The air is clean. The water is clean. Even the dirt is clean! Bowling averages are way up. Minigolf scores are way down. And we have more excellent

Microcontrollers and FPGAs

than any other planet we communicate with. I'm telling you, this place is great!

Ah, but don't worry: it'll all make sense. I'm a professional.

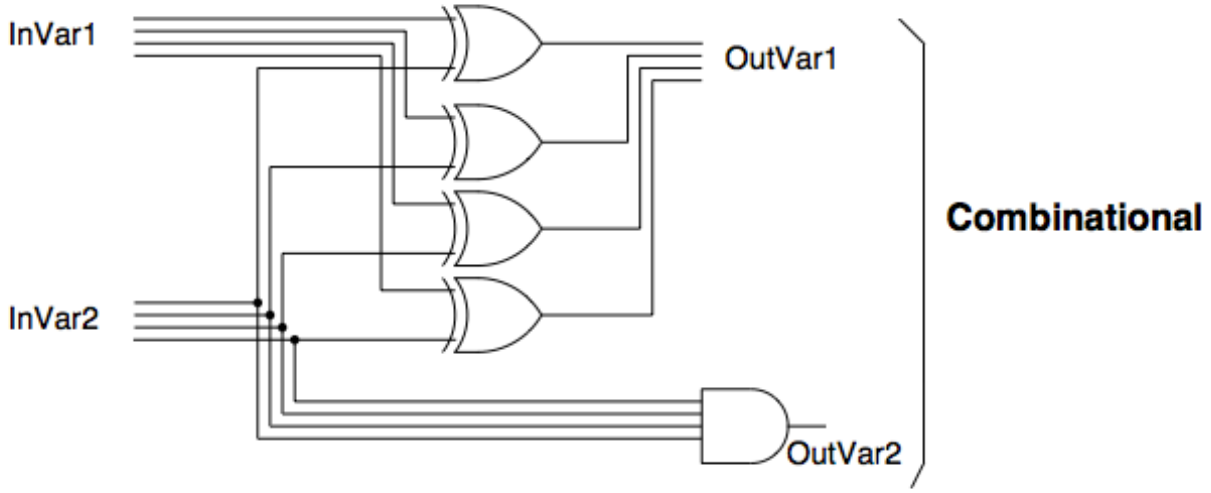
Embedded Systems

Verilog

- Verilog is a Hardware Description Language (HDL).
- It is a surprisingly big language, but most of its instructions are useful for simulation only. The number of structures that can actually be put in an FPGA is much smaller.
- An HDL is not a programming language, it is a formalism for describing digital logic. As such, it's vitally important that you understand how every piece of Verilog that you write will actually look when it has been synthesized in to hardware.
- If your Verilog isn't working, try sketching the logic gates and registers yourself. If you can't make your Verilog look like good hardware, then the synthesizer probably can't either!
- Two primary classes of logic: Combinatorial and Sequential

Verilog

Combinatorial logic with **assign** statements



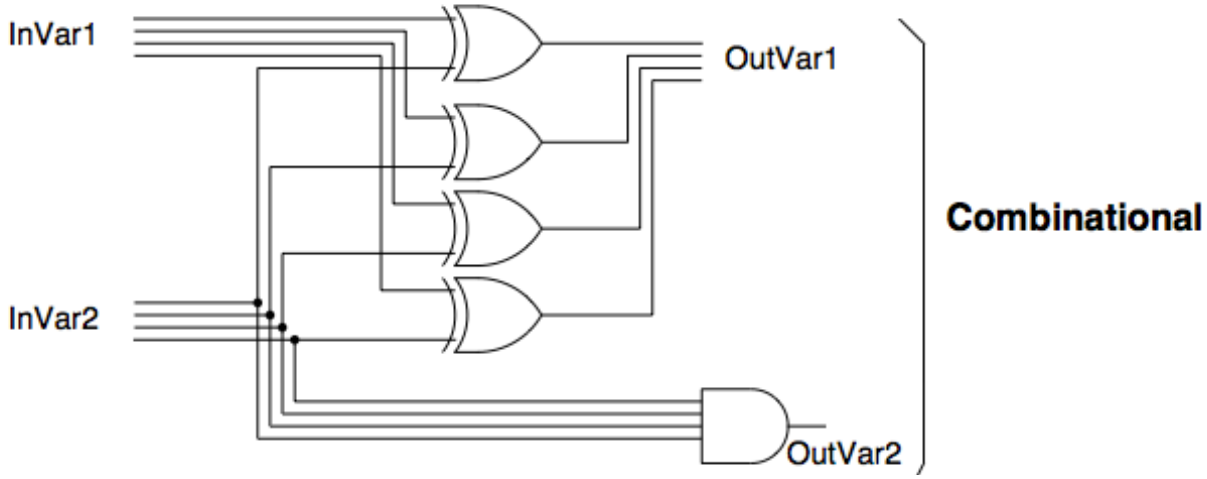

```
module Module_Name( input wire  [3:0] InVar1,
                    input wire  [3:0] InVar2,
                    output wire [3:0] OutVar1,
                    output wire       OutVar2);

    assign OutVar1 = InVar1^InVar2;
    assign OutVar2 = &InVar2;

endmodule
```

Verilog

Combinatorial logic with **always** statements




```
module Module_Name( input wire  [3:0] InVar1,
                    input wire  [3:0] InVar2,
                    output reg [3:0] OutVar1,
                    output reg       OutVar2);

    always @(*) begin
       OutVar1 = InVar1^InVar2;
       OutVar2 = &InVar2;
    end


endmodule
```

Verilog

Rules for combinatorial circuits:

- No feedback allowed.
- **assign** or **always(*)** statements
  - Assigns are more compact and often used for simple wiring tasks like `assign reset_n = key[0];`
  - Always statements are more flexible and lead in to sequential logic
  - The signals between the brackets in the always statement is called the **sensitivity list**. You don't *need* to have a '*' for combinatorial logic, you can list each input signal separately, however if you forget to enter a signal then the logic is no longer strictly combinatorial and may not synthesize as you expect.

```
assign X = Y | Z;
assign Y = X ^ W;
```

Wrong!

Verilog

If statements

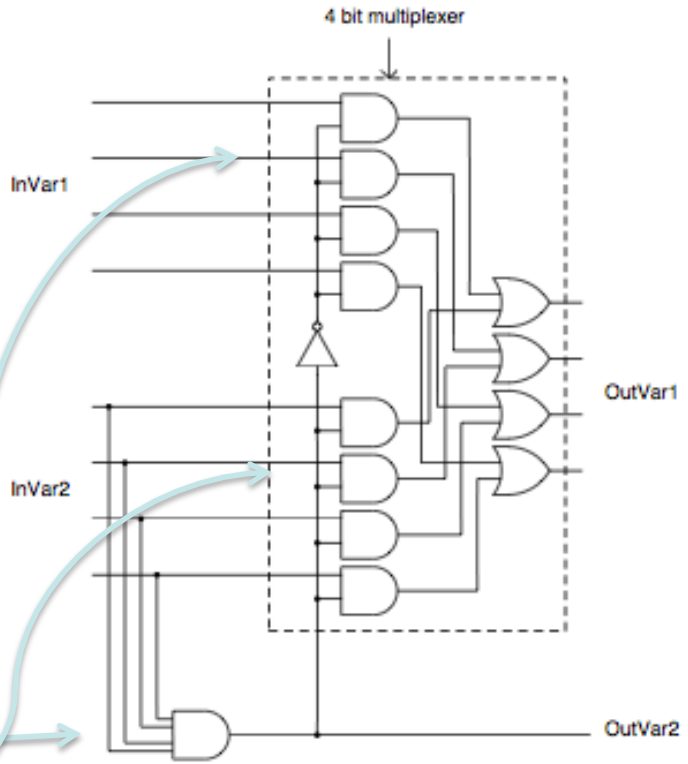
```
module Module_Name( input wire  [3:0] InVar1,
                    input wire  [3:0] InVar2,
                    output reg  [3:0] OutVar1,
                    output reg        OutVar2);

    always @(*) begin

        OutVar2 = &InVar2;
        if(&InVar2) begin
             OutVar1 = InVar2;
        end else begin
             OutVar1 = InVar1;
        end

    end

endmodule
```



Verilog

If Statements:

