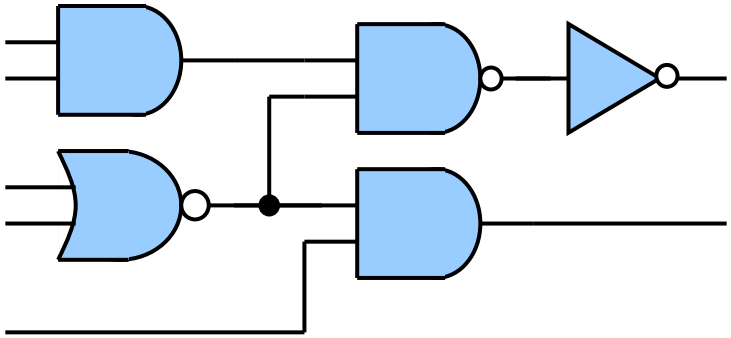
Lecture 2: HDL

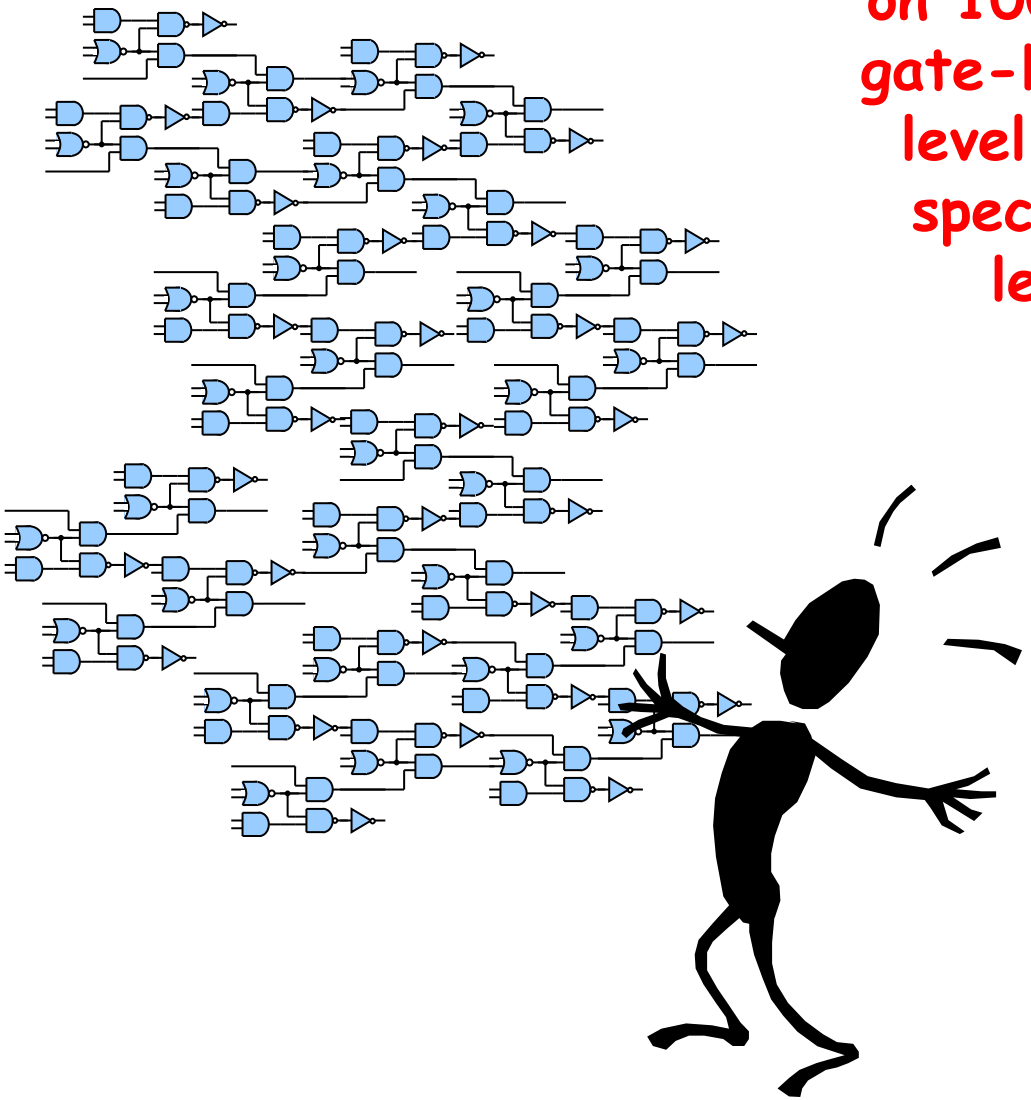
Mostafizur Rahman
rahmanmo@umkc.edu

Hardware Description Languages



In the beginning designs involved just a few gates, and thus it was possible to verify these circuits on paper or with breadboards

