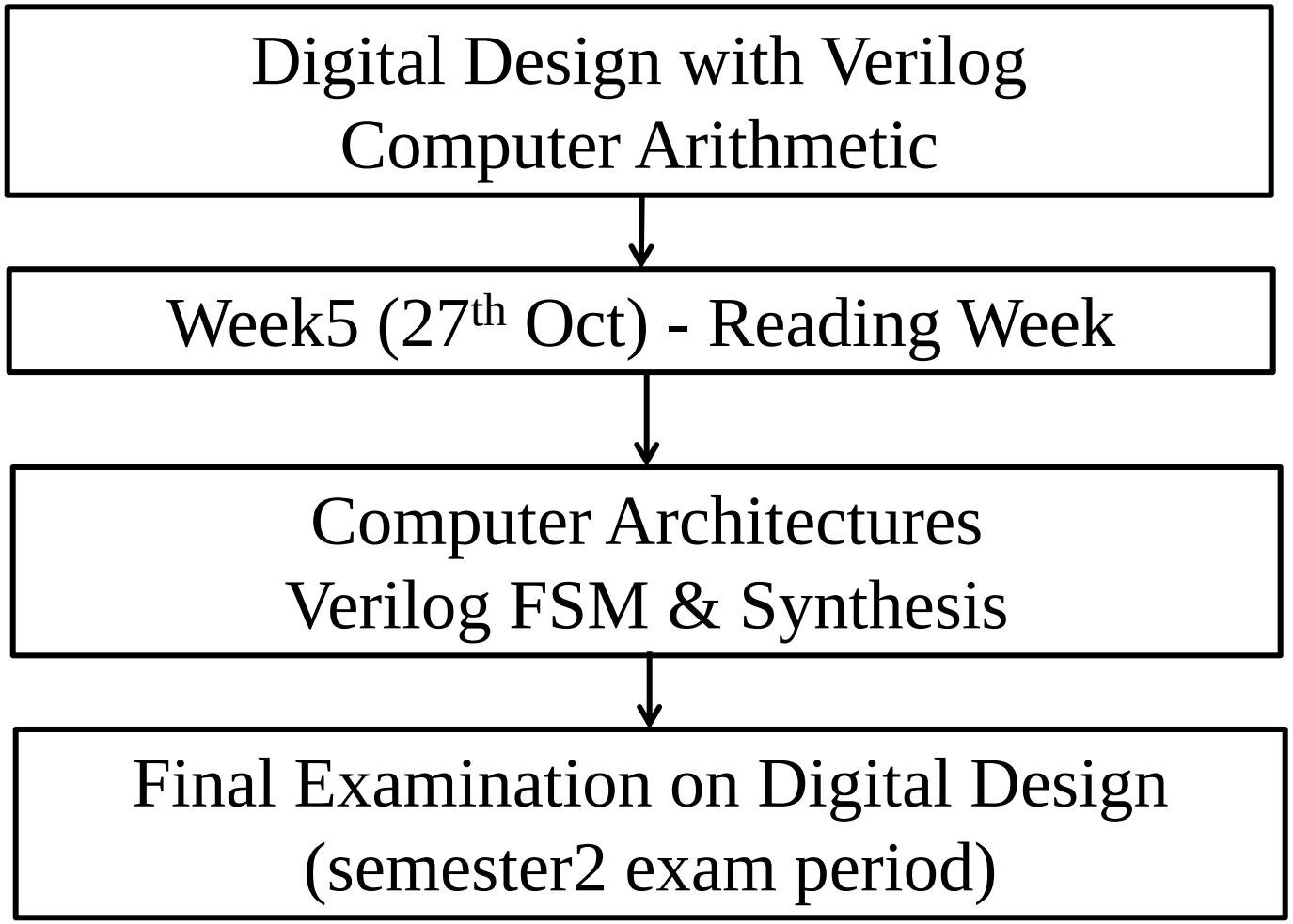
EEE336 - Digital Design

N J Powell

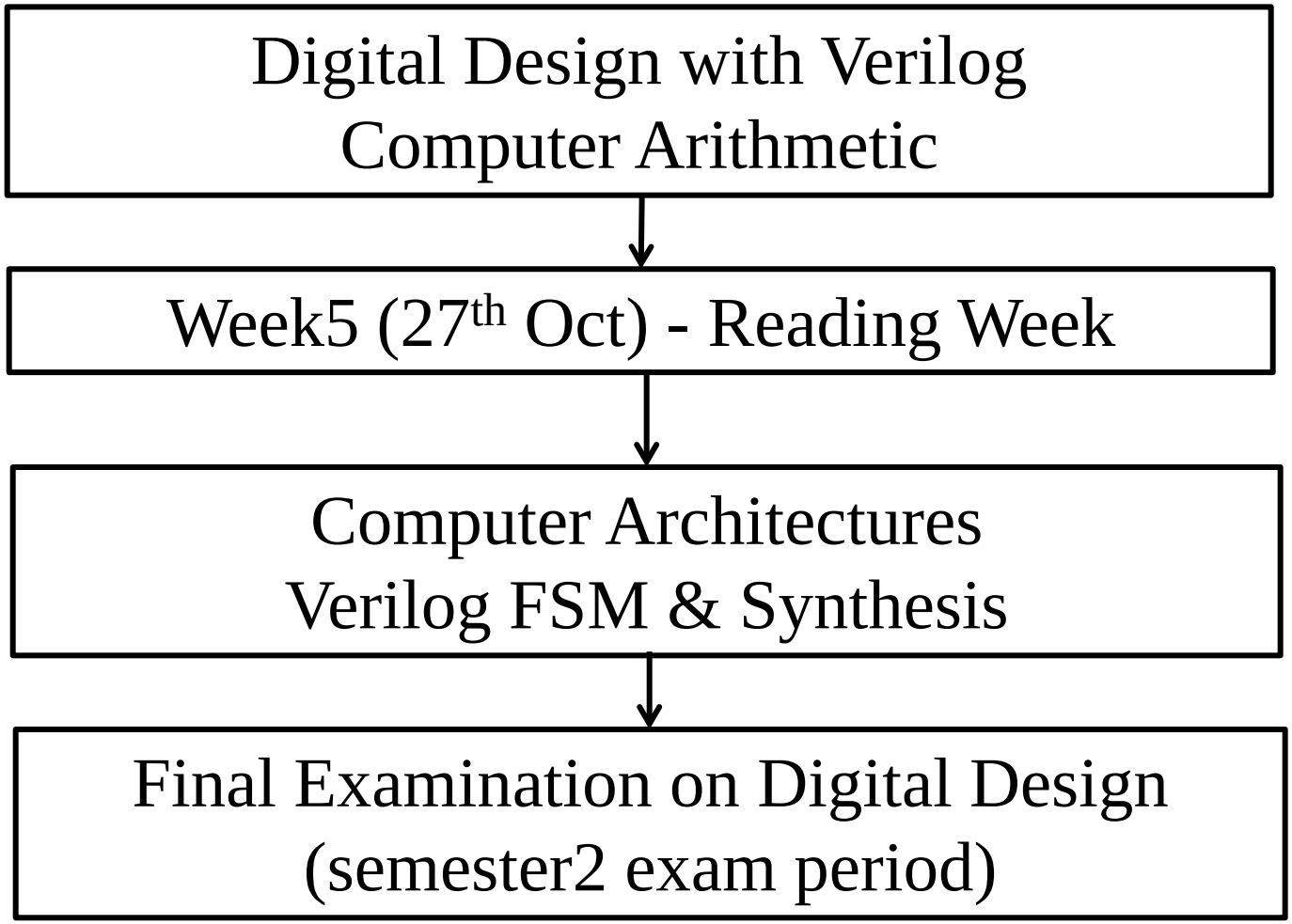
email: N.Powell@shef.ac.uk

room: F141

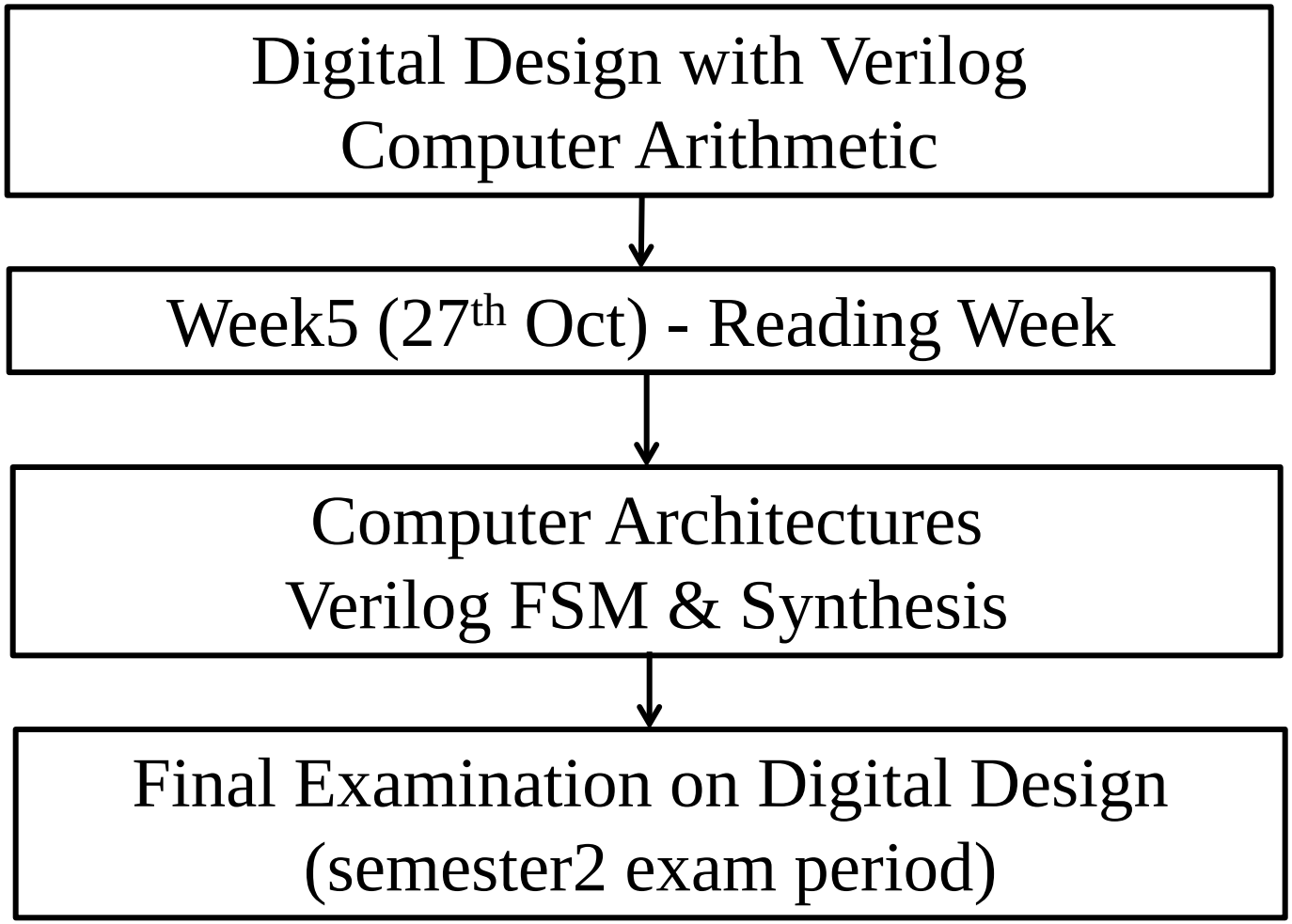
Course Structure



Course Structure



Course Structure



Course Aims

- To introduce the Verilog Hardware Description Language (HDL) and gain a working knowledge of its application to the modeling, simulation and synthesis of digital systems.
- To obtain detailed knowledge of the architecture and organisation of a basic microprocessor and its peripheral interfaces.
- To develop a solid understanding of synchronous digital systems with an awareness of performance limitations and implementation issues.