- Synthesize to two distinct pieces.
  - A wire representing the outcome of the test
  - A multiplexor that selects the correct input signal given the above wire
- Cases should be complete! They don't strictly *have* to be, the below code is valid, but the synthesizer will create latches for to cover the unused case. Latches are almost never what you actually want.

```
always @(*) begin
   if(&InVar2) begin
        OutVar1 =  InVar2;
        OutVar2 = ^InVar1;
   end
end
```

Verilog

If Statements:

- Also note that two latches do not equal a correct multiplexor. The synthesizer will probably not be smart enough what you meant with the code below, even if it's obvious to you

```
always @(*) begin
     if (&InVar1) begin
          OutVar1 = InVar1;
     end
end

always @(*) begin
     if (!(&InVar1)) begin
          OutVar1 = ~InVar1;
     end
end
```

Verilog

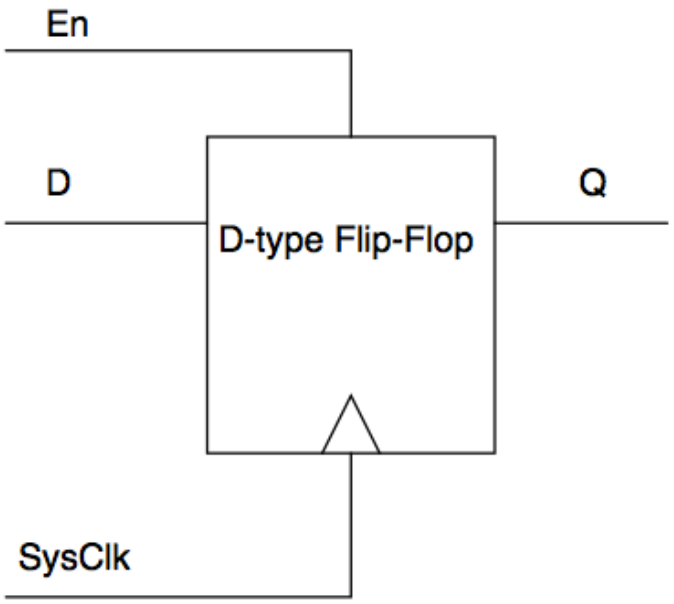
Case Statements:

- If statements are special case statements.
- Like if statements, case statements synthesize to two parts:
  - A wire (generally a bus of wires) representing all possible selections of the case statement
  - A multiplexor that selects the correct input given the above wire
- Like if statements, they should be complete to avoid the generation of latches

Verilog

Sequential logic

The ‘D’ flip-flop



Verilog

Sequential logic:

- Use non-blocking assignments, i.e `<=`
- Always block sensitivity list should be edge-triggered
  - `always @(posedge clk)`
- All signals inside the block must be registered to the clock edge. If your logic is working intermittently, try registering your wires before use

```
always @(posedge clk) clear_syn <= clr;

always @(posedge clk) begin
if(clr_syn) q <= 0;
            else q <= d;
end
```

- In sequential blocks, if and case statements need not be complete. Unmatched cases effectively synthesize to `OutVar <= OutVar`

Verilog

Important point: What is the 'System Clock'?

- When talking about sequential logic, we've been talking about a system clock. This is often simply a net routed from a pin in the top level file
- If this clock rate is not right for you, what do you do?

```
reg [1:0] my_clock;

always @(posedge clk) begin
    my_clock <= my_clock + 1;
end

always @(posedge my_clock[1]) begin
    …
end
```

Wrong!

Verilog

Why? Internally `my_clock[1]` is a logic net, not a clock net (gclk in Altera-speak). This won't matter for slow-speed designs, but as speed increases the time it takes for a logic signal to propagate through the chip begins to matter.

This is our first example of having to think about things in the time domain but it won't be our last!

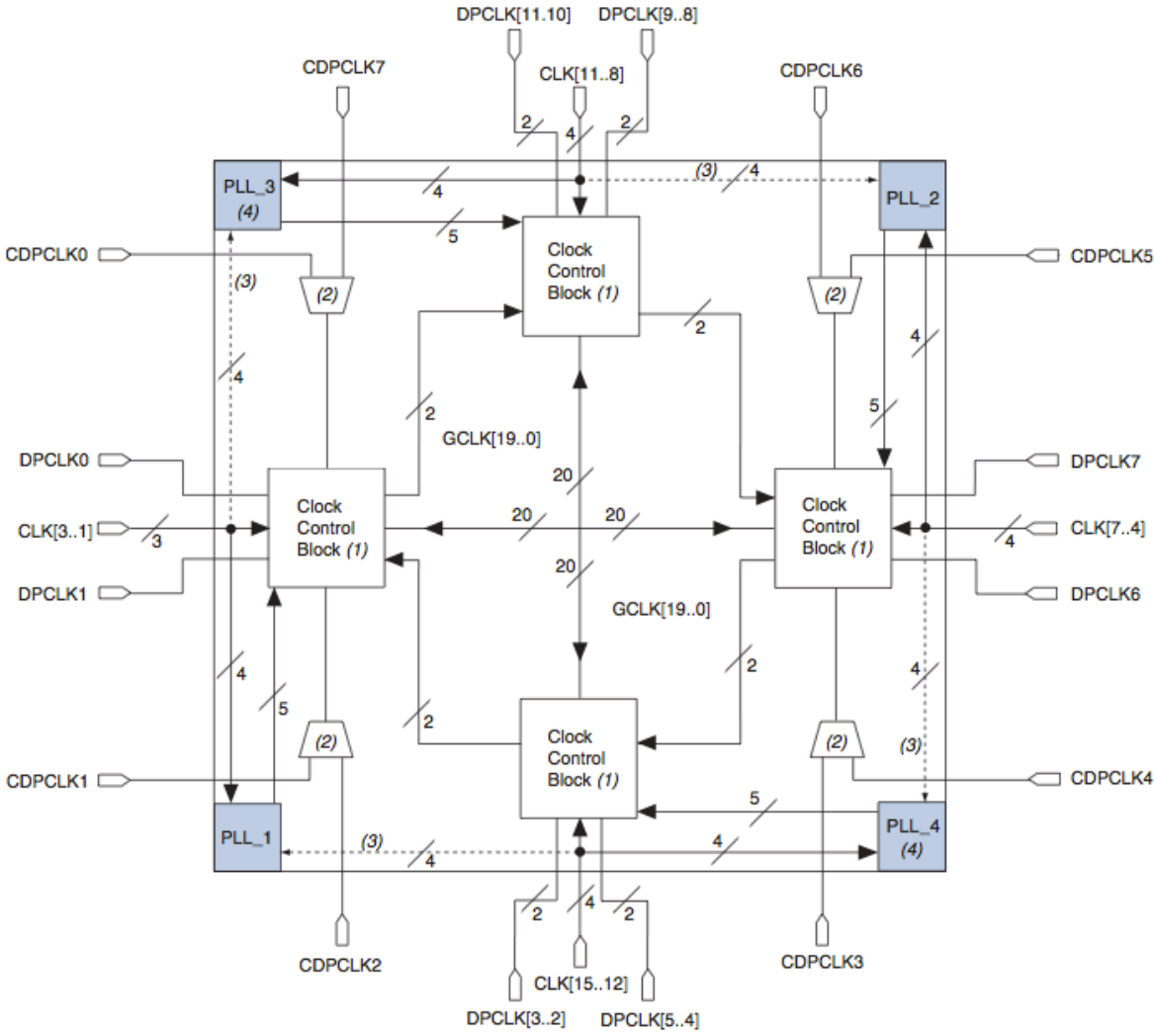
The correct way to do this is to have the external clock in the sensitivity list but use `my_clock[1]` in the always body to drive a state machine or otherwise determine what to actually *do*.

Clocks

The exception is when you can use dedicated hardware on the chip to modify the clock and route it back on to a gclk.

The clock infrastructure of the Cyclone IV part on our DE2 boards is pictured here.

If you need a clock that can't be generated easily by the tricks on the previous slide, you can use a Phase Locked Loop (PLL)



Clocks

A PLL is instantiated like any other Verilog sub-module but the synthesizer recognizes that it's special and builds the logic out of a PLL hardware block instead of generic gates.

