



AN HONORS UNIVERSITY IN MARYLAND

Department of Computer Science and Electrical Engineering

CMPE 415

Verilog Case-Statement Based State Machines II

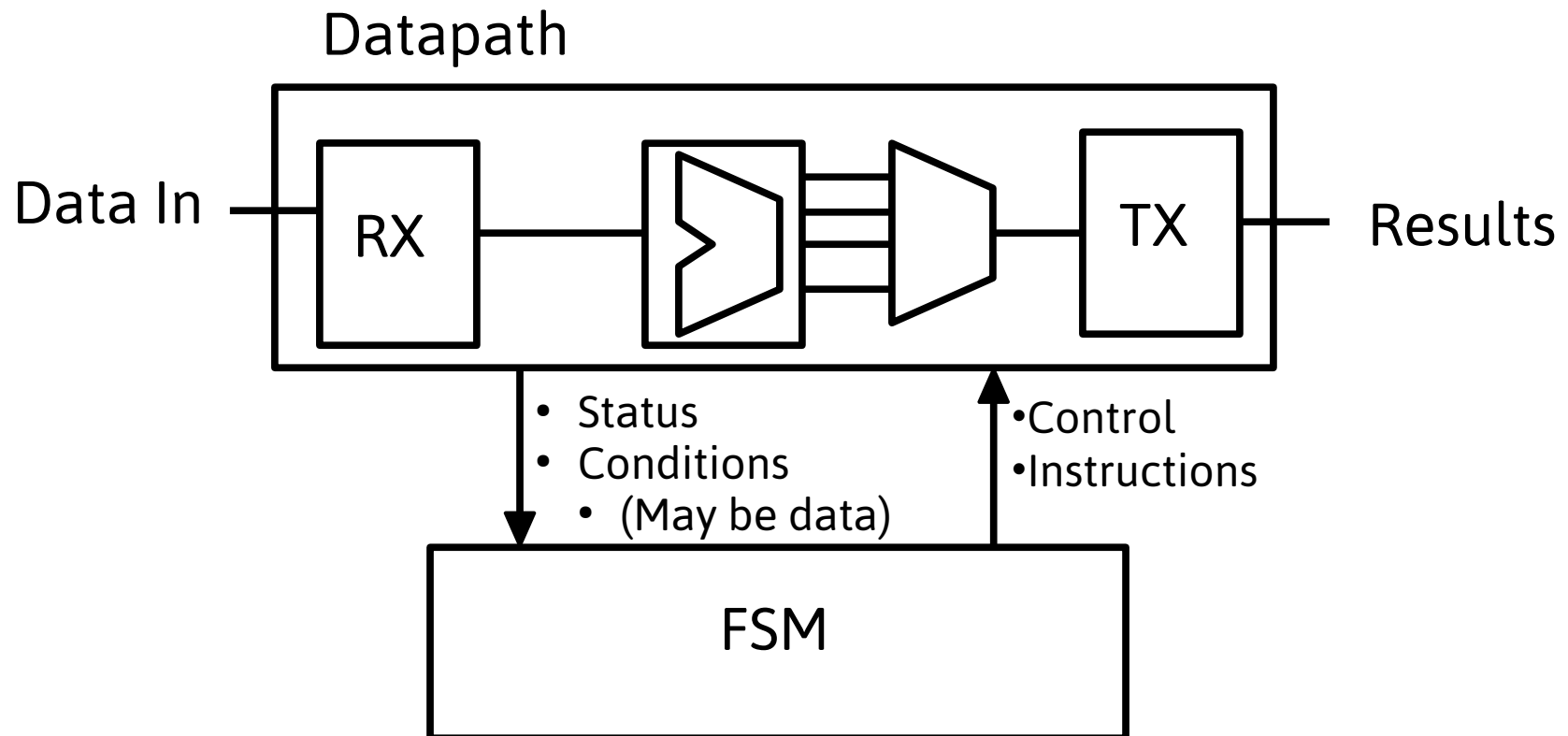
Prof. Ryan Robucci

Finite State Machine (FSM)

- Characterized by
 - A set of states
 - A set of inputs and outputs
 - A state transition function
 - An output function
- Hardware Implementation:
 - Current State held in a register
 - Any additional status information held in status register
 - Next-State and State Control Logic Determines next state and control signals based on registers
 - Datapath implements operations on data under control of state control logic

Finite State Machine with Datapath

- A very common framework being described and implemented is a Finite State Machine with a Datapath: a data path controlled by signals from a FSM

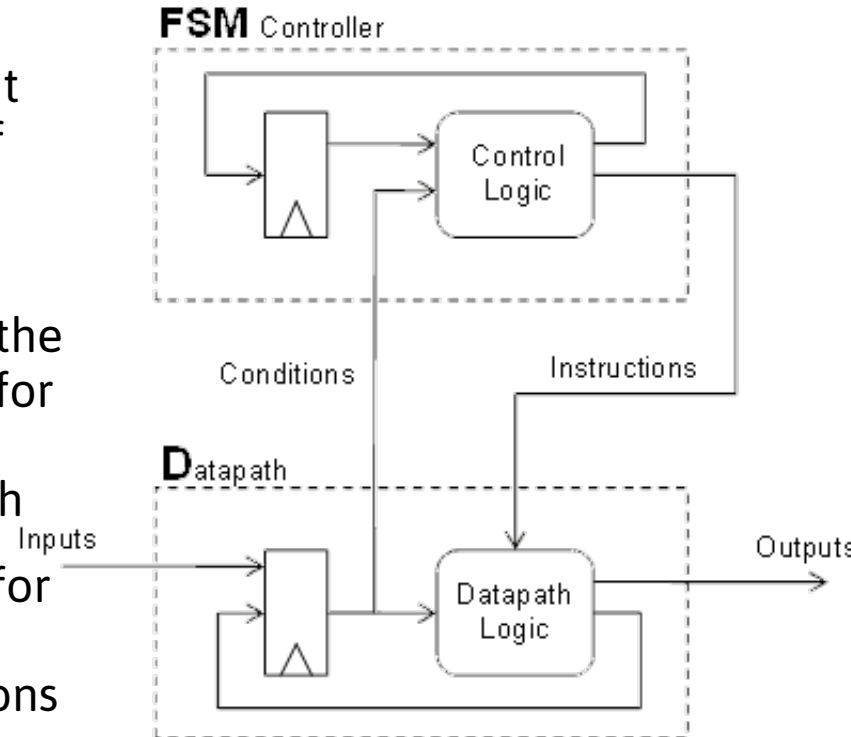


Control and Datapath Partitioning

- Datapath operations can be encoded in the state machine description or can be build separately.
 - If pre-built modules are used, as is common, the datapath is necessarily a separate description.
 - Often the datapath represents the algorithm calculations, separating the datapath code makes the code more readable
 - Coding datapath separately can allow more explicit influence over and insight into the resources used for computation while separating details of control code
 - Can sometimes think of datapath as implementing instructions for the controller to invoke/call
 - Separating the datapath code can allow better reuse under a different scheduling of operations (implementing a different algorithm)
 - In general, designing the datapath first and then the control is a good strategy

Temporal Processes of statemachine

- Just after clock edge, state variables are updated. For controller this means that a state-transition is complete and state registers holds the new current state. For the data path, this means that the register variables are updated as a result of assigning (algebraic, boolean, logical, etc...) expressions to them.
- Controller FSM Combines the control state and the data-path state to evaluate the new next state for the, at the same time it will also select what instructions should be executed by the datapath
- The datapath FSM will evaluate the next-state for the state variables in the datapath, using the updated datapath state as well as the instructions received from the control FSM
- Just before the next clock edge, both the control FSM and the datapath FSM have evaluated and prepared the next-state value for the control state as well as the datapath state



Controller FSM vs Datapath

Datapath

- Regular Structure
- Often easy to describe with expressions or structurally build with blocks
- Registers represent algorithmic states (intermediate or partial results)

Controller FSM

- Irregular Structure
- Very well suited for FSM-style description, difficult to describe with (boolean) expression-based
- Registers represent the familiar sense of the state of the system

Divide and Conquer

- Datapath operations can be encoded within the same procedural code as the state machine description or can be built separately.
- First consider partitioning.
 - Identify elements in the datapath from experience with traditional digital systems (e.g. communications modules, arithmetic modules, multiplexor's and demultiplexors, registers and multi-word buffers, FIFOs, IP Cores etc...).
 - Identify the control signals required and the status/condition information required to make decisions on the control.