Course Outline

- **Verilog HDL:** syntax, simulation, synthesis, digital building blocks, finite state machines.
- **Operations:** arithmetic, floating point representation.
- **Microprocessors:** Instruction Set Architecture (ISA), internal organisation, addressing modes, stacks, interrupts.
- **Peripherals:** memory hierarchy, memory cycles, Direct Memory Access (DMA).
- **Processor Architecture:** ALU, registers, data path design, control, instruction cycle, RISC, CISC.
- **Microcontrollers & DSP:** Architecture, practical examples.

Verilog (I)

- A text-based way to describe electronic circuits
- C-like syntax
- Standard : IEEE 1364-1995, 2001, 2005)
- Originally a simulation language, now used extensively for synthesis

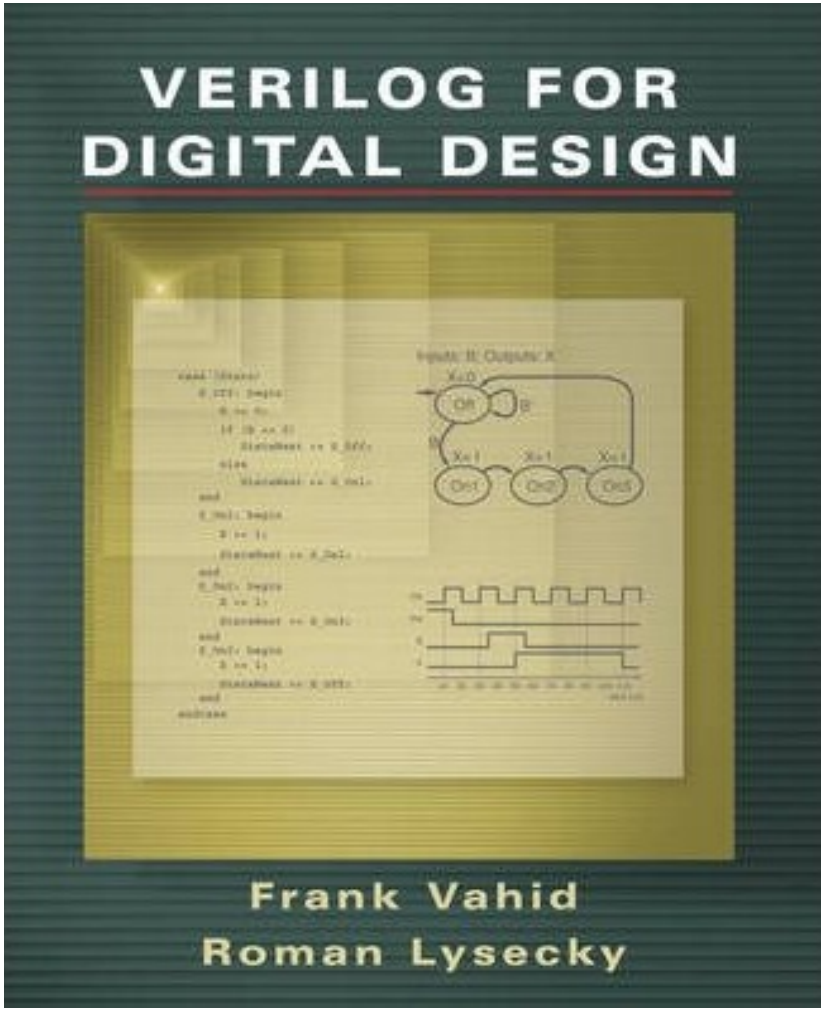
Other HDLs – VHDL, SystemC, SystemVerilog

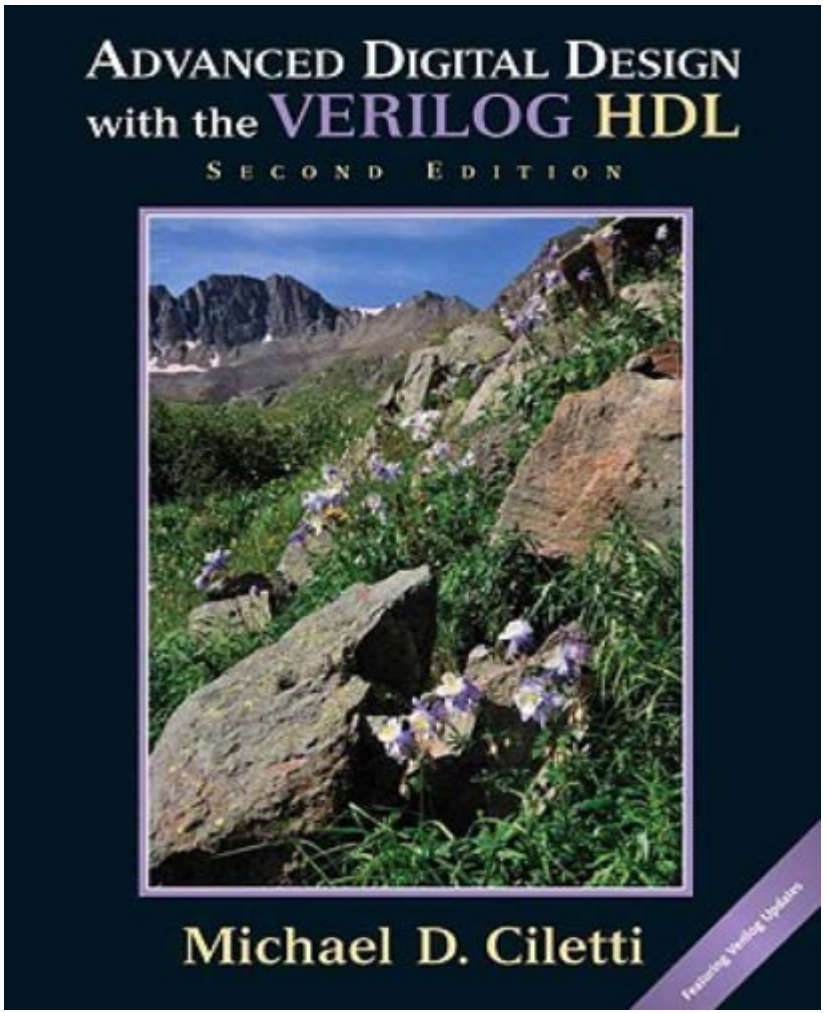
You are only required to be able to understand a Verilog model, you will not have to write a Verilog model in the exam.