PLLs and a host of other pieces of logic can be created using the Altera MegaCore Plugin Wizard. The generated code to turn a 50MHz clock in to 100MHz is shown to the right, obviously not the sort of thing you want to write by hand.

```
altpll          altpll_component (
                                .inclk (sub_wire3),
                                .clk (sub_wire0),
                                .activeclock (),
                                .areset (1'b0),
                                .clkbad (),
                                .clkena ({6{1'b1}}),
                                .clkloss (),
                                .clkswitch (1'b0),
                                .configupdate (1'b0),
                                .enable0 (),
                                .enable1 (),
                                .extclk (),
                                .extclkena ({4{1'b1}}),
                                .fbin (1'b1),
                                .fbmimicbidir (),
                                .fbout (),
                                .fref (),
                                .icdrclk (),
                                .locked (),
                                .pfdena (1'b1),
                                .phasecounterselect ({4{1'b1}}),
                                .phasedone (),
                                .phasestep (1'b1),
                                .phaseupdown (1'b1),
                                .pllena (1'b1),
                                .scanaclr (1'b0),
                                .scanclk (1'b0),
                                .scanclkena (1'b1),
                                .scandata (1'b0),
                                .scandataout (),
                                .scandone (),
                                .scanread (1'b0),
                                .scanwrite (1'b0),
                                .sclkout0 (),
                                .sclkout1 (),
                                .vcooverrange (),
                                .vcounderrange ());
defparam
                                altpll_component.bandwidth_type = "AUTO",
                                altpll_component.clk0_divide_by = 1,
                                altpll_component.clk0_duty_cycle = 50,
                                altpll_component.clk0_multiply_by = 2,
                                altpll_component.clk0_phase_shift = "0",
                                altpll_component.compensate_clock = "CLK0",
                                altpll_component.inclk0_input_frequency = 37037,
                                altpll_component.intended_device_family = "Cyclone IV E",
                                altpll_component.lpm_type = "altpll",
                                altpll_component.operation_mode = "NORMAL",
                                altpll_component.pll_type = "AUTO",
                                altpll_component.port_activeclock = "PORT_UNUSED",
                                altpll_component.port_areset = "PORT_UNUSED",
                                altpll_component.port_clkbad0 = "PORT_UNUSED",
                                altpll_component.port_clkbad1 = "PORT_UNUSED",
                                altpll_component.port_clkloss = "PORT_UNUSED",
                                altpll_component.port_clkswitch = "PORT_UNUSED",
                                altpll_component.port_configupdate = "PORT_UNUSED",
                                altpll_component.port_fbin = "PORT_UNUSED",
                                altpll_component.port_inclk0 = "PORT_USED",
                                altpll_component.port_inclk1 = "PORT_UNUSED",
                                altpll_component.port_locked = "PORT_UNUSED",
                                altpll_component.port_pfdena = "PORT_UNUSED",
                                altpll_component.port_phasecounterselect = "PORT_UNUSED",
                                altpll_component.port_phasedone = "PORT_UNUSED",
                                altpll_component.port_phasestep = "PORT_UNUSED",
                                altpll_component.port_phaseupdown = "PORT_UNUSED",
                                altpll_component.port_pllena = "PORT_UNUSED",
                                altpll_component.port_scanaclr = "PORT_UNUSED",
                                altpll_component.port_scanclk = "PORT_UNUSED",
                                altpll_component.port_scanclkena = "PORT_UNUSED",
                                altpll_component.port_scandata = "PORT_UNUSED",
                                altpll_component.port_scandataout = "PORT_UNUSED",
                                altpll_component.port_scandone = "PORT_UNUSED",
                                altpll_component.port_scanread = "PORT_UNUSED",
                                altpll_component.port_scanwrite = "PORT_UNUSED",
                                altpll_component.port_clk0 = "PORT_USED",
                                altpll_component.port_clk1 = "PORT_UNUSED",
                                altpll_component.port_clk2 = "PORT_UNUSED",
                                altpll_component.port_clk3 = "PORT_UNUSED",
                                altpll_component.port_clk4 = "PORT_UNUSED",
                                altpll_component.port_clk5 = "PORT_UNUSED",
                                altpll_component.port_clkena0 = "PORT_UNUSED",
                                altpll_component.port_clkena1 = "PORT_UNUSED",
                                altpll_component.port_clkena2 = "PORT_UNUSED",
                                altpll_component.port_clkena3 = "PORT_UNUSED",
                                altpll_component.port_clkena4 = "PORT_UNUSED",
                                altpll_component.port_clkena5 = "PORT_UNUSED",
                                altpll_component.port_extclk0 = "PORT_UNUSED",
                                altpll_component.port_extclk1 = "PORT_UNUSED",
                                altpll_component.port_extclk2 = "PORT_UNUSED",
                                altpll_component.port_extclk3 = "PORT_UNUSED",
                                altpll_component.width_clock = 5;
```

Clocks

Aside: A fully-functional FM radio transmitter using a PLL in 7 lines:

```
pll clockgen(clk, clk2);

always @(negedge clk)
begin
    ctr <= ctr + 1;
    pwm <= wav_dat < ctr[7:0] ? 0 : 1;
end

assign out = pwm ? clk : clk2;
```

Get a 100MHz carrier using a Wizard-generated PLL

Generate a PWM signal with duty cycle proportional to the sound data

The output is just the carrier clock modulated by the PWM signal

Cheated a bit, not shown is the bit that loads `wav_dat` with the current waveform point, nor all the definitions.

Clocks

Aside: A fully-functional FM radio transmitter using a PLL in 7 lines:

```
pll clockgen(clk, clk2);

always @(negedge clk)
begin
     ctr <=
     pwm <=                                  0 : 1;
end

assign out = pwm ? clk : clk2;
```

Project idea!

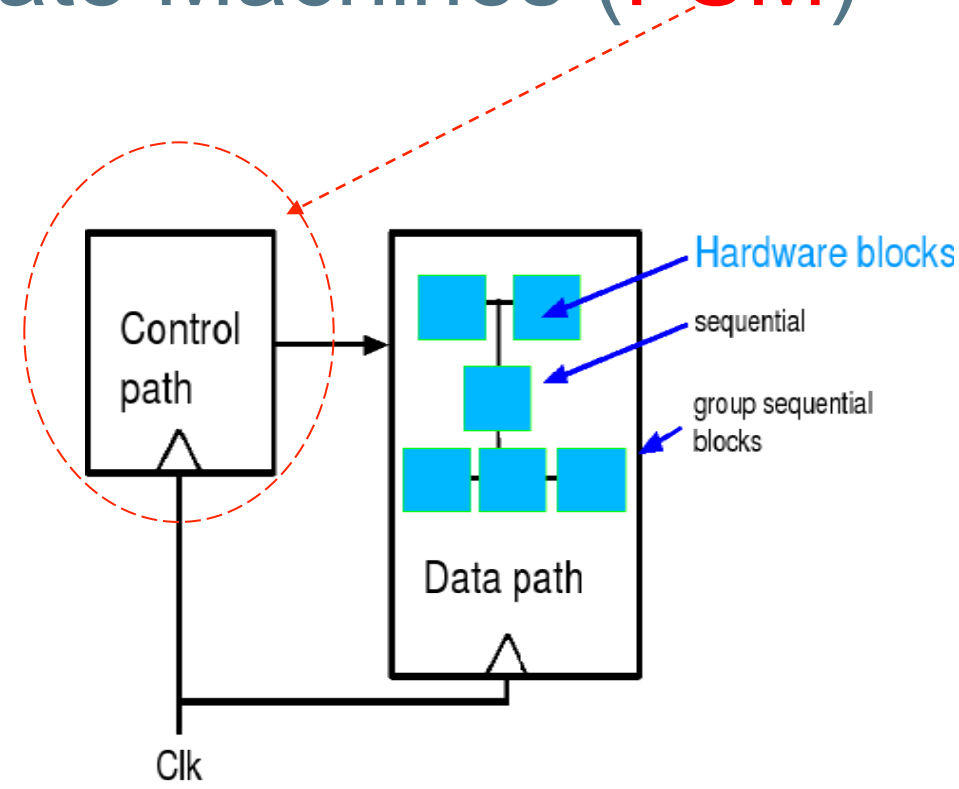
Get a 100MHz carrier using a Wizard-generated PLL