Computational Statemachines

- However, encoding datapath operations WITHIN in the statemachine description with the controller instead of coding them in a separate block is sometimes better.
- It is common to see “computational statemachines” described with control and computation embedded in the same procedural block. These are modules which perform complex computations over multiple cycles and require internal registers/memory.
- We’ll first focus on control state machines first, with an emphasis on timing and external status and control signals then discuss computational statemachines

Timing

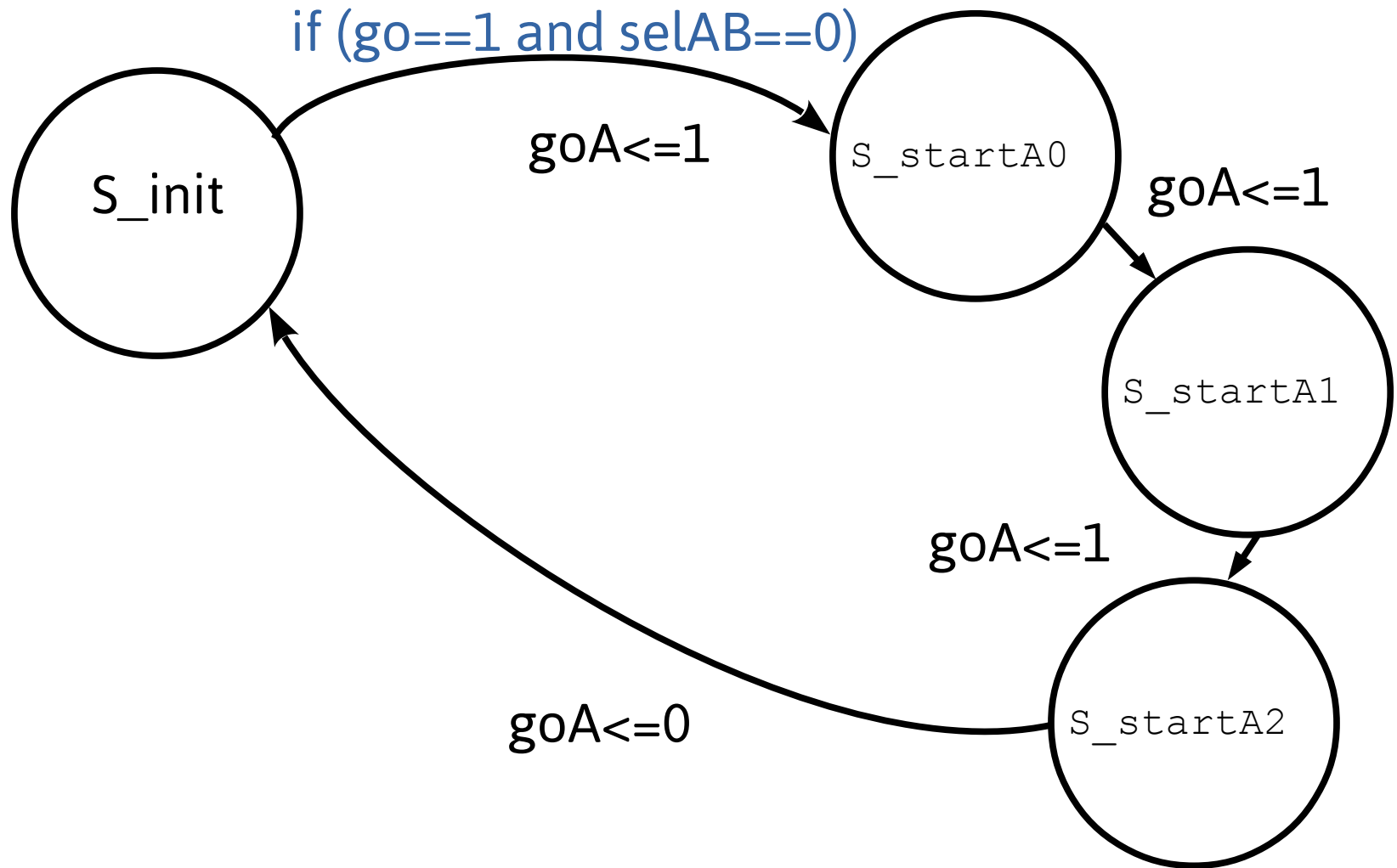
- It is common to require conditional and unconditional waiting
 - Fixed waiting - wait a predetermined number of cycles
 - Minimum wait – wait for some time and then wait for a condition based on external input to be satisfied. This is useful when interfacing with external “slow” entities that need time after being signal to send back a response.

Implementation of Waits

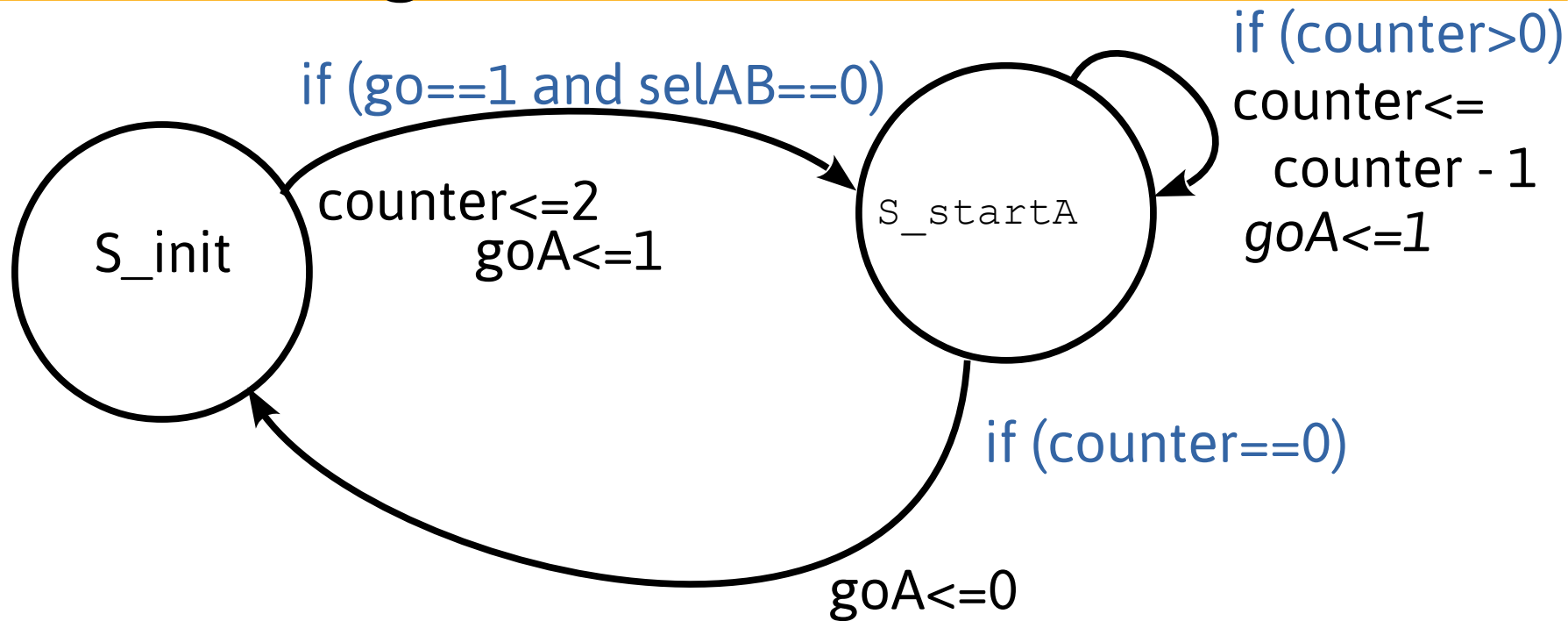
Some Options:

- Add “top” level wait states to state machine in each place needed
- Create a single embedded wait state to jump to and return to from multiple states using a “return” register (embedded states)
- Use a counter
 - a) Use external counter or implement one with another state machine and interface to it
 - b) embed something like a counter in the coding of the state machine thus creating substates using the counter as an extended state register.