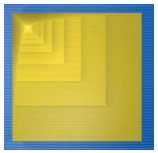
Useful Books

- Vahid — Verilog For Digital Design / Digital Design
- Ciletti — Advanced Digital Design with Verilog
- Mano & Kime — Logic and Computer Design Fundamentals
- Floyd — Digital Fundamentals
- Gajski — Principles of Digital Design
- Katz — Contemporary Logic Design

Course Books







Verilog Models

Structural Model: netlist of gates, partitioning of circuit into major functional blocks such as ALU (arithmetic and logic unit).

Logic gate primitives including the basic logic gates:

and, **nand**, **or**, **nor**, **xor**, **xnor, buf, not**

There are also tristate buffers and inverters:

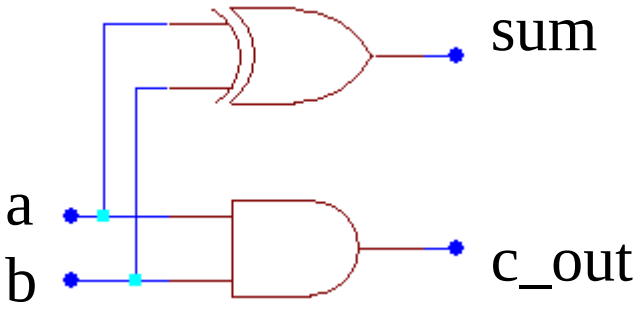
bufif0, bufif1, notif0, notif1

Functionality is obtained from internal truth tables. User Defined Primitives can also be created.

Behavioral Model: Boolean equations, register transfer level model (RTL), algorithms.

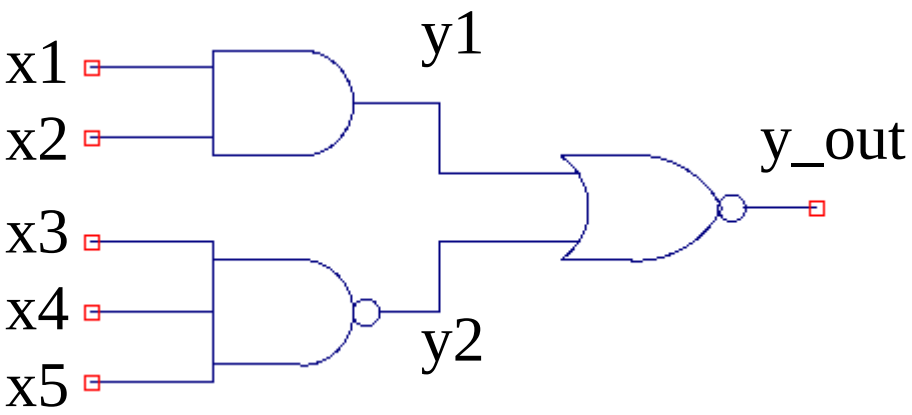
Structural Models

half adder




```
module Add_half(output c_out, sum, input a, b);
   xor (sum, a, b);
   and (c_out, a, b);
endmodule
```

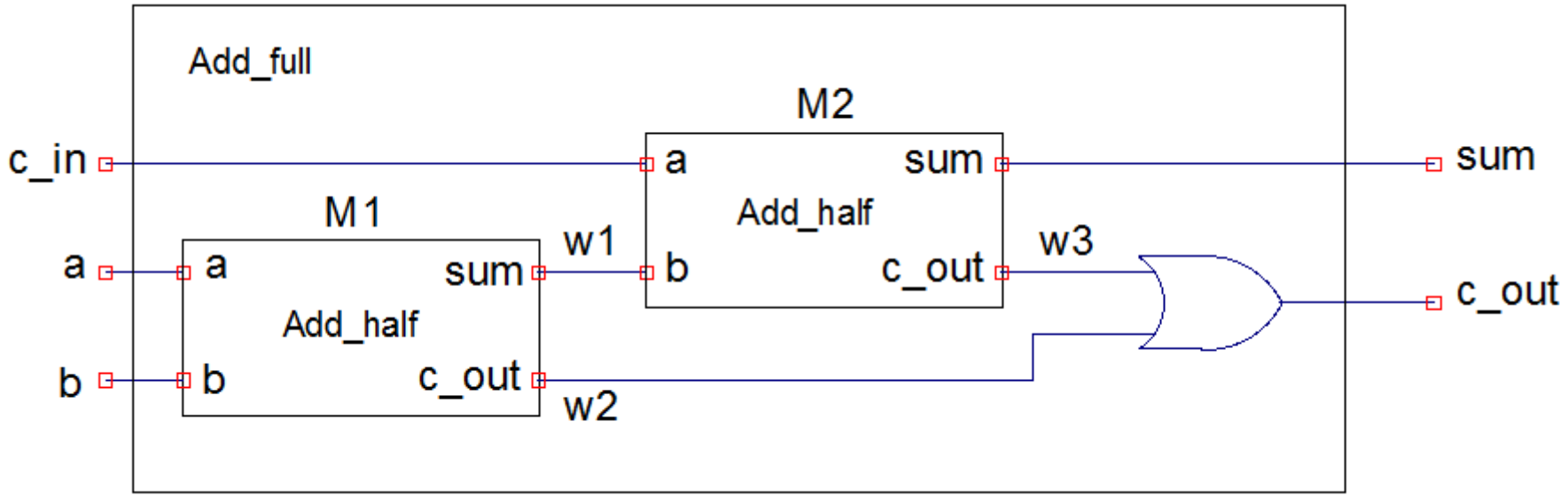
The output port of a primitive must be first in the list of ports.
Multiple inputs can be accommodated.

