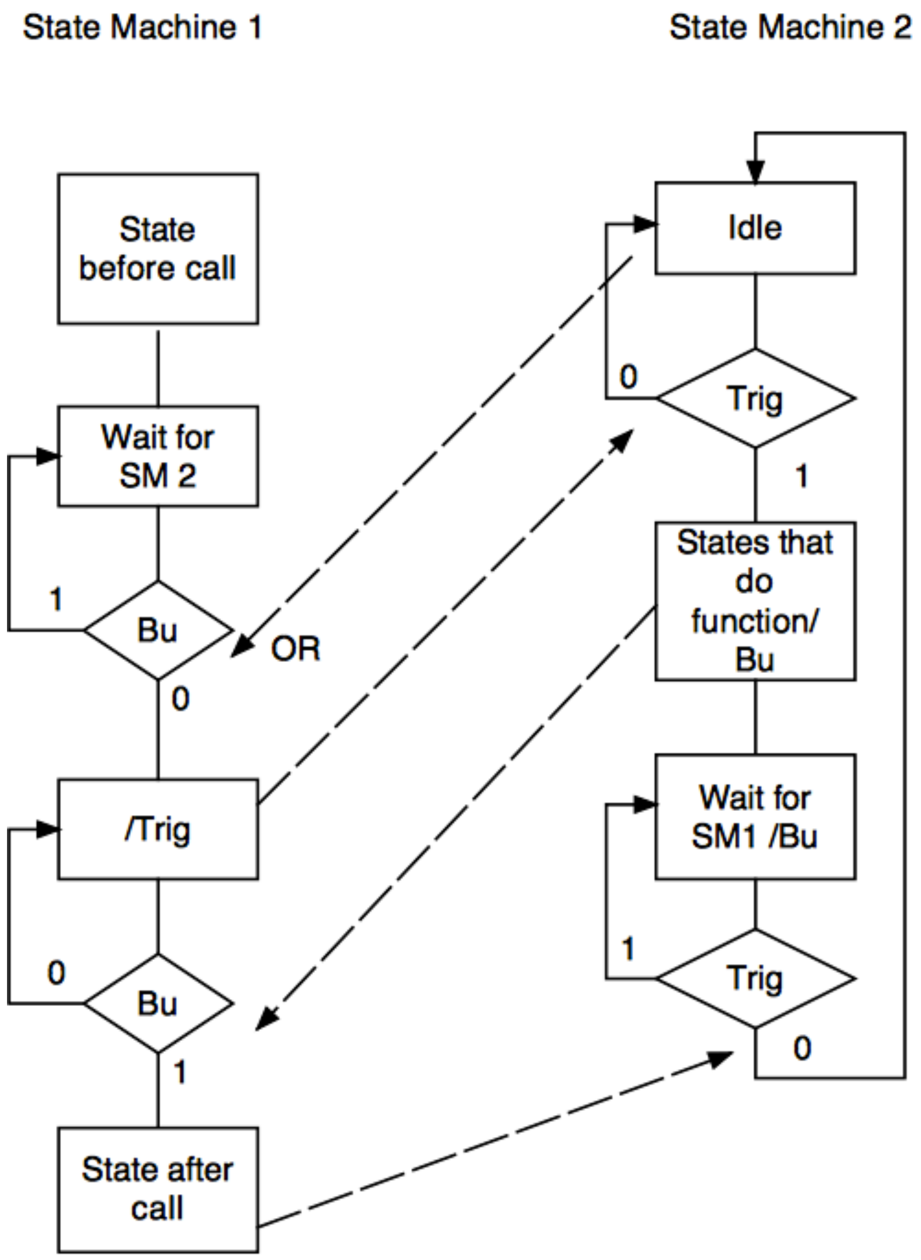


Be careful that data for subroutine is available at time it is called.
Be sure that when the called machine gets back to the idle state that Trig is no longer being asserted '1'.

If the two state machines are in separate clock domains differing primarily in phase (if different frequencies, they must be close) we need to synchronize the *Trig* and *Fi* signals to the receiving clock domains.