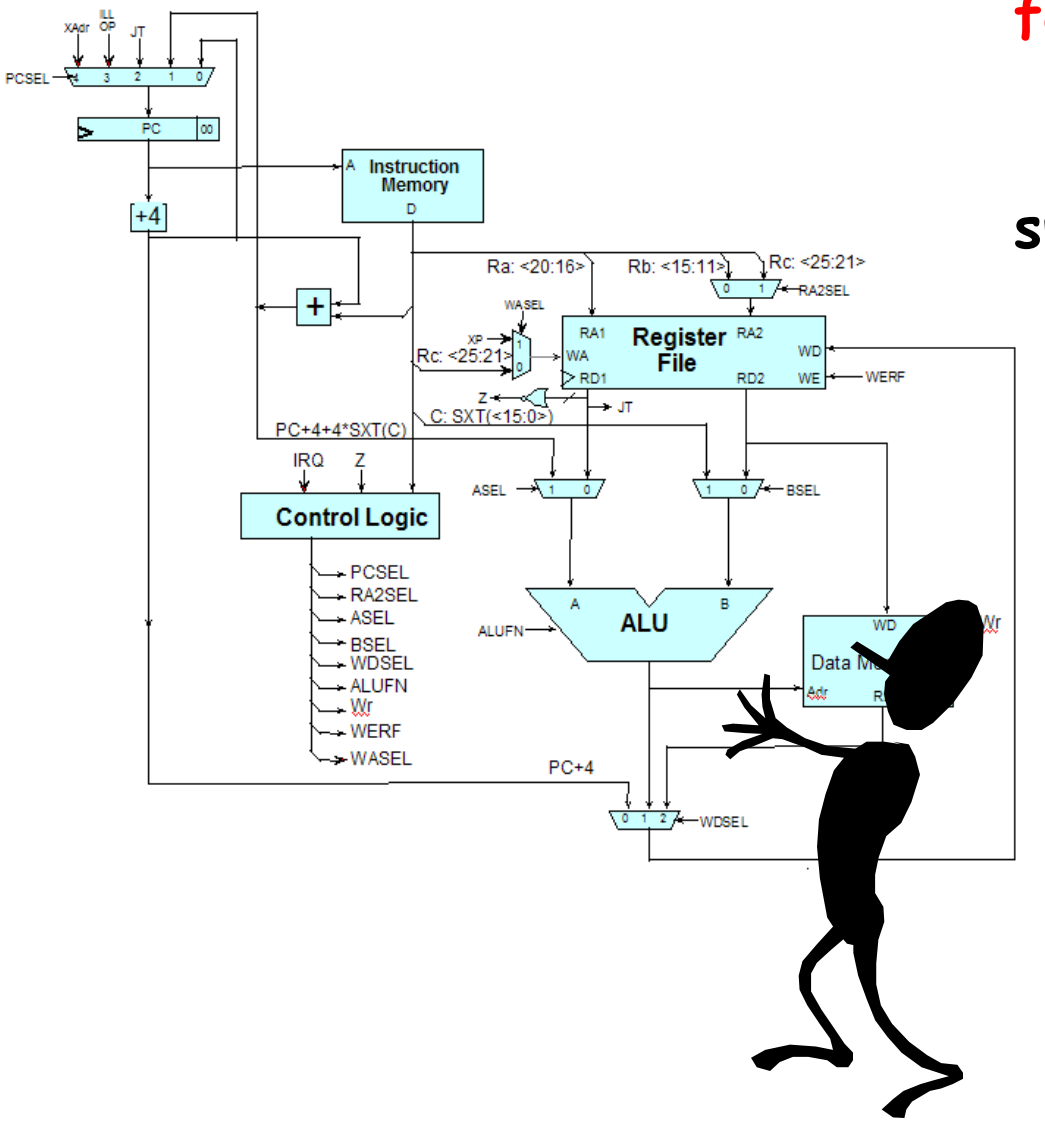
Hardware Description Languages



When designers began working on 100,000 gate designs, these gate-level models were too low-level for the initial functional specification and early high-level design exploration