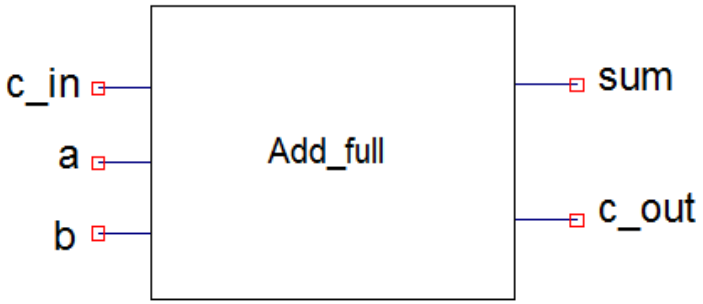



```
wire  y1, y2;
nor   (y_out, y1, y2);
and   (y1, x1, x2);
nand  (y2, x3, x4, x5);
```

Design Hierarchy

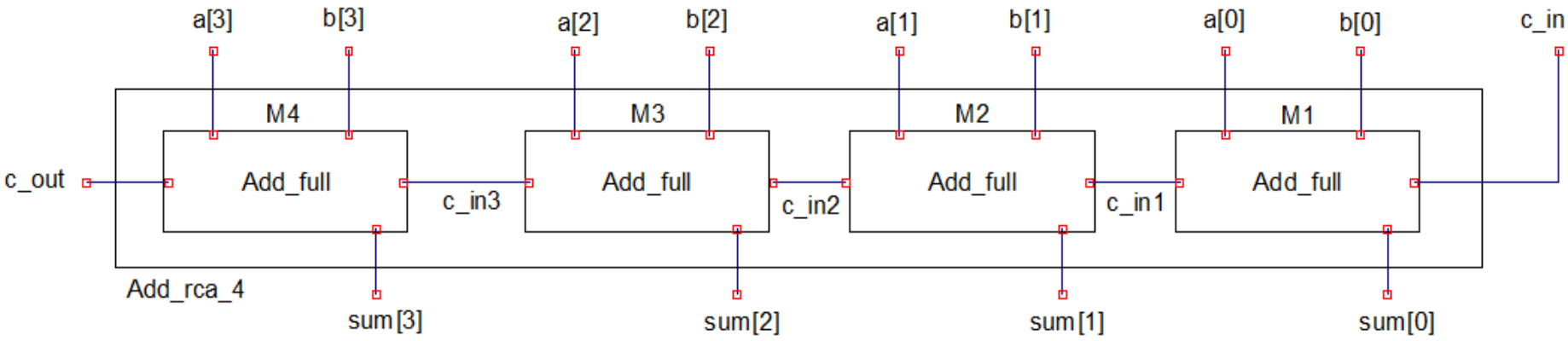
Verilog supports hierarchical structural models. A full adder can be created using two '**instances**' of a half adder.






```
module Add_full(output c_out, sum, input a, b, c_in);
   wire w1, w2, w3;
   Add_half M1 (w2, w1, a, b);
   Add_half M2 (w3, sum, c_in, w1);
   or (c_out, w3, w2);
endmodule
```

A ripple carry adder can be formed by connecting 4 full adders in a chain (Add_rca_4).




```
module Add_rca_4(output c_out, output[3:0] sum, input [3:0] a, b, input c_in);
  wire c_in1, c_in2, c_in3;
  Add_full  M1  (c_in1,  sum[0],  a[0], b[0], c_in);
  Add_full  M2  (c_in2,  sum[1],  a[1], b[1], c_in1);
  Add_full  M3  (c_in3,  sum[2],  a[2], b[2], c_in2);
  Add_full  M4  (c_out,  sum[3],  a[3], b[3], c_in3);
endmodule
```

A 16-bit adder could be formed by instantiating four 4-bit adders etc.

Port Mapping

The ports in the instantiated module have been connected by the order in which they were declared in the module port declaration.

When the number of ports grows larger, it is easier to associate ports by their names.

.formal_name(actual_name)

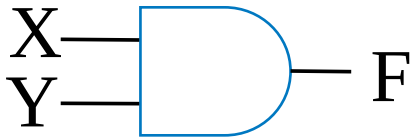
name given in the declaration of the instantiated module

name used in the instantiation of the module

e.g.

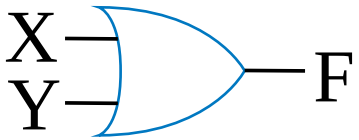
```
Add_half M1 (.b(b),
             .c_out(w2),
             .a(a),
             .sum(w1)
             );
```

Behavioral Models



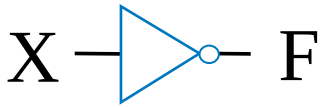
module And2(X, Y, F);

input X, Y;
output F;
...



module Or2(X, Y, F);

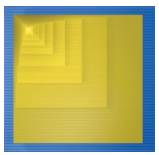
input X, Y;
output F;
...



module Inv(X, F);

input X;
output F;
...

- module – Declares a new type of component
  - Named “And2" in first example above
  - Includes list of ports (module's inputs and outputs)
- input – List indicating which ports are inputs
- output – List indicating which ports are outputs



Verilog Data Types

nets : establish connectivity – like the wires in a circuit

registers : hold information – like the variables in a program

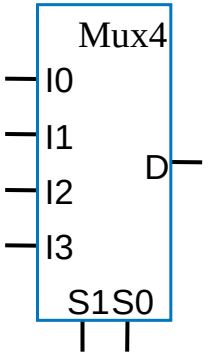
net	register
wire	reg
tri	integer
supply0	real
supply1	time
wand	
wor	

wires are used to connect design objects. They do not store logic values but take a value from the circuit driving them. They are the default type for ports.