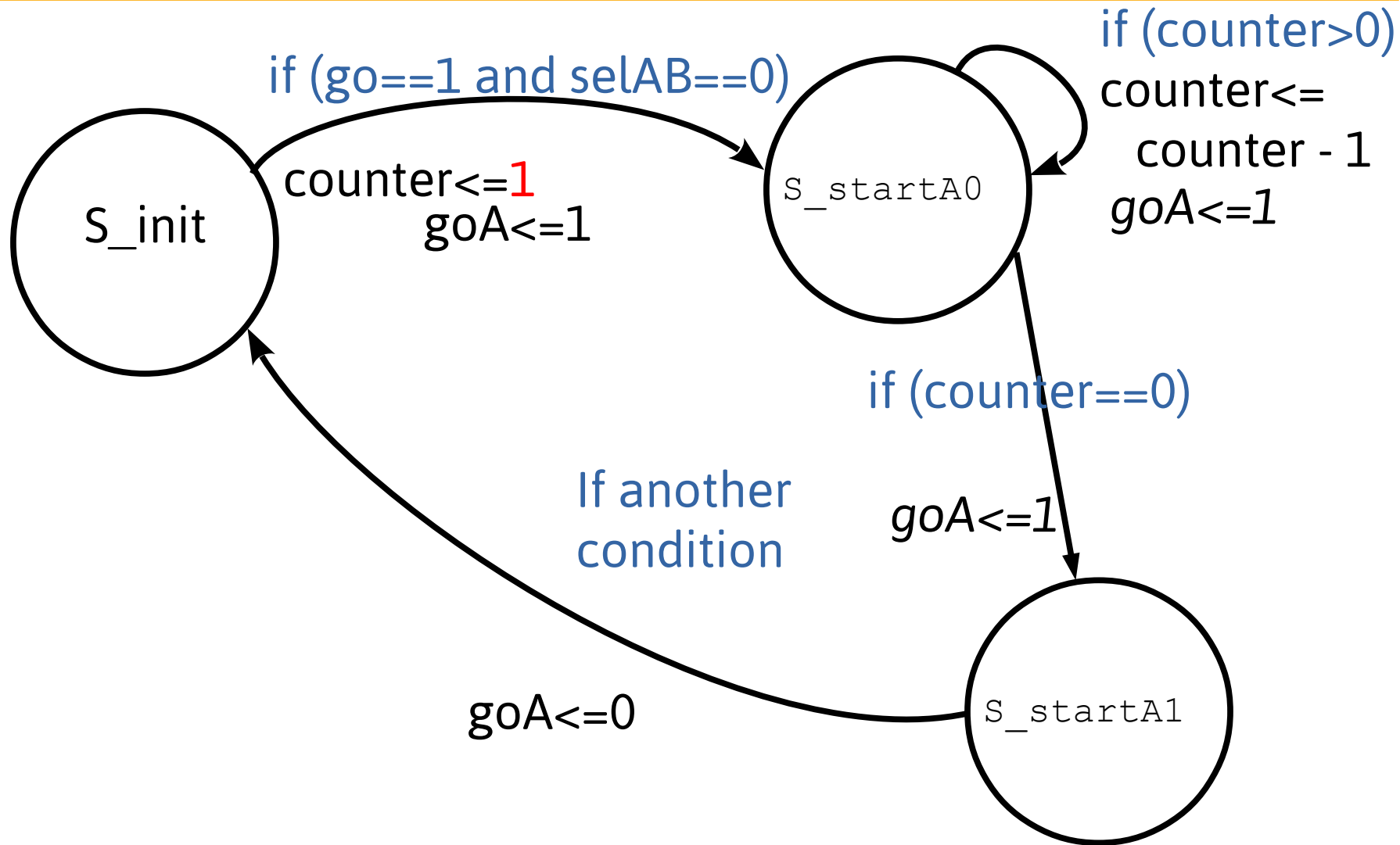
Unconditional Delay Using Extra States



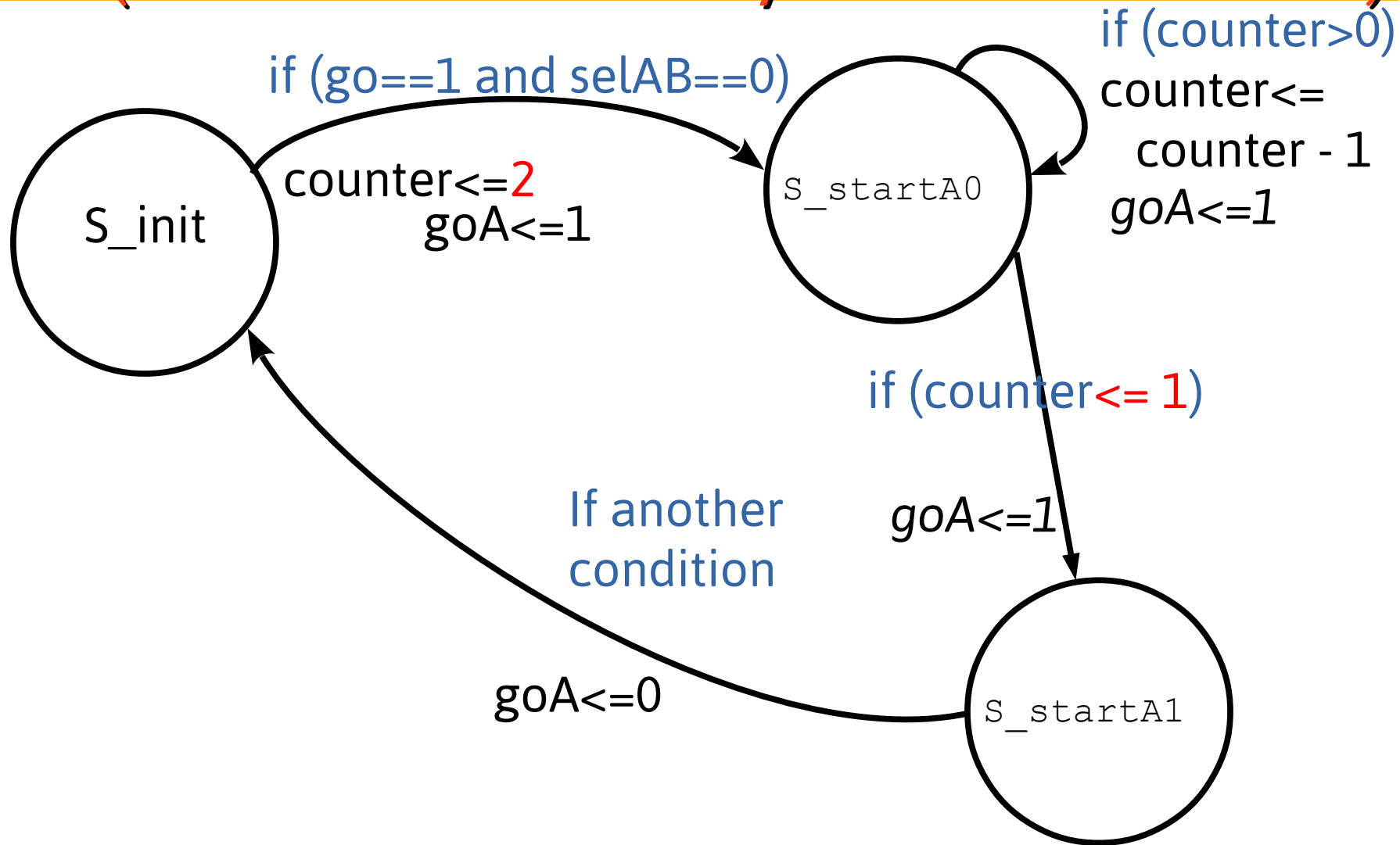
Unconditional Delay Using Extended State Register Variable : Counter