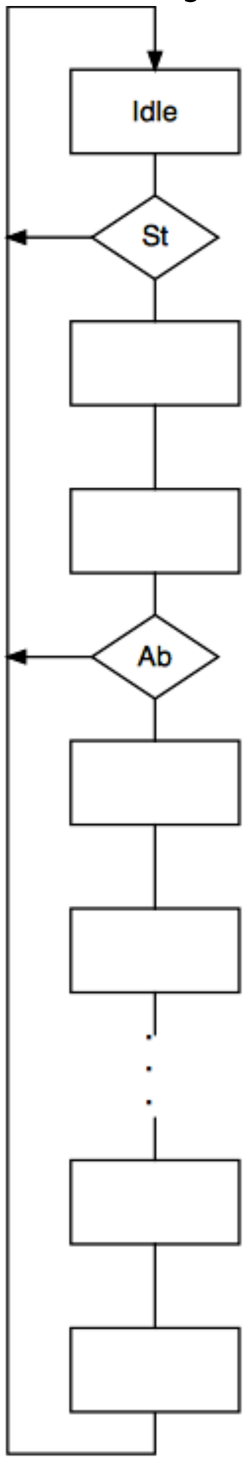
The full handshake version is for two state machines where one is not subordinate to the other. They can be in completely different clock domains (providing the synchronization shown in the previous page is used!).

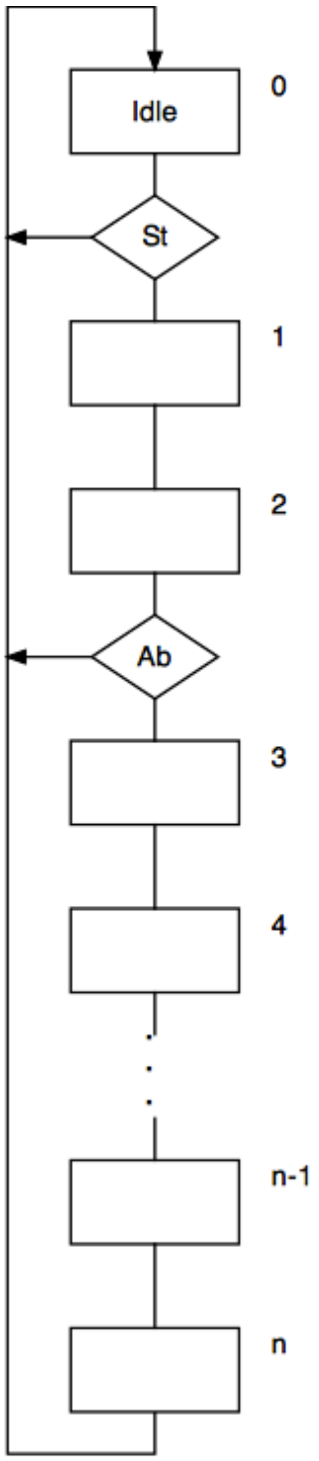


State machine 1 waits for SM2 to be not busy (bu=0) , then asserts Trig until SM2 asserts Bu. Note that “busy” is the same signal as not “finished” of the preceding example. If SM1 needs some result from SM2, it must wait again for bu=0. SM2 has an analogous handshake. I've drawn it like it is a subordinate state machine, but that isn't necessary.

Alternative SM design

When the state machine looks like the one on the left – basically one long sequence with only alternatives being back to the idle state – then the design is best implemented using a counter and not the approach we have done in the past. The SM on the right shows numbered states, the counter values.