Q: Begin a module definition for a 4x1 multiplexor

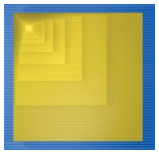
Inputs: I3, I2, I1, I0, S1, S0. Outputs: D

4x1 mux

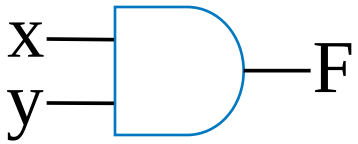



```
module Mux4(I3, I2, I1, I0, S1, S0, D);

  input I3, I2, I1, I0;
  input S1, S0;
  output D;
  ...
```

Note that input ports above are separated into two declarations for clarity



Module Procedures—always




```
module And2(X, Y, F);

   input X, Y;
   output F;
   reg F;

   always @(X, Y) begin
      F <= X & Y;
   end

endmodule
```

wait until X or Y changes

F <= x AND y

always – Procedure that executes repetitively (infinite loop) from simulation start

@ – event control indicating that statements should only execute when values change

"(X,Y)" – execute if X changes or Y changes (change known as an event)

Sometimes called "*sensitivity list*"

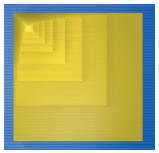
We'll say that procedure is "*sensitive to X and Y*"

"F <= X & Y;" – Procedural statement that sets F to AND of X, Y

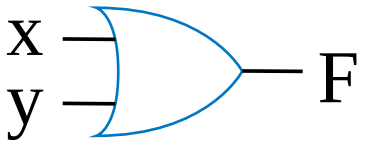
& is built-in bit AND operator

<= assigns value to variable

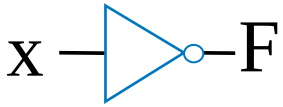
reg – Declares a variable data type, which holds its value between assignments



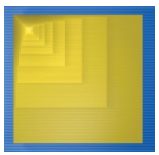
- Q: Given that "|" and "~" are built-in operators for OR and NOT respectively, complete the behavioral modules for a 2-input OR gate and a NOT gate.




```
module Or2(X, Y, F);

 input X, Y;
 output F;

reg F;

always @(X, Y) begin
   F <= X | Y;
end

endmodule
```




```
module Inv(X, F);

 input X;
 output F;

   reg F;

always @(X) begin
   F <= ~X;
end

endmodule
```

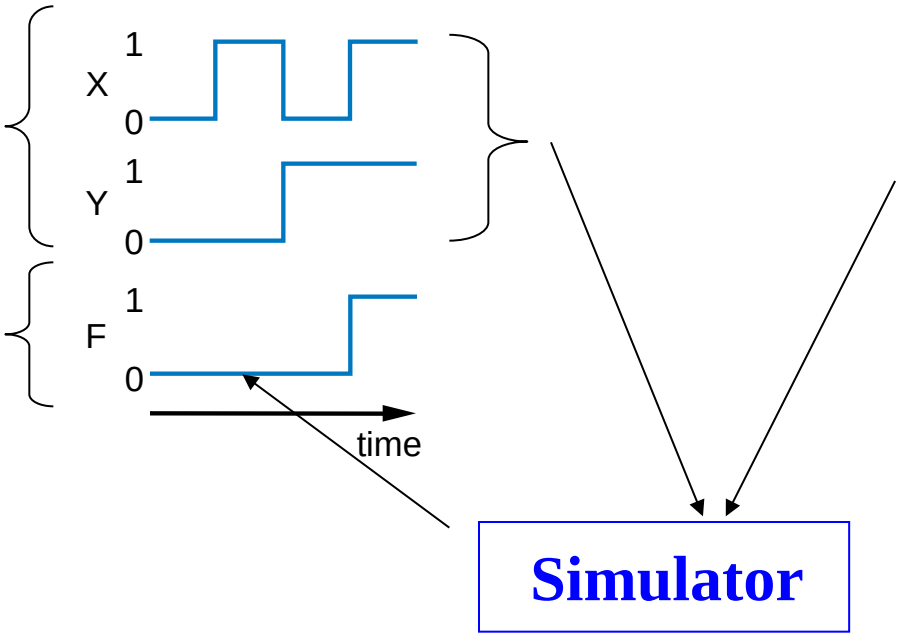


Simulation and Testbenches

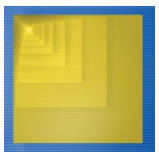
- How does our new module behave?
- Simulation
  - User provides input values, simulator generates output values
    - Test vectors – sequence of input values
    - Waveform – graphical depiction of sequence

User provides test vectors

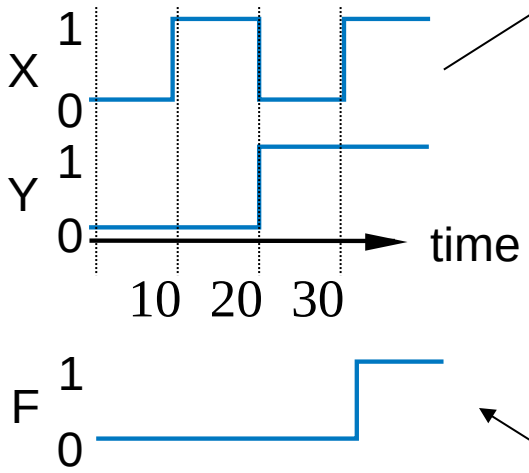
Simulator generates output values based on HDL description




```
module And2(X, Y, F);

  input X, Y;
  output F;
  reg F;

  always @(X, Y) begin
     F <= X & Y;
  end

endmodule
```



- Instead of drawing test vectors, user can describe them with Verilog

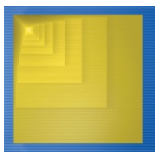



```
...
Y_s <= 0; X_s <= 0;
#10 Y_s <= 0; X_s <= 1;
#10 Y_s <= 1; X_s <= 0;
#10 Y_s <= 1; X_s <= 1;
...
```