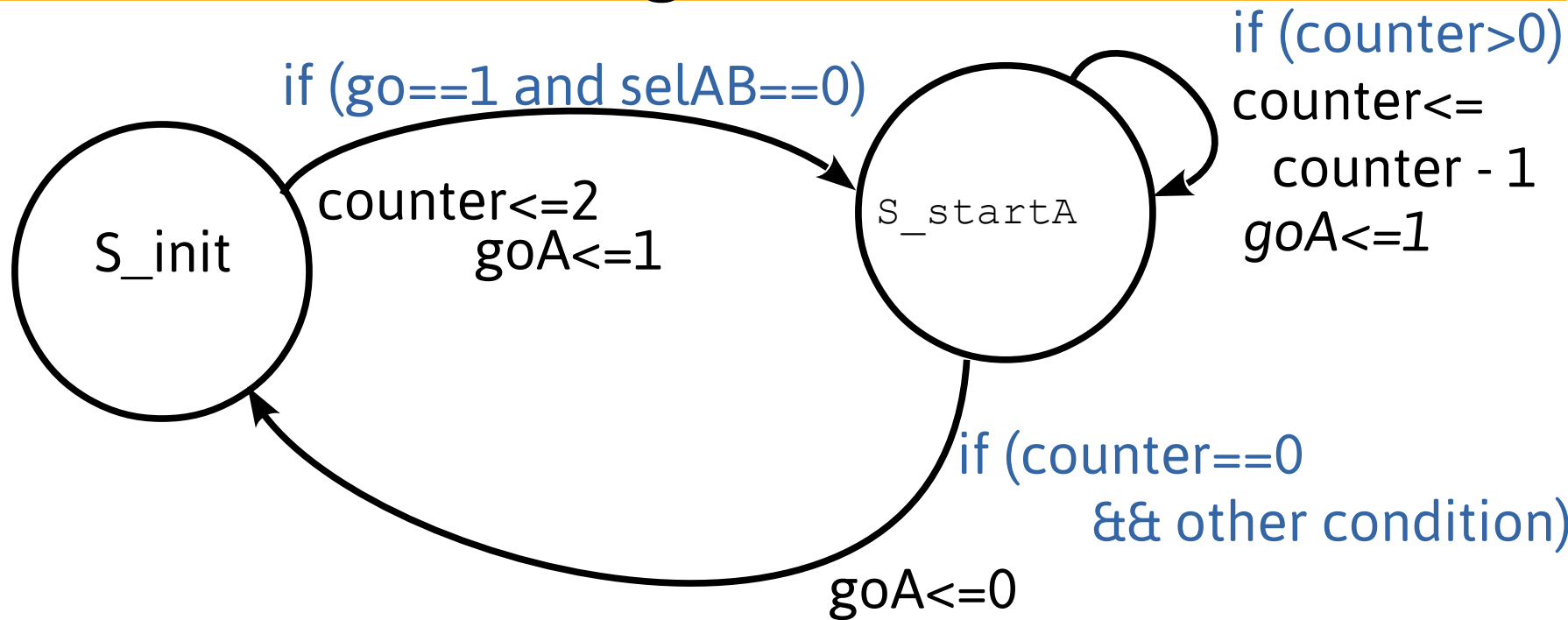
Minimal Pause: Delay + Conditional Exit Using An Extra State (unconditional delay + conditional exit)



Delay + Conditional Exit Using An Extra State (unconditional delay + conditional exit)



Minimal Pause: Delay + Conditional Exit Using Extended State Register Variable Counter

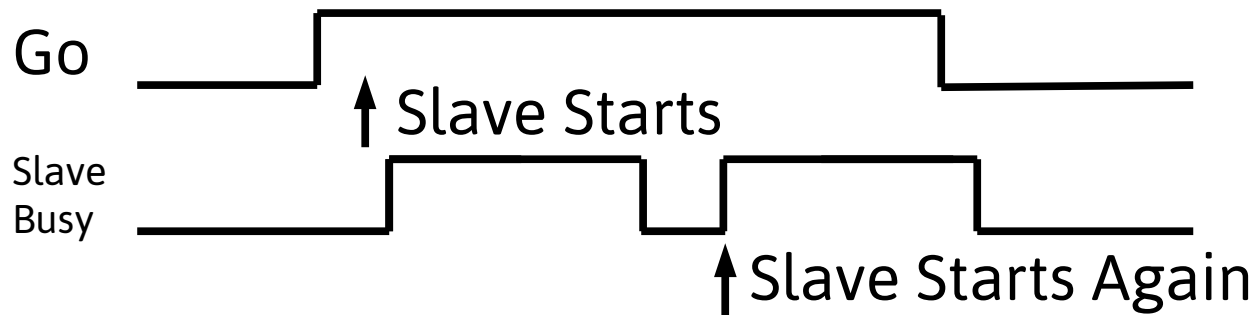


Thought Question

- Better to grow state register or use a separate counter variable?
 - There is no universal answer

Detecting Change

- A slave device commonly starts processes based on a start signal from a master. A slave process may be fast or slow compared to a master
- Consider if master is slow, and go is left high too long.



To alter the behavior, can instead look for a change in the signal
Saving the previous input value:

```
always @ (posedge clk) go_prev<=go;
```

Elsewhere use this as a condition

```
(start==1 && start_prev==0)
```

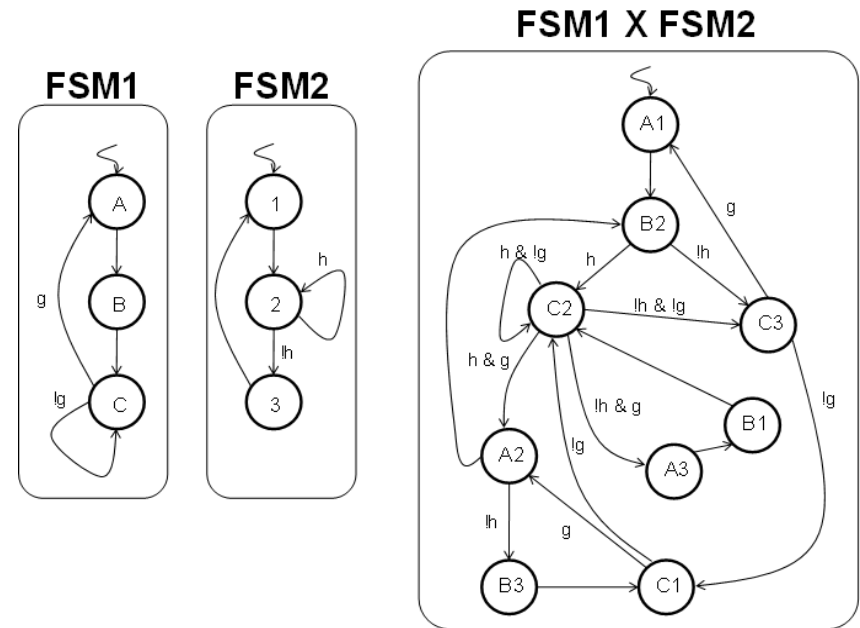
Alternatively, what if we want the system to cycle again immediately without startA going low if start is left high?

Limitations of FSM

- FSM typically lack any hierarchy. There is no way to connect details of state-machines leading to state explosion. Otherwise, a hierarchy of states is quite useful for organizing an algorithm.

State Machine State Explosion

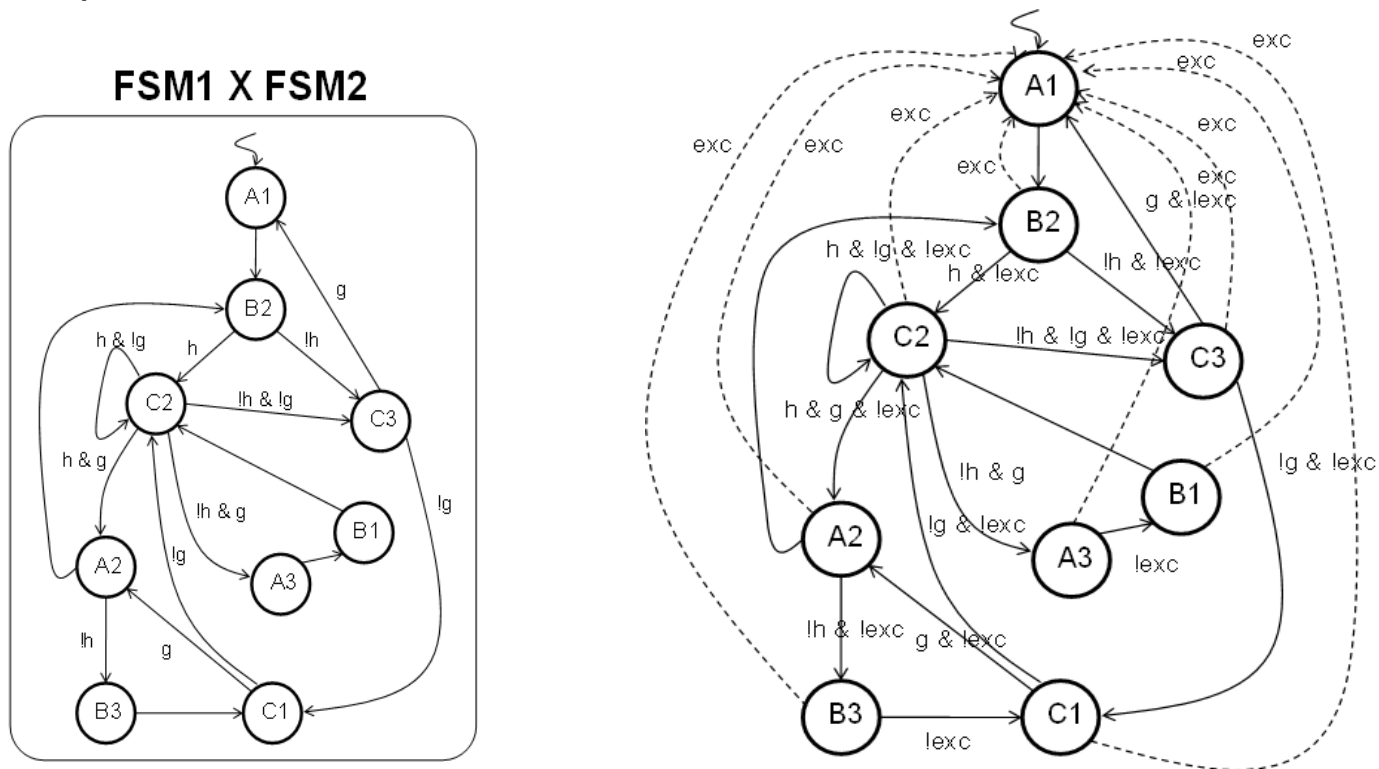
- Consider two state machines. Maybe one captures user input like desired temperature, it has states, inputs and outputs appropriate to perform that task. Perhaps the other controls a heating element based on measured and desired temperature. Separated, they may be fairly simple, but what happens when they are described as a single, flat state machine?
- A multiplicative effect. Two three-state FSMs became one nine-state FSM