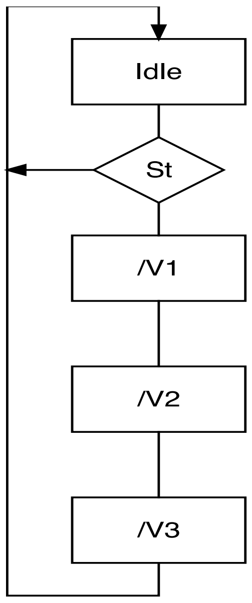


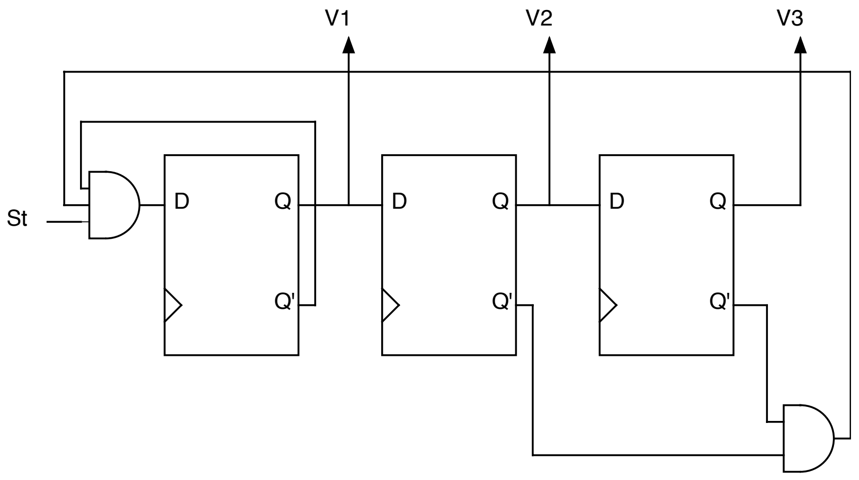
This counter is easily implemented in VHDL as follows assuming n = 100:

```
Signal state : integer range 0 to 100 := 0;
-- we could use std_logic_vector here as well,
-- but that would require know the number of bits.


Process (clk) begin
     If rising_edge(clk) then
          If state = 100
or (state = 2 and ab = '1')
               or (state = 0 and St = '0') then
-- reset the counter
     state <= 0;
else
     -- increment the counter
     state <= state + 1;
end if;
     end if;
end process;
```

For even simpler state machines we can implement as a shift register that is basically a one-hot state machine. In a on-hot design, each flip-flop represents a single state. Recognize the one shot from EE331? This keeps the state machine "one hot" because it is important that the SM not be re-triggered before it finishes, which would allow two flip-flops to be Q=1 and the state machine would be in two states!





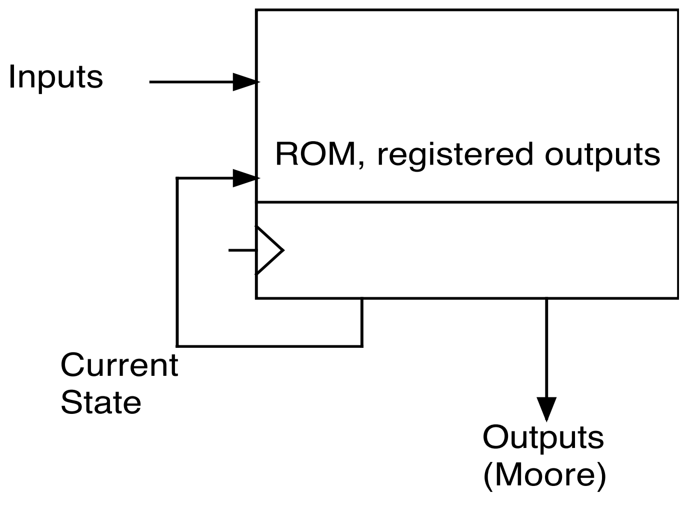
VHDL implementation would be:

```
Process (clock) begin
     If rising_edge(clock) then
          V1 <= St and not (V1 or V2 or V3);
          V2 <= V1;
          V3 <= V2;
     End if;
End process;
```