What challenges would be there other then design time?

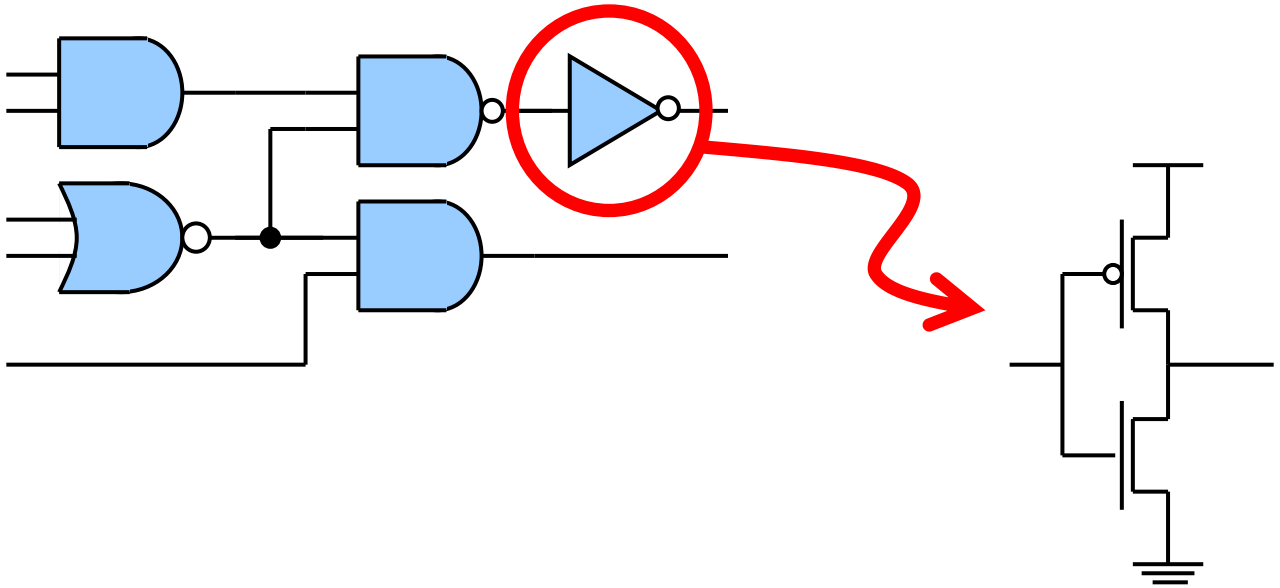
Hardware Description Languages



Designers again turned to HDLs for help – abstract behavioral models written in an HDL provided both a precise specification and a framework for design exploration