- This motivates partitioning into multiple state machines in hardware design

Global Exceptions

- Now, see what happens if a single new condition, a universal exception must be added.
- A dramatic result from only one new condition. Readability and perhaps feasibility of mentally managing the FSM is severely impacted.



Meanwhile in software...

- A single if statement can be added, taking advantage of the hierarchy of logic embedded in code.

```
switch(current_state) {  
  case A: ...  
  case B: ...  
  case C: ...  
  case D: ...  
}  
  
if (exception) {  
  do this  
} else {  
  switch(current_state) {  
    case A: ...  
    case B:  
    case C:  
    case D:  
  }  
}
```

Delay/Pause/Waiting

- Think about the delay examples given earlier using a counter. The counter was implementing one form of hierarchical states.
- Consider waiting for an acknowledgment signal for 2^{20} clock cycles. This would require ~1Millon states with the condition to move to the next state or proceed to end state if acknowledge was received.

Multi-Cycle Computations as FSMs

- In the following slides, we will consider examples of **computational** FSMs. These are used to describe and synthesize multi-cycle computations. The number of state transitions defined are typically less than with a complex **controller** FSM.
- A more regular forward orderly progression through the states is typical, and the designer's thought processes may be focused on the computation being performed in each state and the resulting partial results (in registers), as opposed to the output during the state.
 - Thus, it becomes more reasonable to use a single edge-triggered always block in the design. The fewer number of transition decisions (branches) makes the concern of code bloat negligible.

FSM Optimizations

As we are designing Multi-Cycle computations, we may consider two optimizations:

Rescheduling – moving internal operations to other states

Resource sharing – utilizing same component in multiple states

Module Header

Head

```
module FSM_opt(  
    output reg [7:0] f,  
    input clk,  
    input wire [7:0] i,  
    input wire [7:0] j,  
    input wire [7:0] k,  
    input rst  
);  
  
reg [7:0] CS;  
reg [7:0] i_int, j_int, k_int;  
  
localparam S_0 = 8'b00000000;  
localparam S_1 = 8'b00000001;  
localparam S_2 = 8'b00000010;
```

Module Body

Body

```
always @ (posedge clk) begin
    if (rst) begin
        CS<=S_0;
        f<=0;
    end else begin
        case (CS)
            S_0: begin
                i_int<=i;
                j_int<=j;
                k_int<=k;
                CS<=S_1;    end
            S_1: begin
                CS<=S_2;    end
            S_2: begin
                m=i_int*j_int;
                f<=m*k_int;
                CS<=S_0;    end
        endcase
    end
end
endmodule
```

Multi-cycle computation:

Note, the single-block style here was used since the state transition sequence is straightforward. Also, the data path is combined with the controller.

We are not worried about “code bloat” from having to code each output on every transition.