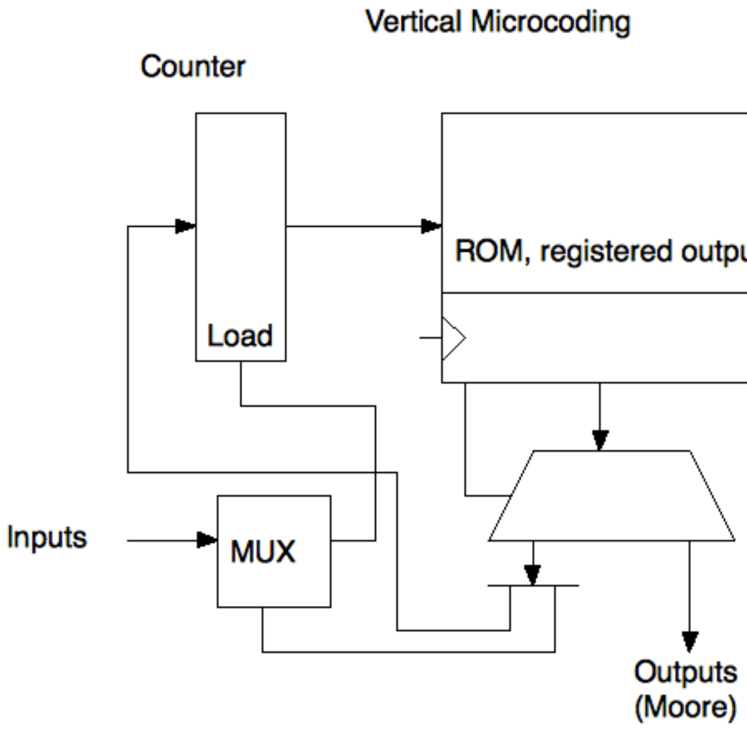
Microprogramming

Another solution to complex state machines is microprogramming. First, a quick look at traditional SM implementation using a ROM. ROM address lines are driven from SM inputs and "current state" . The ROM data lines are registered. Some represent "next state", and others the Moore outputs (because they are registered). This design is simple but generally inefficient because of the large number of bits necessary. Using a PAL or PLA simplifies the design in most cases by providing the equivalent of a sparse ROM.

Traditional ROM SM Design



Microprogramming is basically a ROM based state machine design where each state is similar to a processor instruction. Vertical organization has small instruction word sizes and executes sequential locations by default while horizontal organization specifies the next location in each instruction. In the vertical organization, the data lines used for the conditional next address and mux are typically shared with the output data lines, with an additional control bit to determine if the instruction is a conditional branch or other function. Outputs can be partially encoded to save bits. For instance, if there are three output signals that are mutually exclusive, then can be encoded in a single two bit field, saving a bit.



In the horizontal version, each instruction can specify the next instruction for any selected input being 1 or being 0. All outputs are available on each instruction. Variations cut down on the width of the memory by having only one next address field and having the inputs controlling a single address bit (so addresses are in pairs when a branch is used. Again, outputs can also be partially encoded (for those that are mutually exclusive) to save bits.

Homework Assignment #3, Due April 23

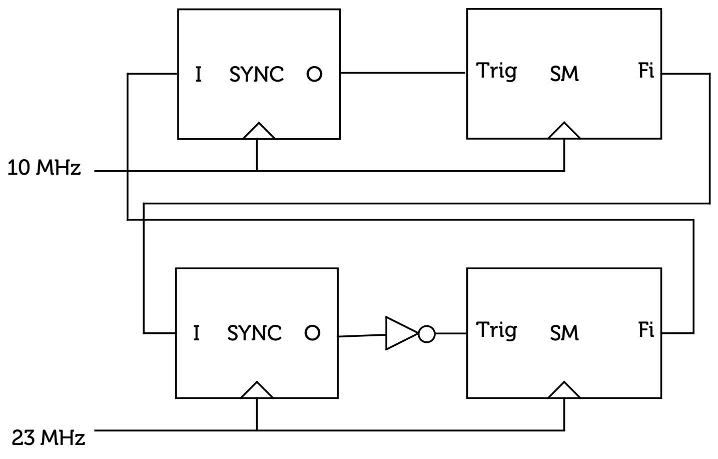
Design a Moore state machine that does nothing but handshaking. It should implement a full handshake. Place the state machine in a module. The ports should be: *clk* and *trig* as inputs and *fi* as an output. Of course there is also a clock input. The state table is:

Current State	Next State		Output, fi
	trig=0	trig=1	
S0	S0	S1	0
S1	S1	S2	0
S2	S3	S3	0
S3	S4	S4	0
S4	S0	S4	1

Make a top level with an two instances of the state machines. Connect the fi outputof the first state machine "through an inverter" to the trig input of the second machine and the fi output of the second state machine to the trig input of the first state machine. With the state machines connected to the same clock source, the state machines will alternate running. Submit the VHDL for this part of the assignment.

Now split the clock in two, one for each state machine, and add clock synchronizers where the signals cross the clock domains. Run one clock at 2.3 times the frequency of the other. Submit the simulation output (showing the trig and fi signals for the two state machines) and the VHDL for this part of the assignment.

Second Part of Assignment



Current Project Assignment
Keep up the good work, and don't forget the journal!