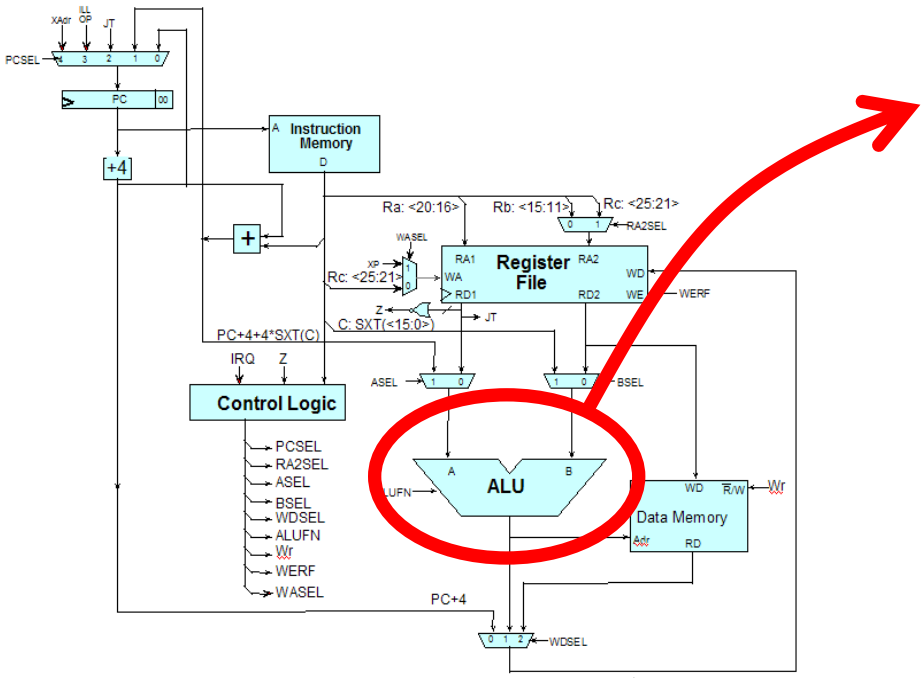
Advantages of HDLs

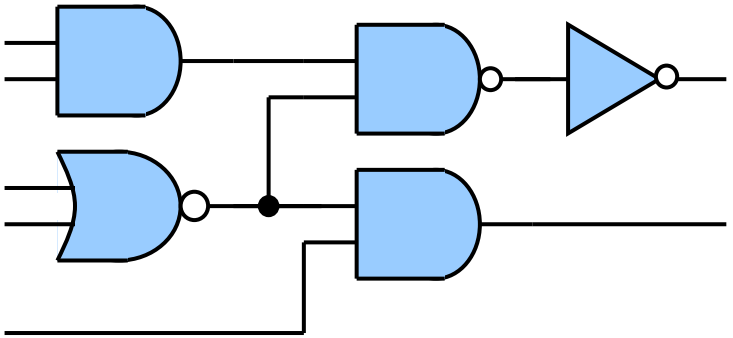
- Allows designers to talk about what the hardware should do without actually designing the hardware itself, or in other words HDLs allow designers to separate behavior from implementation at various levels of abstraction