Generate a PWM signal with duty cycle proportional to the sound data

The output is just the carrier clock modulated by the PWM signal

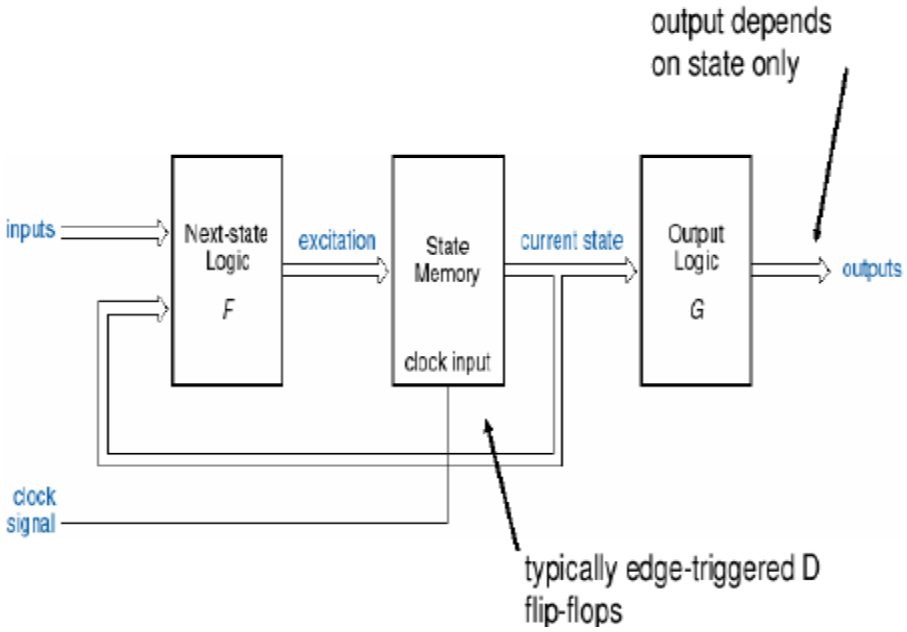
Cheated a bit, not shown is the bit that loads `wav_dat` with the current waveform point, nor all the definitions.

Finite State Machines (FSM)

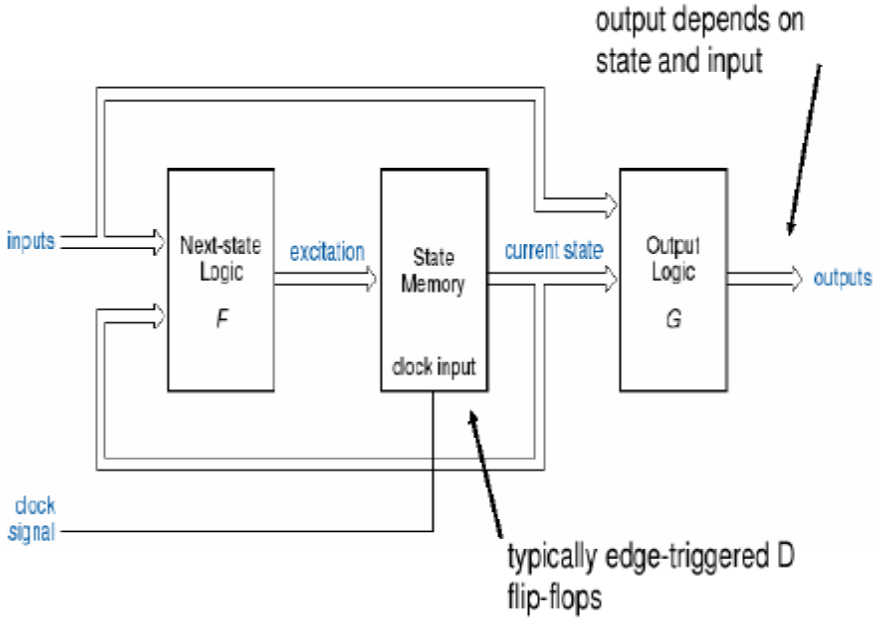


FSM

State-machine structure (Moore)



State-machine structure (Mealy)



FSM

- Let x(t) - input, s(t) - state, z(t) - output
- *Moore machine:*

  s(t+1) = F(s(t), x(t)) −state equation

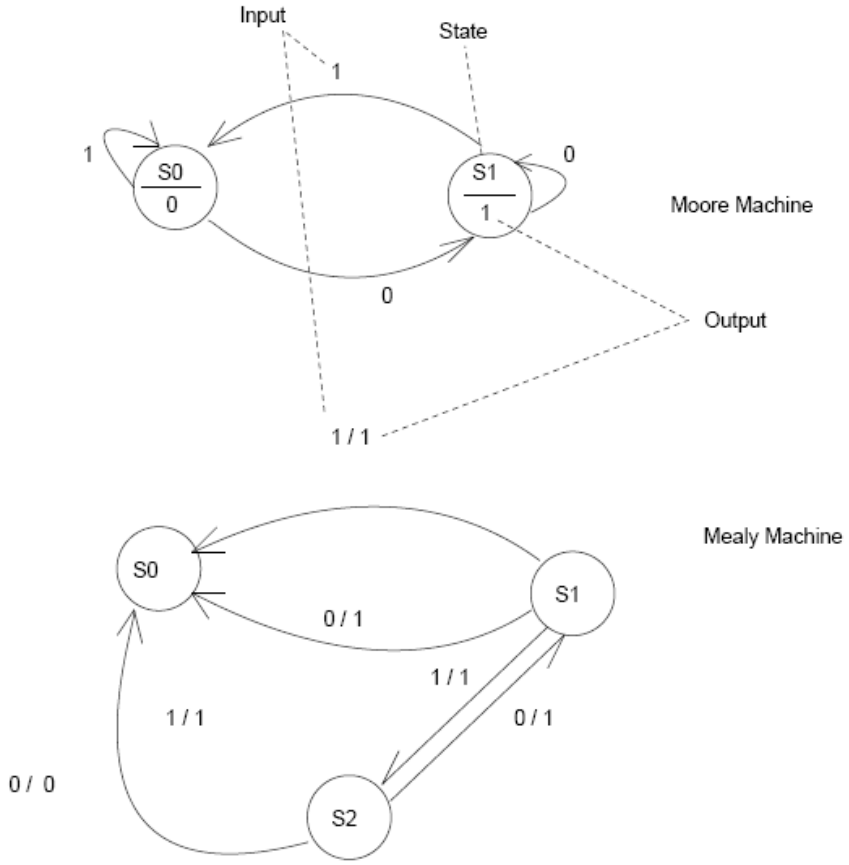
  z(t) = G(s(t)) −output equation
- *Mealy machine:*

  s(t+1) = F(s(t), x(t)) −state equation

  z(t) = G(s(t), x(t)) −output equation

- ENGN3213 only uses Synchronous State Machines

FSM


Input
State
1
1
S0
0
S1
1
0
Moore Machine
0
Output
1 / 1
Mealy Machine
S0
S1
0 / 1
1 / 1
1 / 1
0 / 1
0 / 0
S2

FSM in Verilog

http://www.altera.com/support/examples/verilog/ver_statem.html

statem.v

```
module statem(clk, in, reset, out);

input clk, in, reset;
output [3:0] out;

reg [3:0] out;
reg [1:0] state;

parameter zero=0, one=1, two=2, three=3;

always @(state)
     begin
          case (state)
               zero:
                    out = 4'b0000;
               one:
                    out = 4'b0001;
               two:
                    out = 4'b0010;
               three:
                    out = 4'b0100;
               default:
                    out = 4'b0000;
          endcase
     end

always @(posedge clk or posedge reset)
     begin
          if (reset)
               state = zero;
          else
               case (state)
                    zero:
                         state = one;
                    one:
                         if (in)
                              state = zero;
                         else
                              state = two;
                    two:
                         state = three;
                    three:
                         state = zero;
               endcase
     end

endmodule
```