Instead of concentrating on
timing coincidence of state and outputs

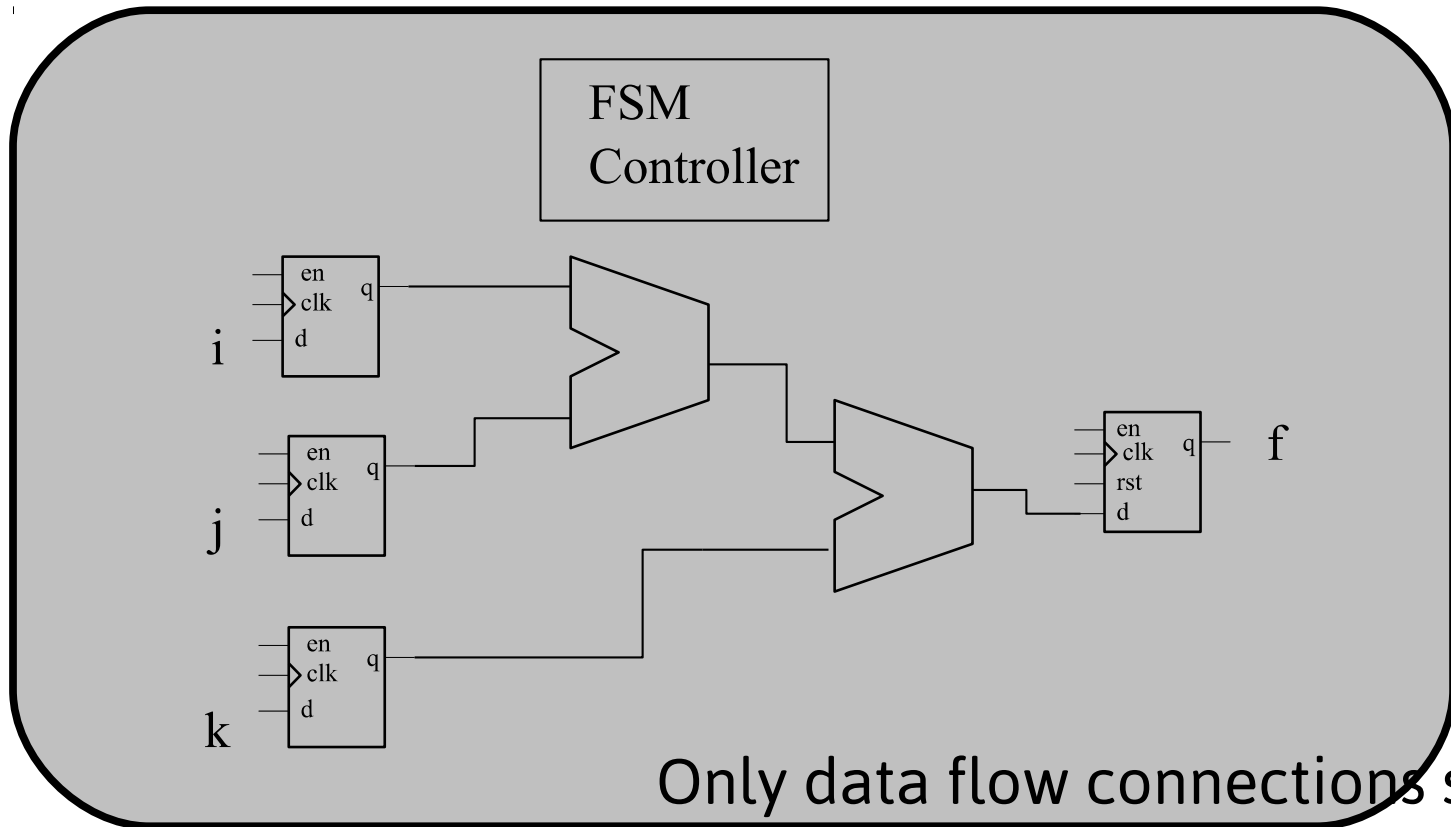
vs

coding coincidence of state and outputs,

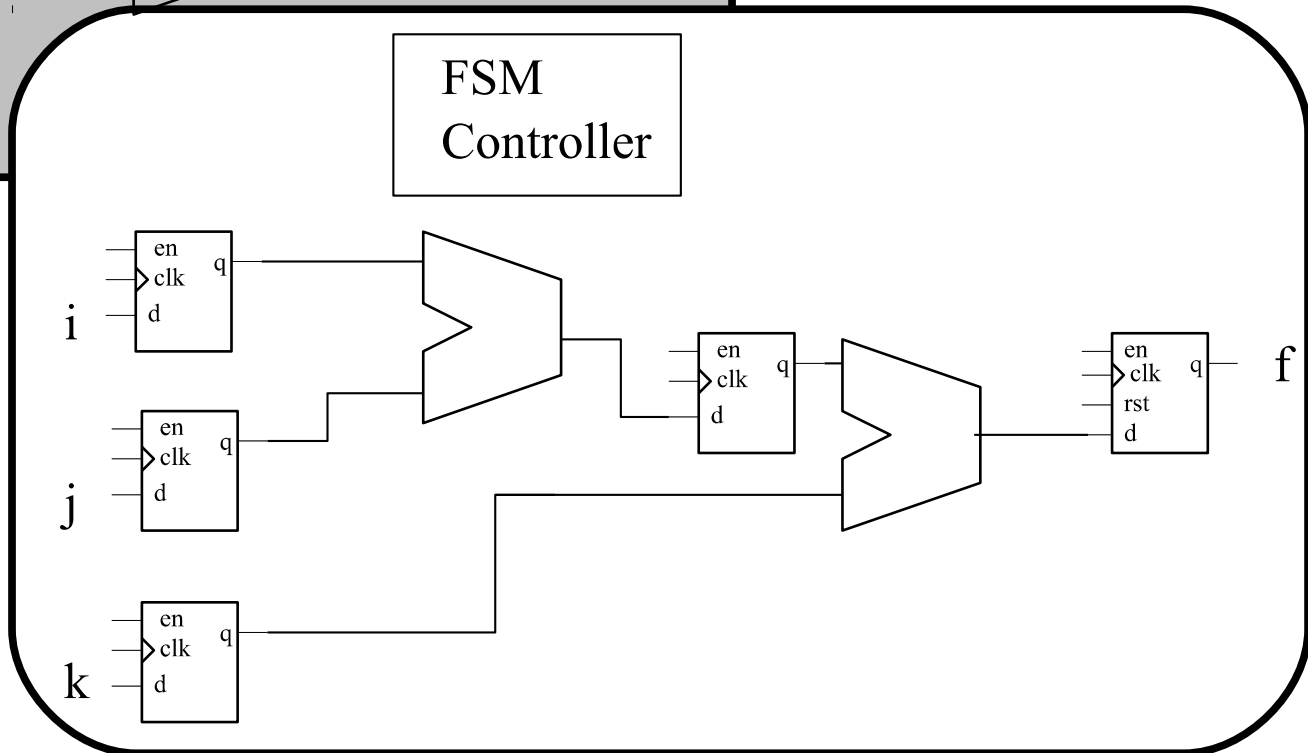
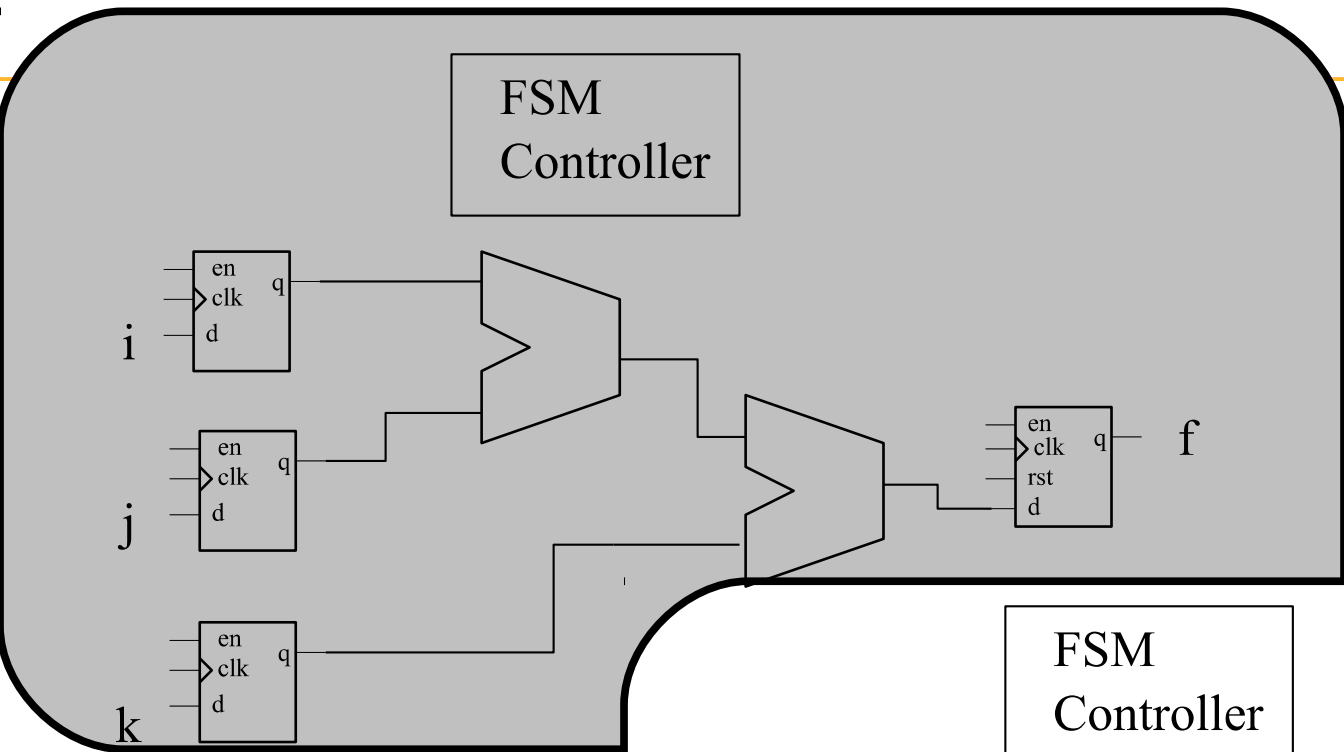
we are focused on the

coincidence of states and computations. In this single block style the computation performed and resources required for each each are clear, though the corresponding output of each computation is seen and can be used on the cycle following the corresponding state indicated in the state register.

RTL suggested from Code

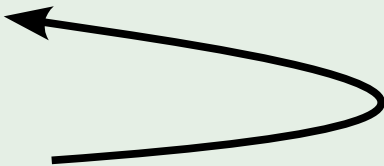


Alternative RTL with faster operation



Rescheduling

```
always @ (posedge clk) begin
    if (rst) begin
        CS<=S_0;
        f<=0;
    end else begin
        case (CS)
            S_0: begin
                i_int<=i; j_int<=j; k_int<=k;
                CS<=S_1; end
            S_1: begin
                m<=i_int*j_int;
                CS<=S_2; end
            S_2: begin
                //m=i_int*j_int;
                f<=m*k_int;
                CS<=S_0; end
        endcase
    end
end
endmodule
```



Rescheduling this multiply allows for faster clock speeds (we assume two clock cycles were required at the system level). Some synthesizers may do similar types of rescheduling for you.