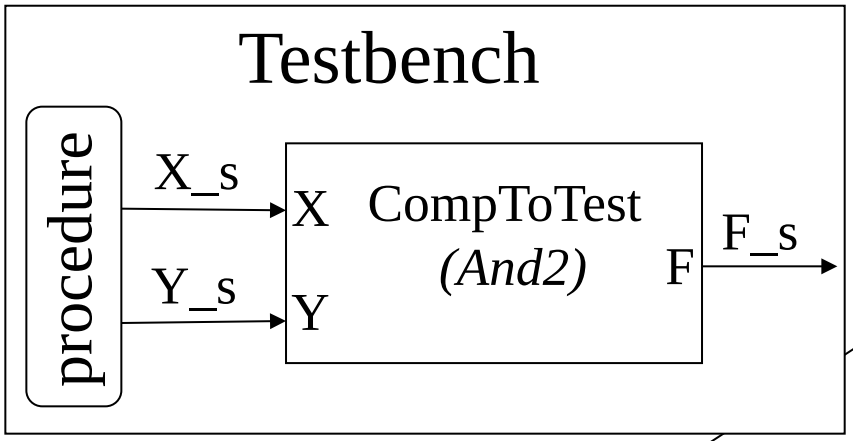
```
module And2(X, Y, F);

  input X, Y;
  output F;
  reg F;

  always @(X, Y) begin
     F <= X & Y;
  end

endmodule
```

"#10" tells simulator to keep present values for 10 time units, before executing the next statement

Simulator

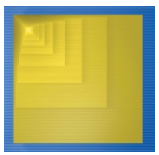


"Testbench" module that provides test vectors to component's inputs:



- Verilog testbench
  - Module with no ports
  - Declare reg variable for each input port, wire for each output port
  - Instantiate module, map variables to ports
  - Set variable values at desired times

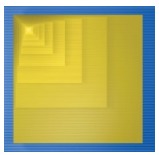
```
`timescale 1 ns/1 ns

module Testbench();

  reg X_s, Y_s;
  wire F_s;

  And2 CompToTest(X_s, Y_s, F_s);

  initial begin
    // Test all combinations
    Y_s <= 0; X_s <= 0;
    #10 Y_s <= 0; X_s <= 1;
    #10 Y_s <= 1; X_s <= 0;
    #10 Y_s <= 1; X_s <= 1;
  end

endmodule
```



- wire – Declares a net data type, which does not store its value in contrast to **reg** data type which stores its value
  - Nets used for connections
  - Net's value determined by what it is connected to

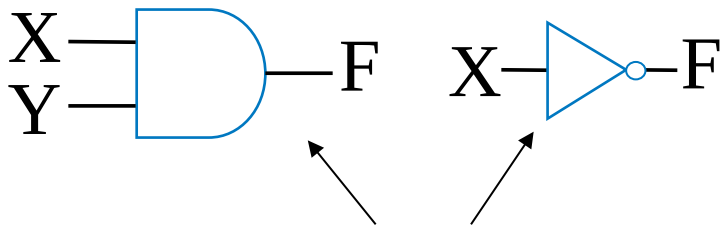
- initial –procedure that executes at simulation start, but *executes only once* in contrast to an "always" procedure that also executes at simulation start, but that *repeats*

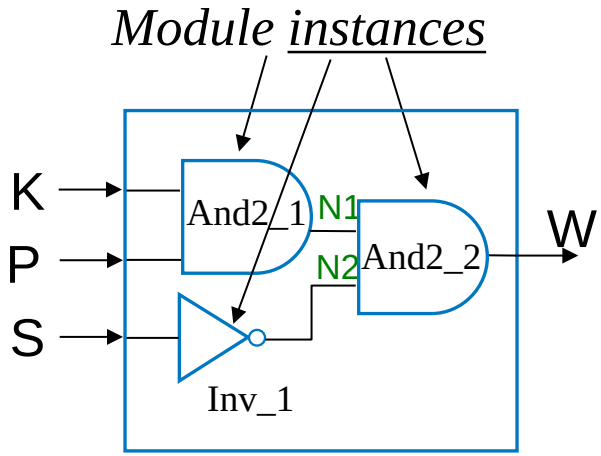
- # – Delay control: number of time units to delay this statement's execution relative to previous statement
  - `timescale – compiler directive telling compiler that from this point forward, 1 time unit means 1 ns
    - Valid time units – s (seconds), ms (milliseconds), us (microseconds), ns (nanoseconds), ps (picoseconds), and fs (femtoseconds)
    - 1 ns/1 ns – time unit / time precision. Precision is for internal rounding. For our purposes, precision will be set same as time unit.



Combinational Circuits

- Circuit – A connection of modules
  - Also known as structure,
    - vs. using an always procedure, as earlier
- Instance – An occurrence of a module in a circuit
    - May be multiple instances of a module
    - e.g., Car's modules: tyres, engine, windows, etc., with 4 tyre instances, 1 engine instance, 6 window instances.

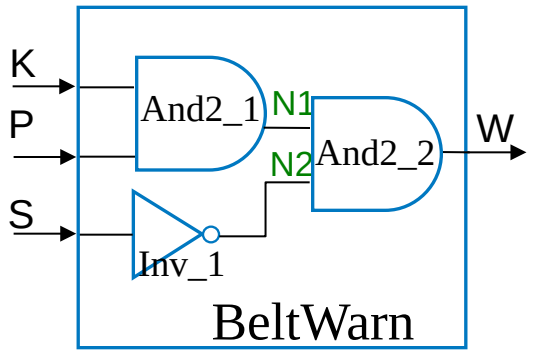




- Creating a circuit
  1. Start definition of a new module
  2. Declare nets for connecting module instances
     - N1, N2
     - Note: W is also a declared as a net. By defaults outputs are considered wire nets unless explicitly declared as a reg variable
  3. Create module instances, create connections