FSM – Design Steps

1. Determine the inputs / outputs. **Determine the states and give them mnemonic names**

2. Draw up a state diagram and a next state table.

3. Render the inputs, outputs and states in binary format (VERILOG parameter).

4. Draw an **excitation table**

5. Draw an **Output table**

6. Use K-maps to obtain produce minimal next state and output combinational logic.

7. Use the standard VERILOG formulation to simulate your design and check for correct operation. Revise as appropriate.

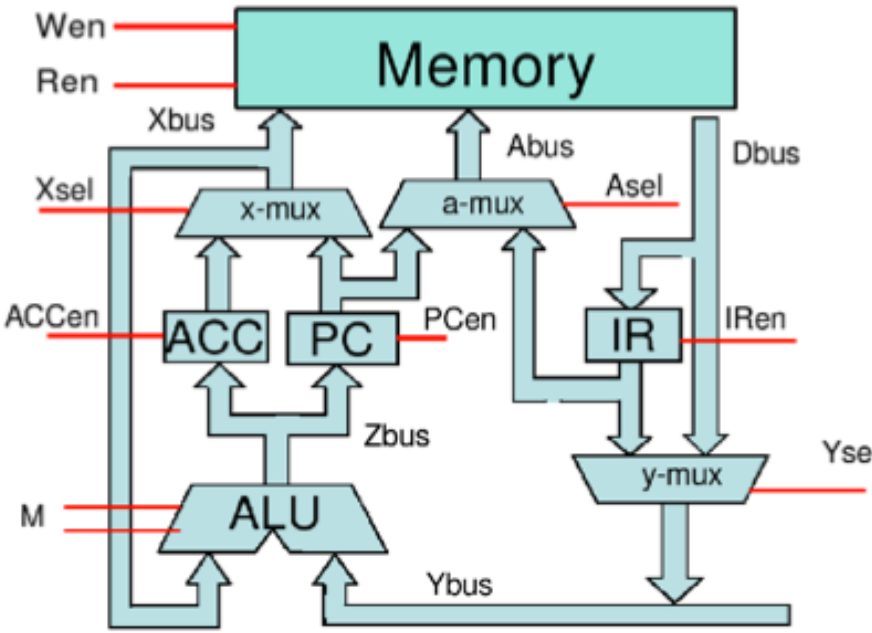
8. Check for potential practical problems (e.g. non-ideal effects).

Microprocessors

MU0

A realistic but simple processor which is capable of executing one instruction every two clock cycles.

We won't be looking at this in labs, but the data path and control flows will be used in illustrating many of the more advanced topics in microprocessors over the coming lectures

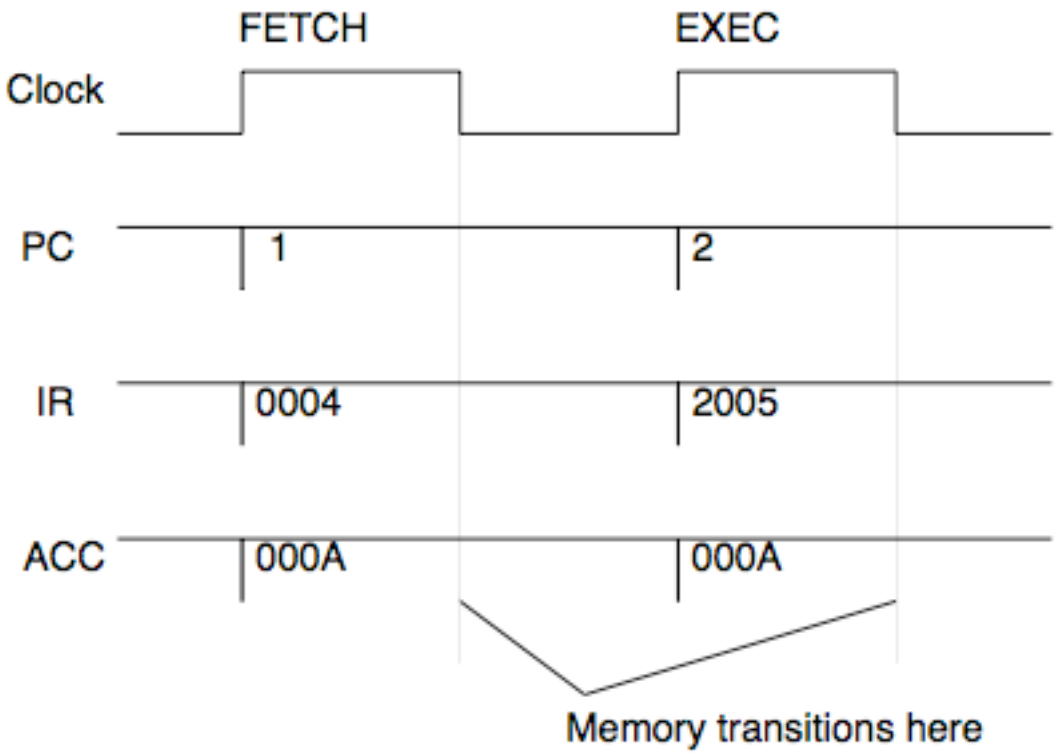


Microprocessors

Timing is important to understand. The processing of a single instruction is split in to two parts, one per clock cycle: Fetch and Execute.

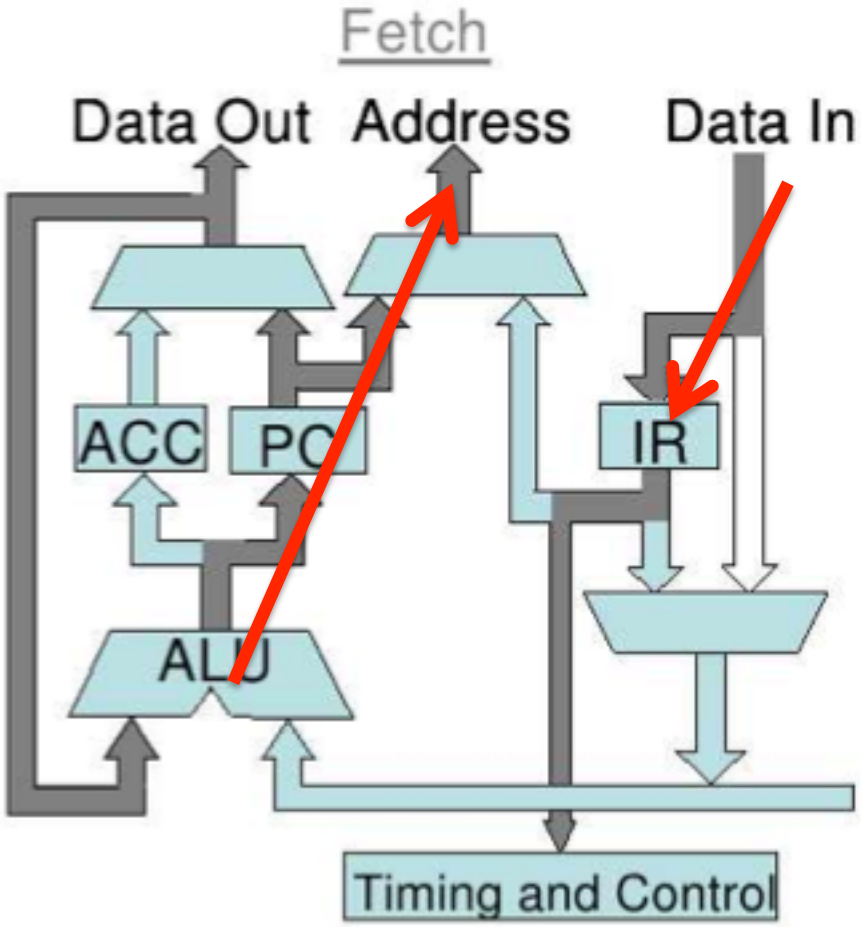
Thus each instruction takes two clock cycles to complete.