Advantages of HDLs

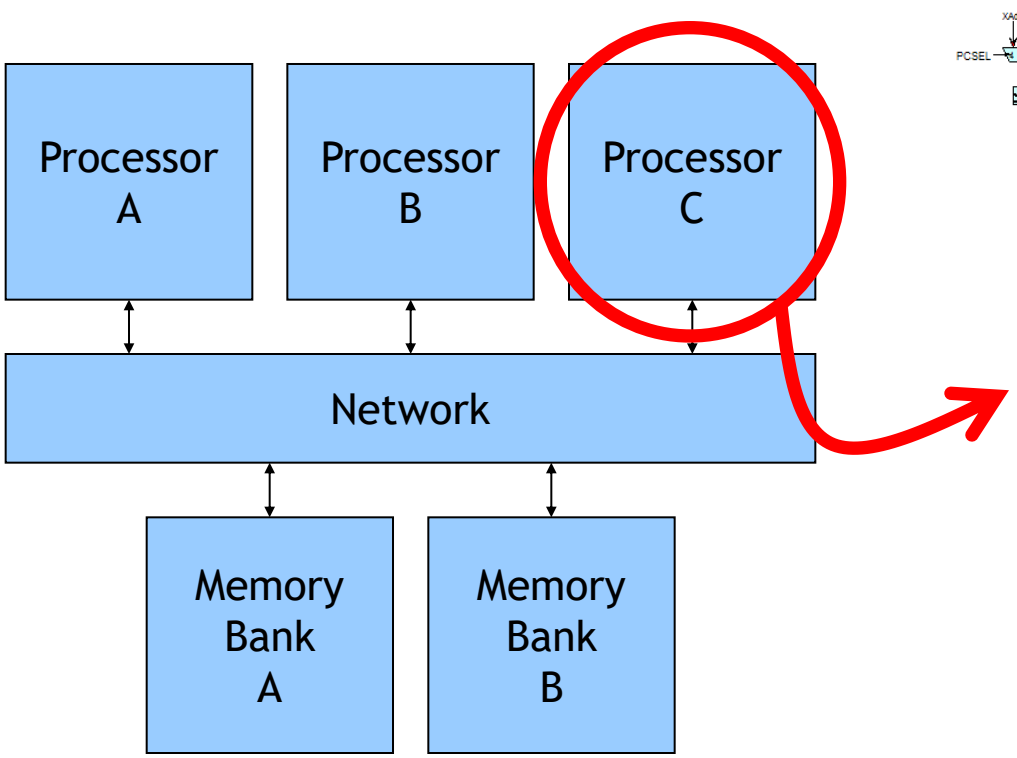
- Allows designers to talk about what the hardware should do without actually designing the hardware itself, or in other words HDLs allow designers to separate behavior from implementation at various levels of abstraction

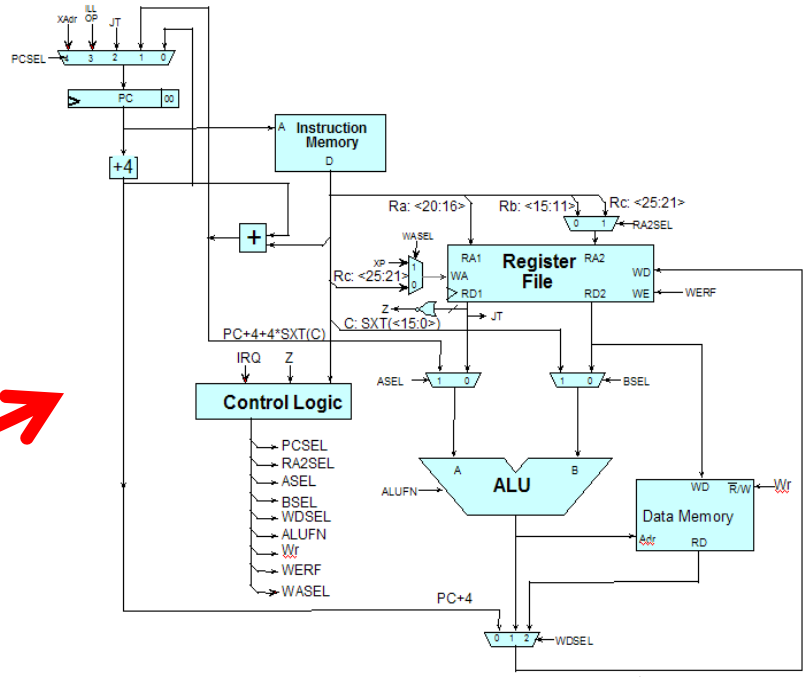




Advantages of HDLs

- Allows designers to talk about what the hardware should do without actually designing the hardware itself, or in other words HDLs allow designers to separate behavior from implementation at various levels of abstraction