Computation Module

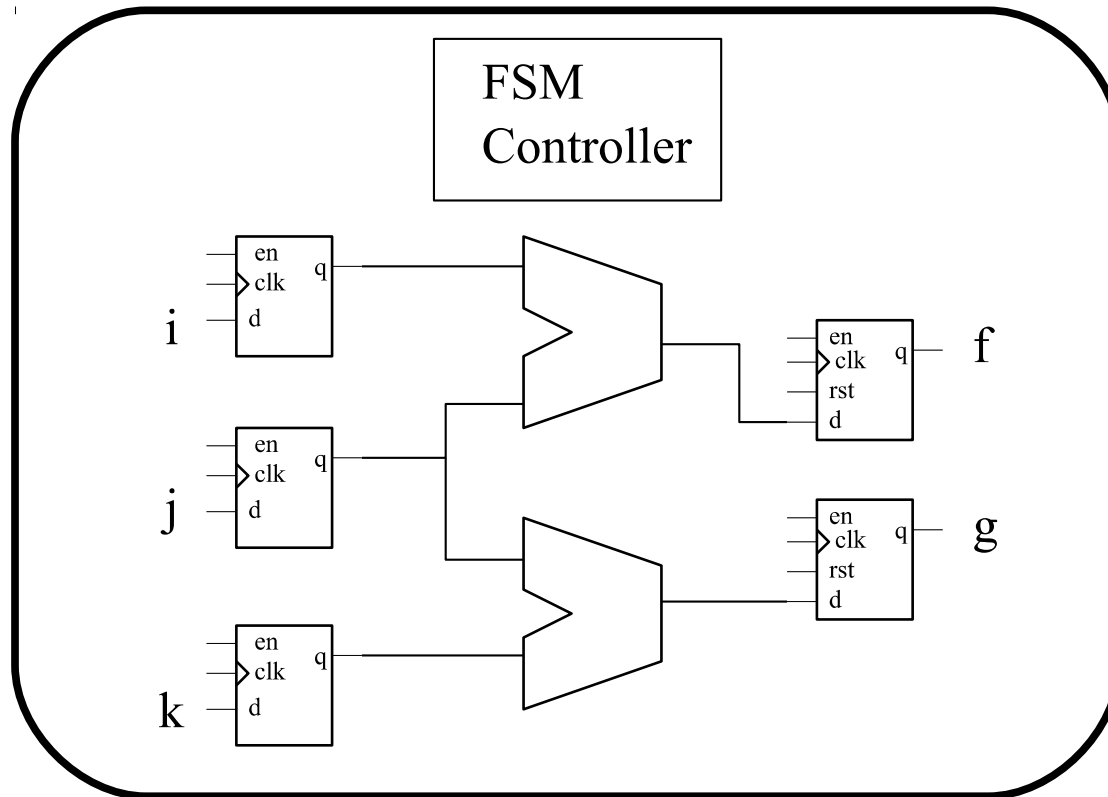
Head

```
module FSM_opt (  
    output reg [7:0] f,  
    output reg [7:0] g,  
    input clk,  
    input wire [7:0] i,  
    input wire [7:0] j,  
    input wire [7:0] k,  
    input rst  
);  
  
reg [7:0] CS;  
reg [7:0] i_int, j_int, k_int;  
  
localparam S_0 = 8'b00000000;  
localparam S_1 = 8'b00000001;  
localparam S_2 = 8'b00000010;
```

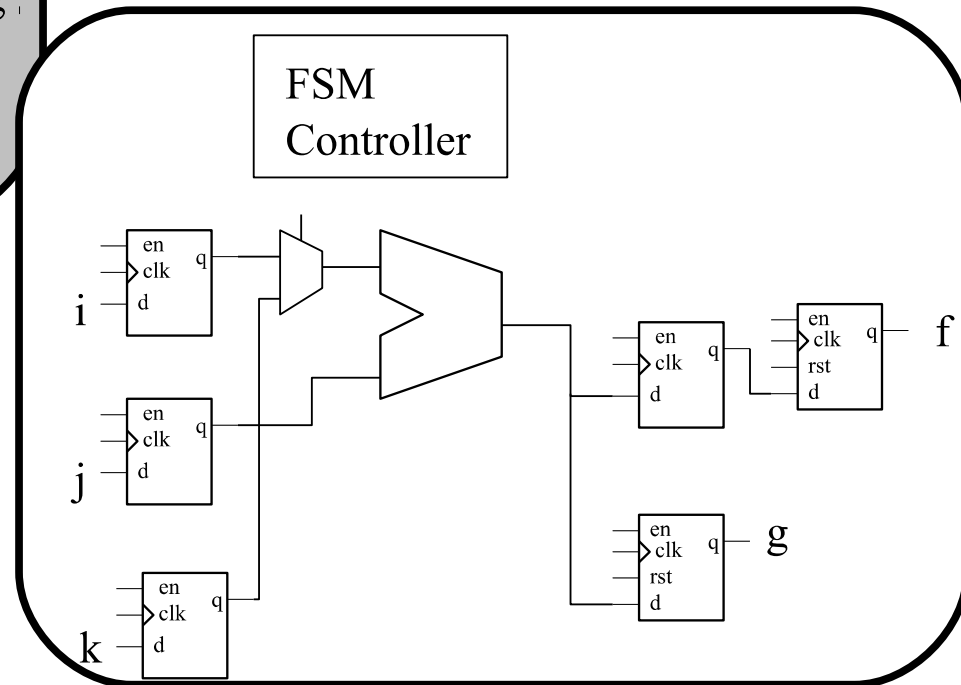
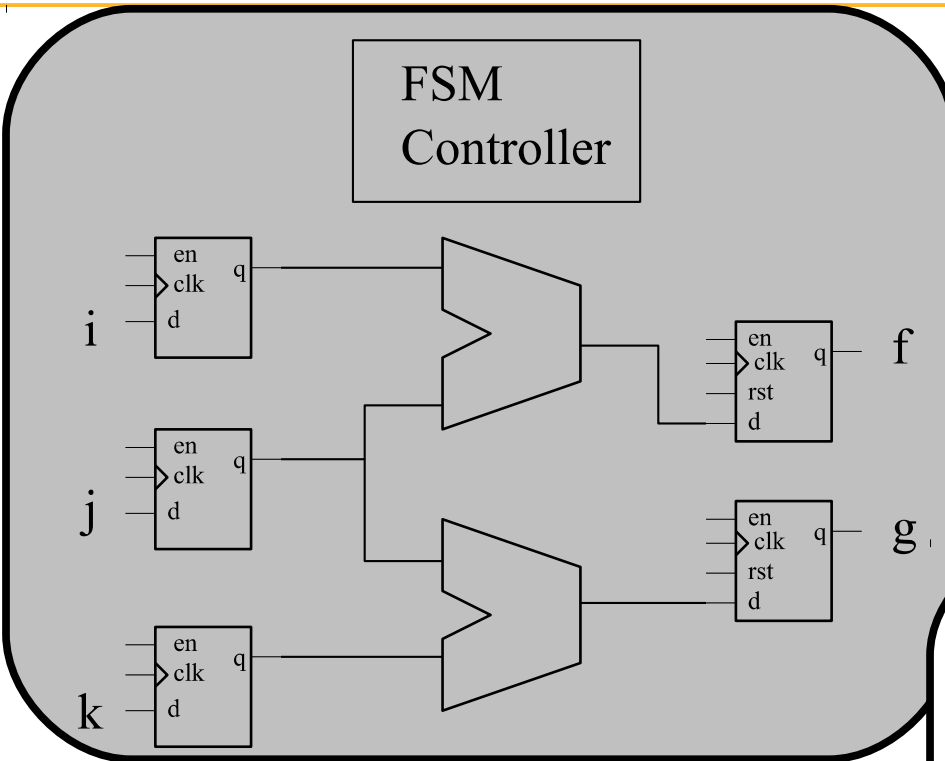
Body

```
always @ (posedge clk) begin  
    if (rst) begin  
        CS<=S_0;  
        f<=0;  
        h<=0;  
    end else begin  
        case (CS)  
            S_0: begin  
                i_int<=i;  
                j_int<=j;  
                k_int<=k;  
                CS<=S_1;    end  
            S_1: begin  
                CS<=S_2;    end  
            S_2: begin  
                f<=i_int*j_int;  
                h<=j_int*k_int;  
                CS<=S_0;    end  
        endcase  
    end  
end  
endmodule
```

Suggested RTL



Alternative RTL with lower gate count



Alt. Module Body

```
always @ (posedge clk) begin
  if (rst) begin
    f<=0; g<=0;
    CS<=S_0;
  end else begin
    case (CS)
      S_0: begin
        i_int<=i;
        j_int<=j;
        k_int<=k;
        CS<=S_1; end

      S_1: begin
        CS<=S_2; end

      S_2: begin
        f<=i_int*j_int;
        g<=j_int*k_int;
        CS<=S_0; end

    endcase
  end
end
endmodule
```

Explicitly
coding the
rescheduling
is simple but
this doesn't
ensure
resource
sharing

```
always @ (posedge clk) begin
  if (rst) begin
    f<=0; g<=0;
    CS<=S_0;
  end else begin
    case (CS)
      S_0: begin
        i_int<=i;
        j_int<=j;
        k_int<=k;
        CS<=S_1; end

      S_1: begin
        f_int<=i_int*j_int;
        CS<=S_2; end

      S_2: begin
        g<=k_int*j_int;
        f<=f_int;
        CS<=S_0; end

    endcase
  end
end
endmodule
```