Actually there are four states with two occurring on the falling edges. This is valid as MU0 is simple and runs slowly. When we get to more complex systems, we'll note that these extra states are made explicit



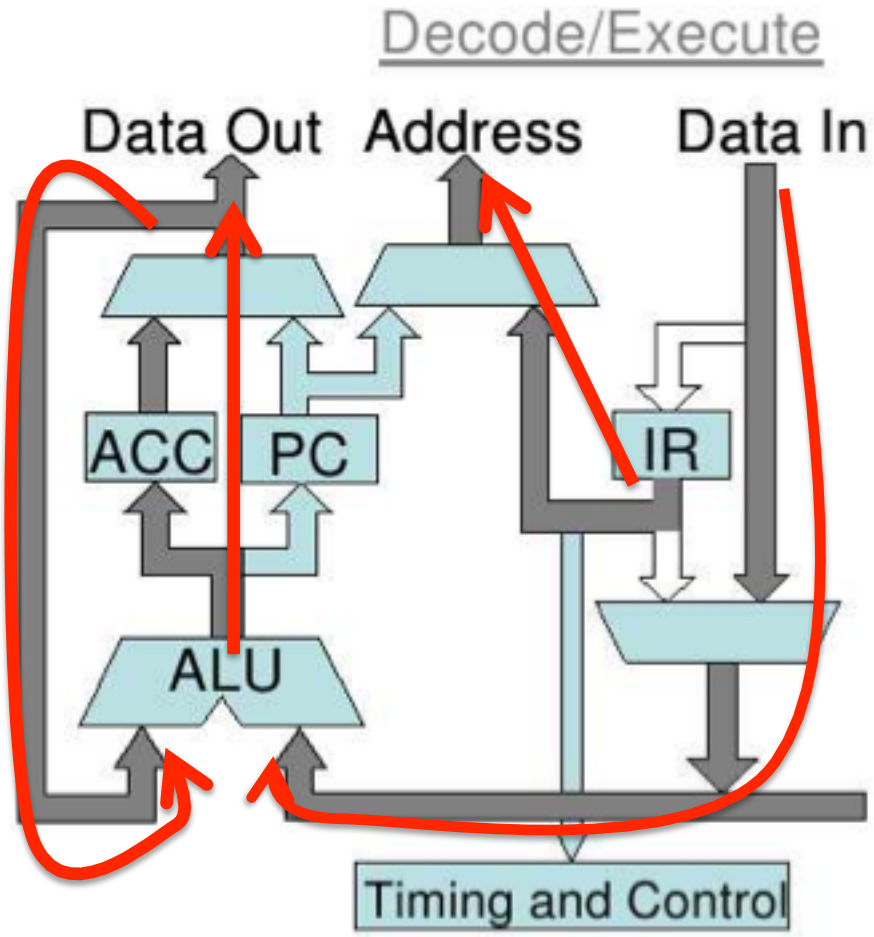
Microprocessors

During the Fetch cycle, the Program Counter is wired in to the memory address, and the instruction at that location is loaded in to the Instruction Register.



Microprocessors

The Execute phase is centred around the Arithmetic Logic Unit (ALU). It takes two operands, one of which is always a register (either the Accumulator or the Program Counter depending on the operation). The other is either a memory location whose address is coded in to the instruction, or some portion of the instruction itself.

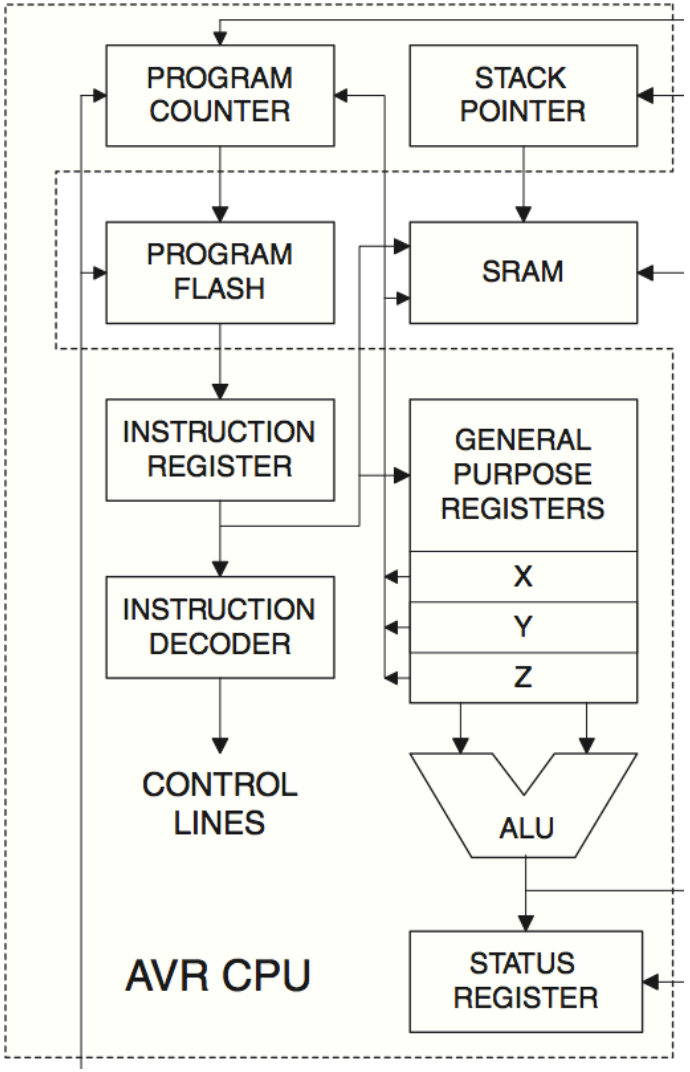


Microprocessors

This basic structure is very similar across all CPUs. Here we see the CPU portion of the Atmel ATMega8 microcontroller.

Program Counter, Instruction Register, ALU, Memory busses all pretty much the same.

Differences: MU0 has one working register, ACC, Mega8 has 32 (6 of which can be paired in to 3 special registers, XYZ, of twice the size). Different memory architecture, MU0 has a single memory to hold instructions and data, the AVR has two blocks labeled Program Flash and SRAM respectively.

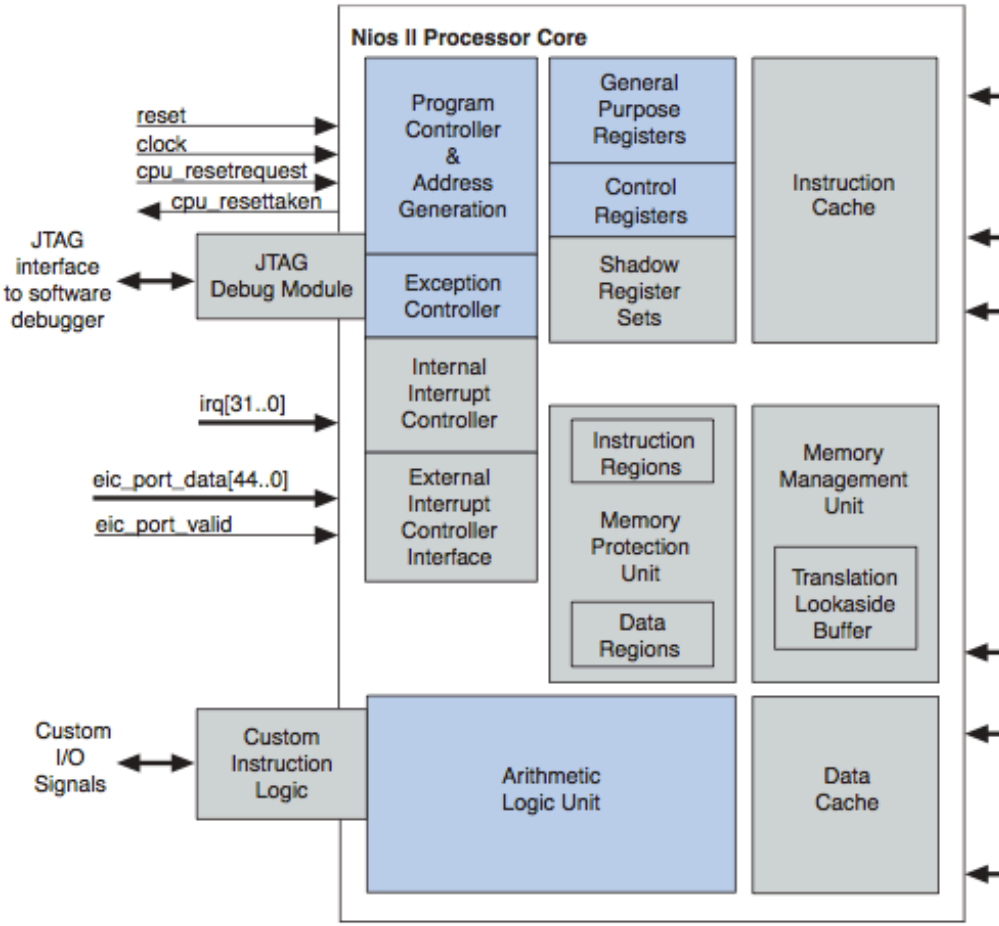


Microprocessors

The Nios2 core we will be using in labs is more complex again, but uses the same concepts still.