"BeltWarn" example: Turn on warning light (w=1) if car key is in ignition (k=1), person is seated (p=1), and seatbelt is not fastened (s=0)

```
module BeltWarn(K, P, S, W);

  input K, P, S;
  output W;

  wire N1, N2;

  And2 And2_1(K, P, N1);
  Inv  Inv_1(S, N2);
  And2 And2_2(N1, N2, W);

endmodule
```



- Module instantiation statement

```
And2 And2_1(K, P, N1);
```

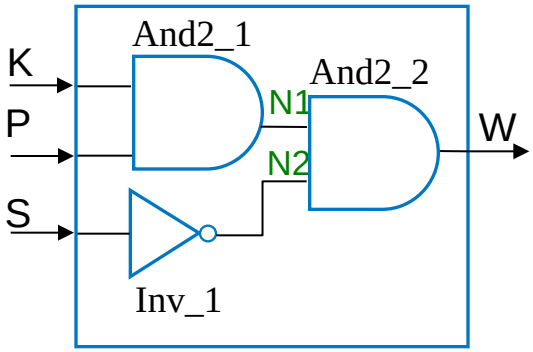
Connects instantiated module's ports to nets and variables

Name of new module instance
Must be distinct; hence And2_***1*** and And2_***2***

Name of module to instantiate

Note: Ports ordered as in original And2 module definition

```
module BeltWarn(K, P, S, W);

  input K, P, S;
  output W;

  wire N1, N2;

  And2 And2_1(K, P, N1);
  Inv  Inv_1(S, N2);
  And2 And2_2(N1, N2, W);

endmodule
```

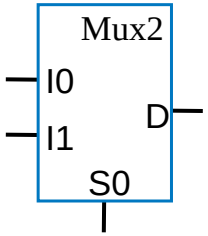


BeltWarn

- Q: Complete the 2x1 mux circuit's module instantiations

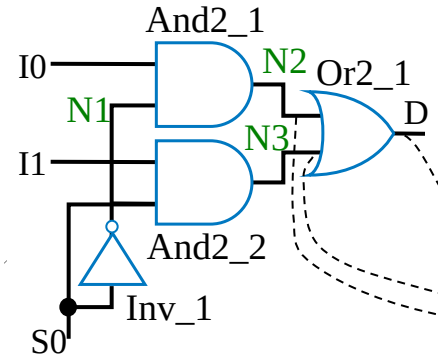
1. Start definition of a new module (done)

(Draw desired circuit, if not already done)

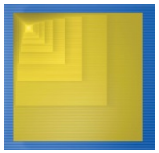


2. Declare nets for internal wires

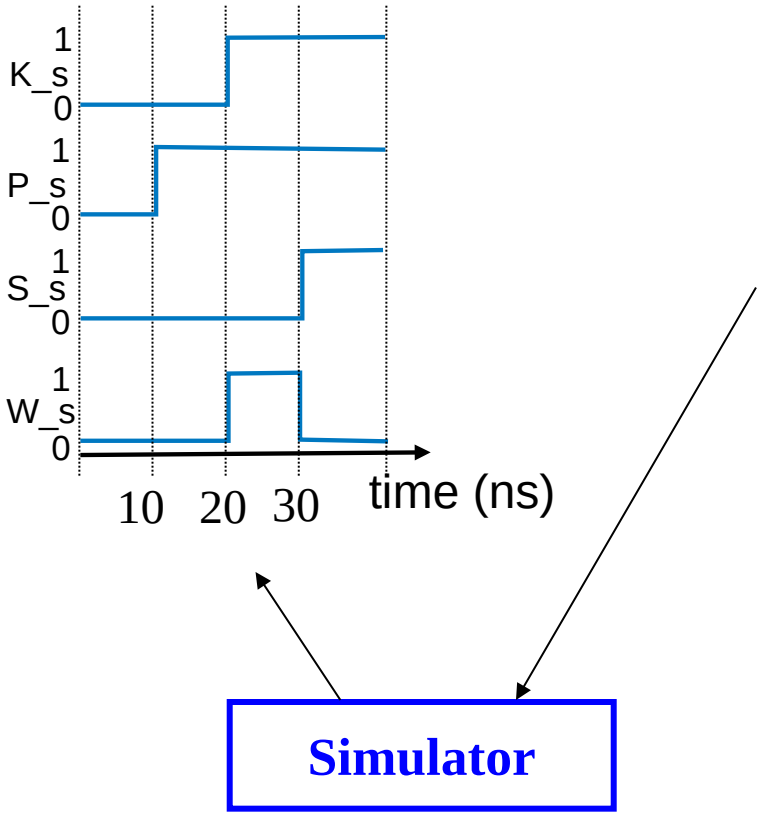
3. Create module instances and connect ports



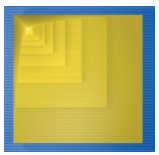

```
module Mux2(I1, I0, S0, D);

   input I1, I0;
   input S0;
   output D;

   wire N1, N2, N3;

   Inv  Inv_1 (S0, N1);
   And2 And2_1(I0, N1, N2);
   And2 And2_2(I1, S0, N3);
   Or2  Or2_1  N2, N3, D);

endmodule
```



- Simulate testbench file to obtain waveforms




```
`timescale 1 ns/1 ns

module Testbench();

  reg K_s, P_s, S_s;
  wire W_s;

  BeltWarn CompToTest(K_s, P_s, S_s, W_s);

  initial begin
    K_s <= 0; P_s <= 0; S_s <= 0;
    #10 K_s <= 0; P_s <= 1; S_s <= 0;
    #10 K_s <= 1; P_s <= 1; S_s <= 0;
    #10 K_s <= 1; P_s <= 1; S_s <= 1;
  end

endmodule
```



- More on testbenches
  - Note that a single module instantiation statement used
  - reg and wire declarations (K_s, P_s, S_s, W_s) used because procedure cannot access instantiated module's ports directly
    - Inputs declared as regs so can assign values (which are held between assignments)
  - Note module instantiation statement and procedure can both appear in one module