Another Rescheduling Example

Version 1

```
always @ (posedge clk)
...
case (CS)
S_0:begin q<=r+s;      CS<=S_1; end
S_1:begin              CS<=S_2; end
S_2:begin qout<=q+5;  CS<=S_0; end
...
```

Version 2

```
...
always @ (posedge clk)
...
case (CS)
S_0:begin q<=r+s;      CS<=S_1; end
S_1:begin q<=q+5;      CS<=S_2; end
S_2:begin qout<=q;    CS<=S_0; end
...
```

Another Rescheduling Example

Code Provided to the Synthesizer

```
input  [7:0] i,j,k;
output [7:0] f,h;
reg     [7:0] f,g,h,q,r,s;
always @ (posedge clk)
...
    case (CS)
      S_0:begin f<=i+j;g<=j*23;    CS<=S_1; end
      S_1:begin h<=f+k;            CS<=S_2; end
      S_2:begin f<=f*g; q<=r*s;    CS<=S_0; end
    ...
```

FSM Synthesizers
can automatically
try variations

Externally Equivalent Variation 1:

```
S_0:begin f<=i+j;g<=j*23;    CS<=S_1; end
S_1:begin h<=f+k; q_int<=r*s; CS<=S_2; end
S_2:begin f<=f*g; q<=q_int;  CS<=S_0; end
```

Movement of
 $q=r*s$
using a
temporary
variable

Externally Equivalent Variation 2:

```
S_0:begin f<=i+j;g<=j*23;    CS<=S_1; end
S_1:begin h<=f+k;g<=f*g;     CS<=S_2; end
S_2:begin f<=g; q<=r*s;      CS<=S_0; end
```

Movement of $f*g$

State Encoding

http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/xst.pdf

The state encoding effects the size of the decoder, speed, dependent logic optimization, etc.

Encodings supported by Xilinx:

Auto
One-Hot
Gray
Compact
Johnson
Sequential
User
Speed1
RAM-based

State Encoding

- Auto: In this mode, XST tries to select the best suited encoding algorithm for each FSM.
- One-Hot: One-hot encoding is the default encoding scheme. Its principle is to associate one code bit and also one flip-flop to each state. At a given clock cycle during operation, one and only one bit of the state variable is asserted. Only two bits toggle during a transition between two states. One-hot encoding is very appropriate with most FPGA targets where a large number of flip-flops are available. It is also a good alternative when trying to optimize speed or to reduce power dissipation.

State Encoding

- Gray: Gray encoding guarantees that only one bit switches between two consecutive states. It is appropriate for controllers exhibiting long paths without branching. In addition, this coding technique minimizes hazards and glitches. Very good results can be obtained when implementing the state register with T flip-flops.
- Compact: Compact encoding consists of minimizing the number of bits in the state variables and flip-flops. This technique is based on hypercube immersion. Compact encoding is appropriate when trying to optimize area.

State Encoding

- Johnson: Like Gray, Johnson encoding shows benefits with state machines containing long paths with no branching.
- Sequential: Sequential encoding consists of identifying long paths and applying successive radix two codes to the states on these paths. Next state equations are minimized.

State Encoding

- Speed1: Speed1 encoding is oriented for speed optimization. The number of bits for a state register depends on the particular FSM, but generally it is greater than the number of FSM states.
- User: In this mode, XST uses original encoding, specified in the HDL file. For example, if you use enumerated types for a state register, then in addition you can use the `ENUM_ENCODING` constraint to assign a specific binary value to each state. Please refer to "Design Constraints" chapter for more details.

State Encoding

- RAM-Based Finite State Machine (FSM) Synthesis: Large Finite State Machine (FSM) components can be made more compact and faster by implementing them in the block RAM resources provided in Virtex® devices and later technologies. FSM Style (FSM_STYLE) directs XST to use block RAM resources for FSMs.

Safe Finite State Machine (FSM) Implementation

- XST can add logic to your Finite State Machine (FSM) implementation that will let your state machine recover from an invalid state. If during its execution, a state machine enters an invalid state, the logic added by XST will bring it back to a known state, called a recovery state. This is known as Safe Implementation mode.
- By default, XST automatically selects a reset state as the recovery state. ...

Encoding Summary

- One-hot coding is good for speed and simplicity of state decoding logic and state incrementing.
- More compact codes such as standard binary encoding generally require a smaller state register than one-hot coding at the possible cost of size and speed. But this depends on the density of comb. logic vs. registers the supporting HW platform and in the design. FPGAs have many registers and so the cost of additional combinatorial logic may be large compared to the savings from needing less registers.
- Codes where only one or two bits change at a time in the state register may be beneficial. Less transitions may lead to less power depending on overall design. Can minimize the chance of metastability errors (such as systems with tight timing, or radiation vulnerability). These codes may also minimize logic glitches.

FSM Log File

- The XST log file reports the full information of recognized Finite State Machine (FSM) components during the Macro Recognition step. Moreover, if you allow XST to choose the best encoding algorithm for your FSMs, it reports the one it chose for each FSM. As soon as encoding is selected, XST reports the original and final FSM encoding. If the target is an FPGA device, XST reports this encoding at the HDL Synthesis step. If the target is a CPLD device, then XST reports this encoding at the Low Level Optimization step.

...

Synthesizing Unit <fsm_1>.

Related source file is "/state_machines_1.vhd".

Found finite state machine <FSM_0> for signal <state>.

States	4	
Transitions	5	
Inputs	1	
Outputs	4	
Clock	clk (rising_edge)	
Reset	reset (positive)	
Reset type	asynchronous	
Reset State	s1	
Power Up State	s1	
Encoding	automatic	
Implementation	LUT	

Found 1-bit register for signal <outp>.

Summary:

inferred 1 Finite State Machine(s).

inferred 1 D-type flip-flop(s).

Unit <fsm_1> synthesized.

=====

HDL Synthesis Report
Macro Statistics
Registers : 1

1-bit register : 1
=====

* Advanced HDL Synthesis *

=====

Advanced Registered AddSub inference ...
Analyzing FSM <FSM_0> for best encoding.
Optimizing FSM <state/FSM_0> on signal <state[1:2]> with
gray encoding.

State		Encoding
-------	--	----------

s1		00
s2		01
s3		11
s4		10

=====

HDL Synthesis Report
Macro Statistics
FSMs : 1

Constraints/Compiler Directives

- Most synthesizers support various instructions to modify synthesis behavior

• http://www.xilinx.com/itp/xilinx8/books/data/docs/xst/xst0064_8.html#wp255324

- Many options can be applied globally or on a per module instance or even per block level.
- Some are entered in constraint files, as a command-line option or inline via special commented tags:

```
casex select // synthesis full_case
4'b1xxx: res = data1;
4'bx1xx: res = data2;
4'bxx1x: res = data3;
4'bxxx1: res = data4;
```

Compiler directive/constraint



Xilinx FSM Constraints

- Xilinx Synthesis Constraints for state machines
 - http://www.xilinx.com/itp/3_1i/data/fise/xst/chap02/xst02014.htm
- Related constraints are:
 - fsm_extract
 - determines if state machines are detected/extracted
 - http://www.xilinx.com/itp/xilinx7/books/data/docs/cgd/cgd0093_54.html
 - fsm_encoding
 - can set state encoding globally or per-instance
 - http://www.xilinx.com/itp/xilinx7/books/data/docs/cgd/cgd0092_53.html
 - fsm_fftype
 - use D or toggle flip flops for state register
 - <http://www.xilinx.com/itp/xilinx4/data/docs/cgd/f8.html>
 - enum_encoding
 - sets encoding when fsm_extract is used to select user
 - <http://www.xilinx.com/itp/xilinx4/data/docs/cgd/e3.html>

A word on default case

- You should not automatically add default case to synthesize a FSM since the logic has to cover many unnecessary states
 - Check documentation for your synthesizer's behavior
 - http://www.trilobyte.com/pdf/golson_snug94.pdf
 - Special effort may be required to have the synthesizer ignore the default case yet allow logging in simulation:

```
// synopsys translate_off  
default: $display("He's dead, Jim.") ;  
// synopsys translate_on
```

A word on states, the state register, and registers for variables serving as extended state variables

- In-class example given
- Technically, every register in a digital system is a part of the state. What variables you decide to think of as state in your state diagram and what is coded in the CS register and used in the case statement is up to you.
- When there are many similar states, sometimes combining them in code and adding a register for a variable makes sense. This can reduce the state decoding and state logic and may make the code more maintainable and easier to read.

Additional Synthesis Options and Directives

http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/pp_db_xst_hdl_synthesis_options.htm