Note the register sets, the ALU and the Program Controller (that includes the Program Counter, Instruction Register etc.).

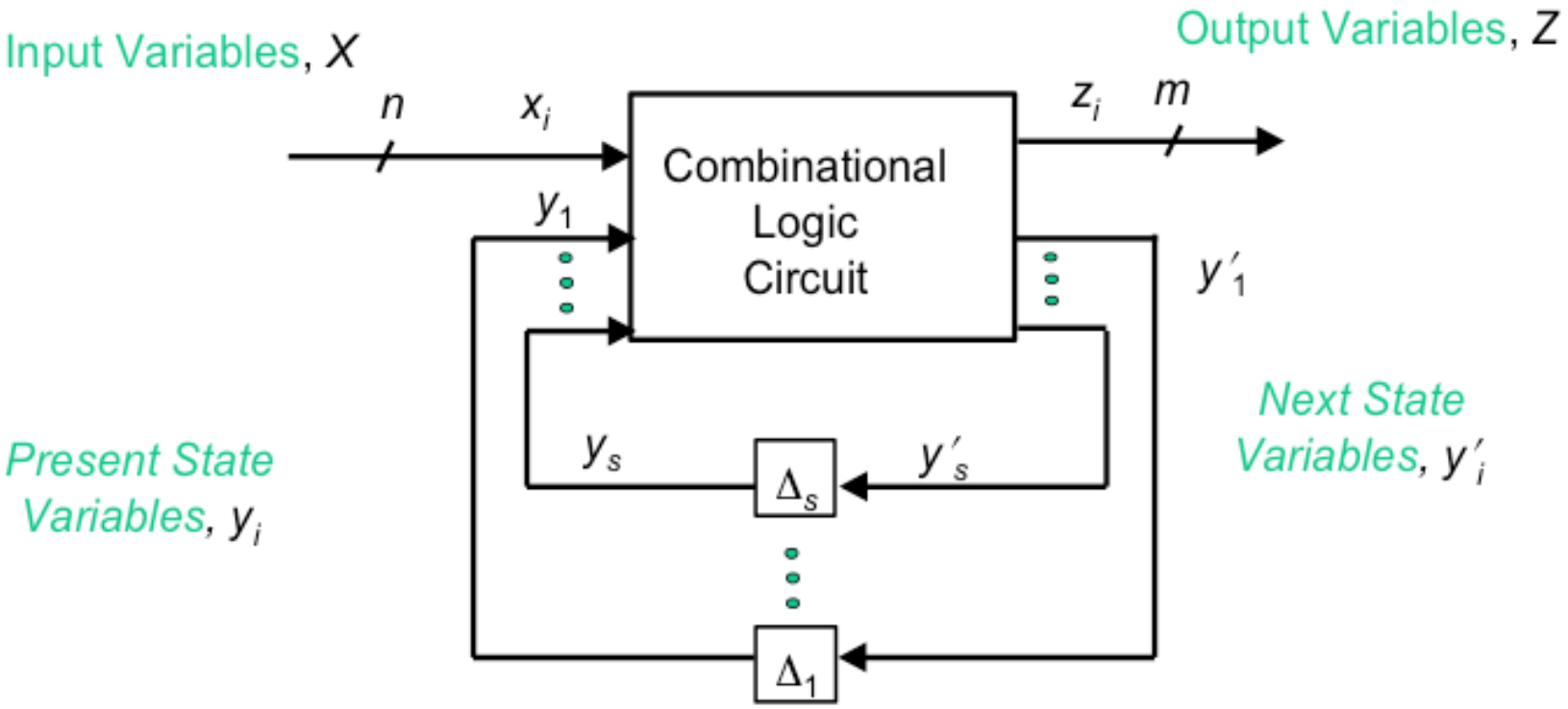
Most of the extra complexity here is to do with interrupts (not yet covered) or simply accessing many more types of memories.



Optional: Asynchronous Machines

- Used some slides from Prof.Marek Perkowski (PSU, ECE573-2009)

Feedback Model for Asynchronous Sequential Networks



Asynchronous FSM

Fundamental Mode Assumption

- Only one input *can change* at a time
  - Analysis too complicated if multiple inputs are allowed to change simultaneously
- Circuit must be allowed to settle to its final value before an input is allowed to change
  - Behavior is *unpredictable* (nondeterministic) if circuit not allowed to settle

Asynch. Design Difficulties

Delay in Feedback Path

- Not reproducible from implementation to implementation
- Variable
  - may be temperature or electrical parameter dependent within the same device
- Analog
  - not known exactly

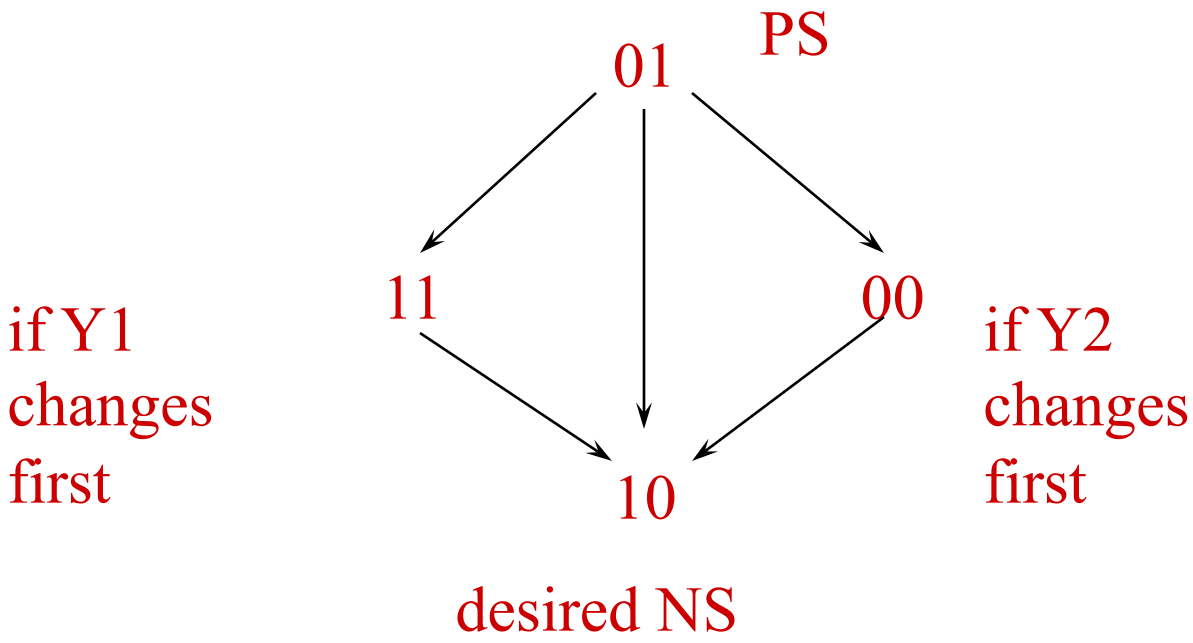
Stable State

- PS = present state
- NS = next state
- PS = NS = Stability
  - Machine may pass through none or more intermediate states on the way to a stable state
  - Desired behavior since only time delay separates PS from NS
- Oscillation
  - Machine never stabilizes in a single state

Races

- A Race Occurs in a Transition From One State to the Next When More Than One Next State Variables Changes in Response to a Change in an Input
- Slight Environment Differences Can Cause Different State Transitions to Occur
  - Supply voltage
  - Temperature, *etc*.

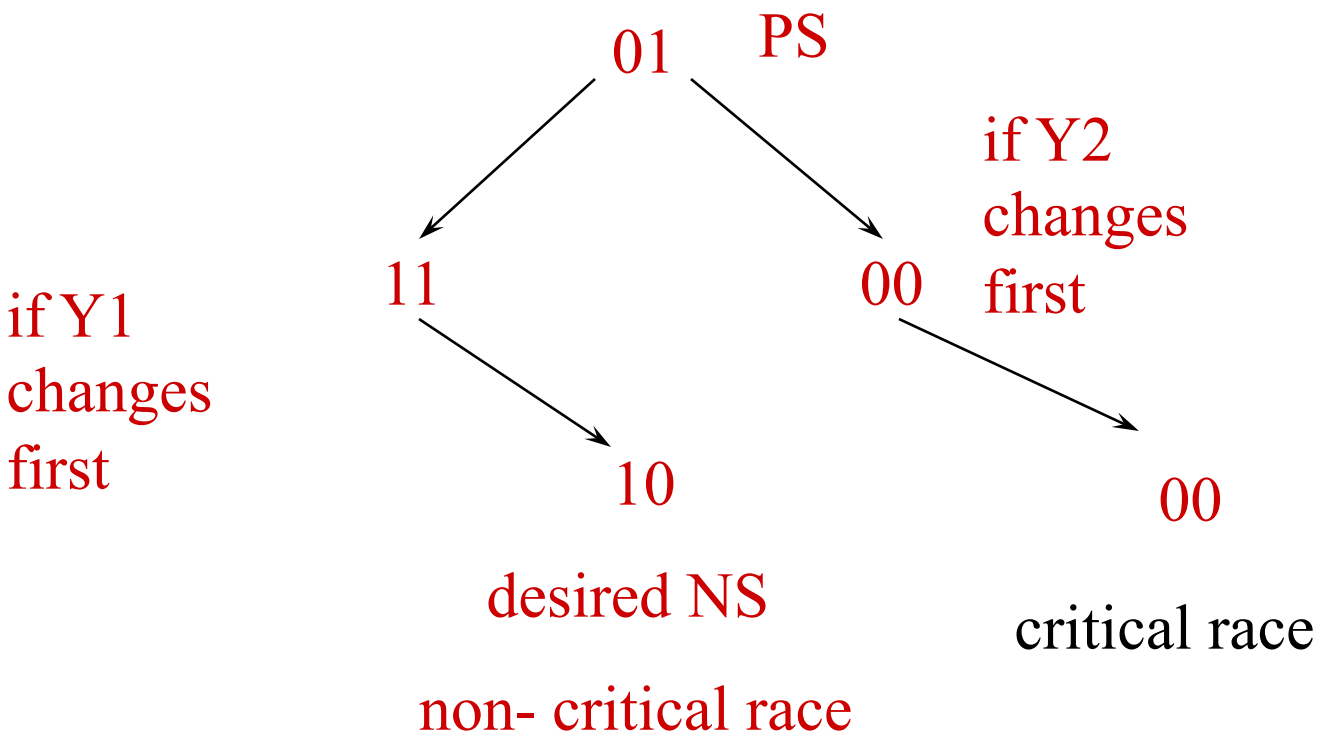
Races


PS
01
11
00
10
if Y1 changes first
if Y2 changes first
desired NS

Types of Races

- Non-Critical
  - Machine stabilizes in desired state, but may transition through other states on the way
- Critical
  - Machine does not stabilize in the desired state

Races


01
PS
if Y2 changes first
11
00
if Y1 changes first
10
00
desired NS
critical race
non- critical race

Types of Races

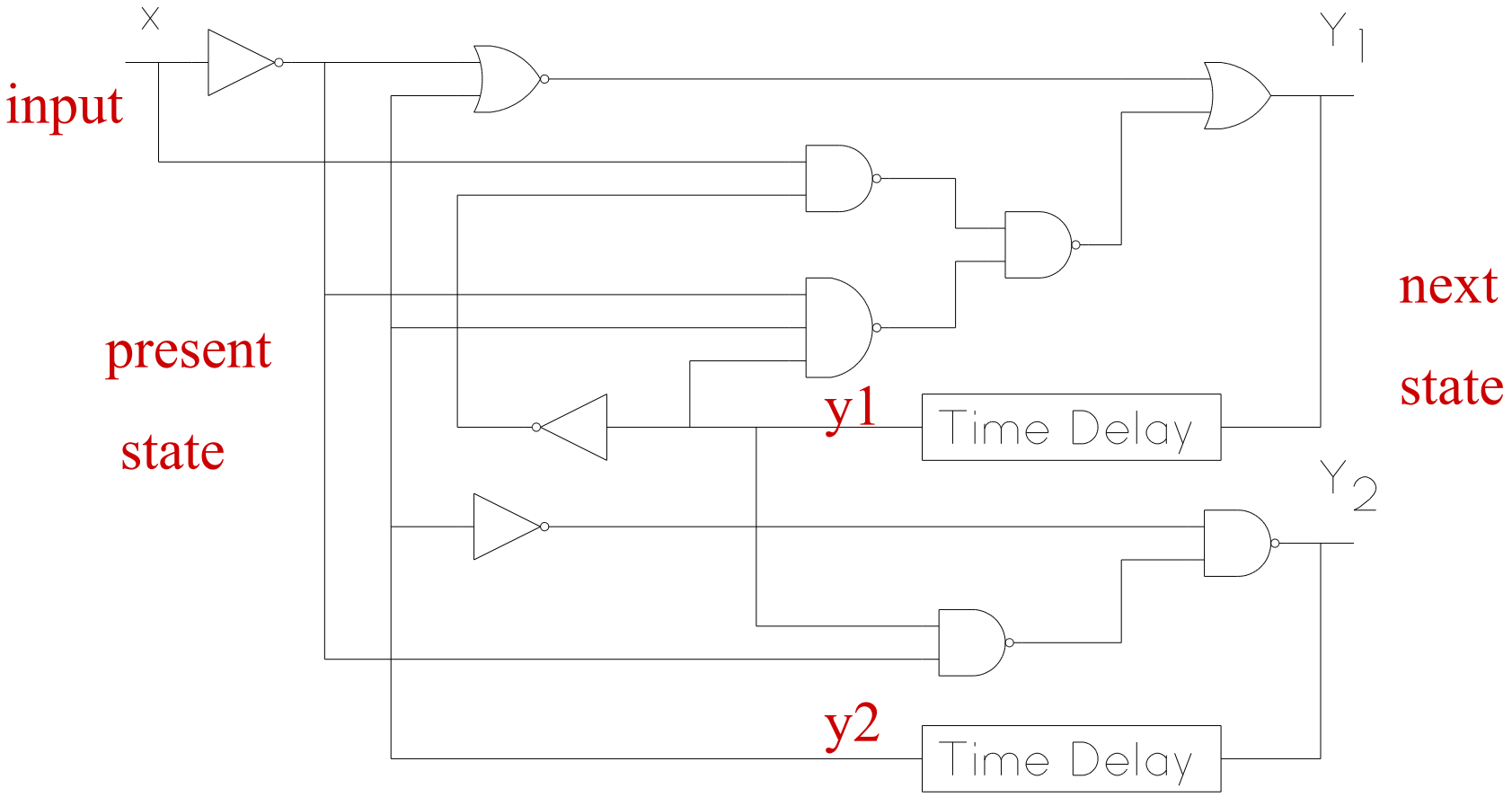
Asynchronous State machines have similarities to multi-threaded software as different elements are not coherently synchronized to a single time source.

We will revisit the concept of races and stability in the context of embedded software starting in Week 5

Asynchronous FSM Benefits

- Fastest FSM
- Economical
  - No need for clock generator
- Output Changes When Signals Change, Not When Clock Occurs
- Data Can Be Passed Between Two Circuits Which Are Not Synchronized
- In some technologies, like quantum, clock is just not possible to exist, no clocks in live organisms.

Asynchronous FSM example



Next State Variable

$$Y_1(x, y_1, y_2) = \overline{\left(\overline{x\ y_1}\right)\left(\overline{\overline{x}\ y_1\ y_2}\right)} + \overline{\overline{x} + y_2}$$

$$= x\ \overline{y_1} + \overline{x}\ y_1\ y_2 + x\ \overline{y_2}$$

$$Y_2(x, y_1, y_2) = \overline{\overline{y_2}\ \overline{x}\ y}$$

$$= y_2 + \overline{x}\ y_1$$

You should analyze this machine at home

Asynchronous State Tables

States are either Stable or Unstable.

Stable states encircled with ○ symbol.

Present state	Next state, output	
	x=0	x=1
Q_0	(○ Q_0,0)	Q_1,0
Q_1	Q_2,0	(Q_1,0)
Q_2	(Q_2,0)	Q_3,1
Q_3	Q_0, 0	(Q_3,1)

Oscillations occur if all states are unstable for an input value.

Total State is a pair (x, Q_i)