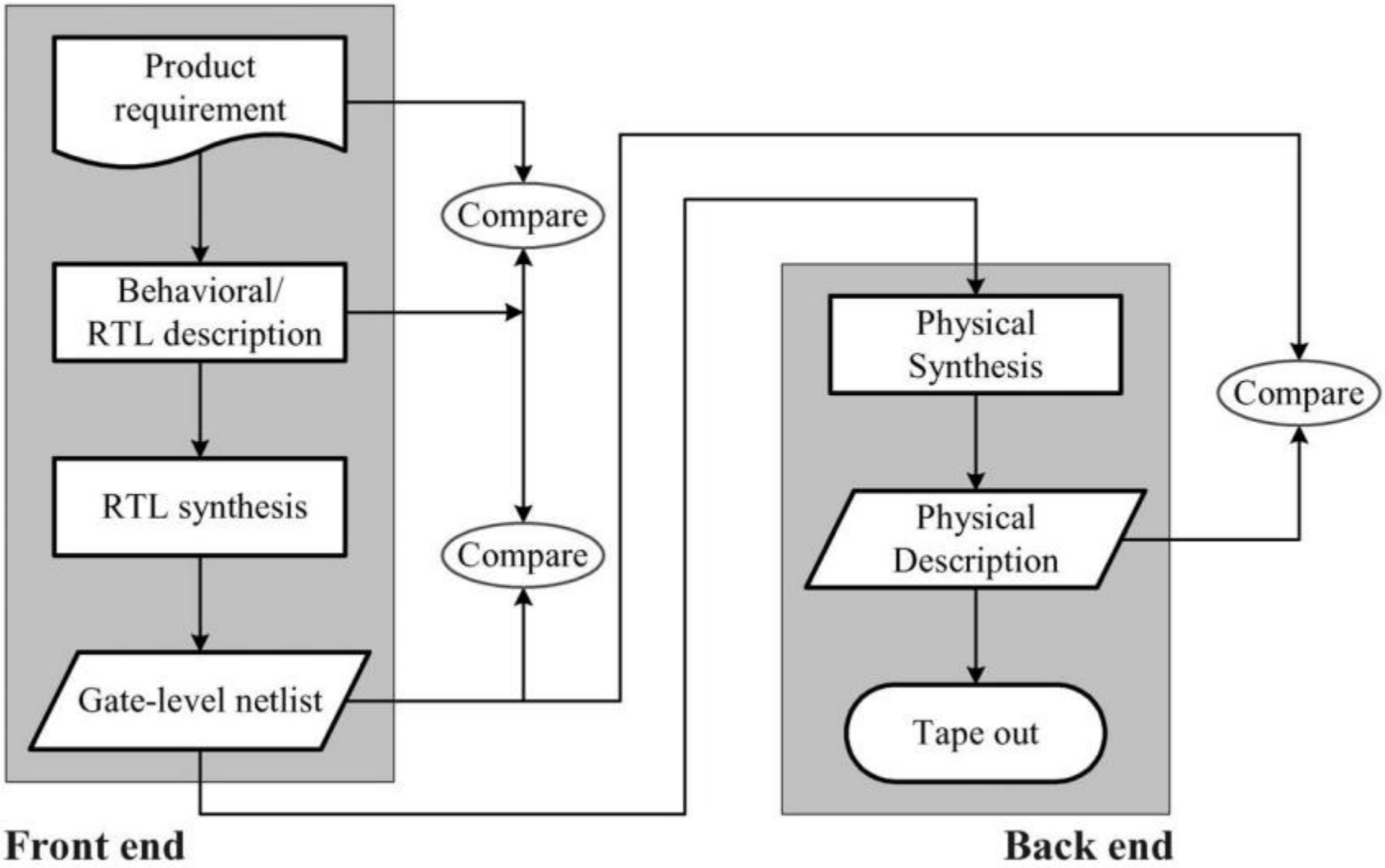




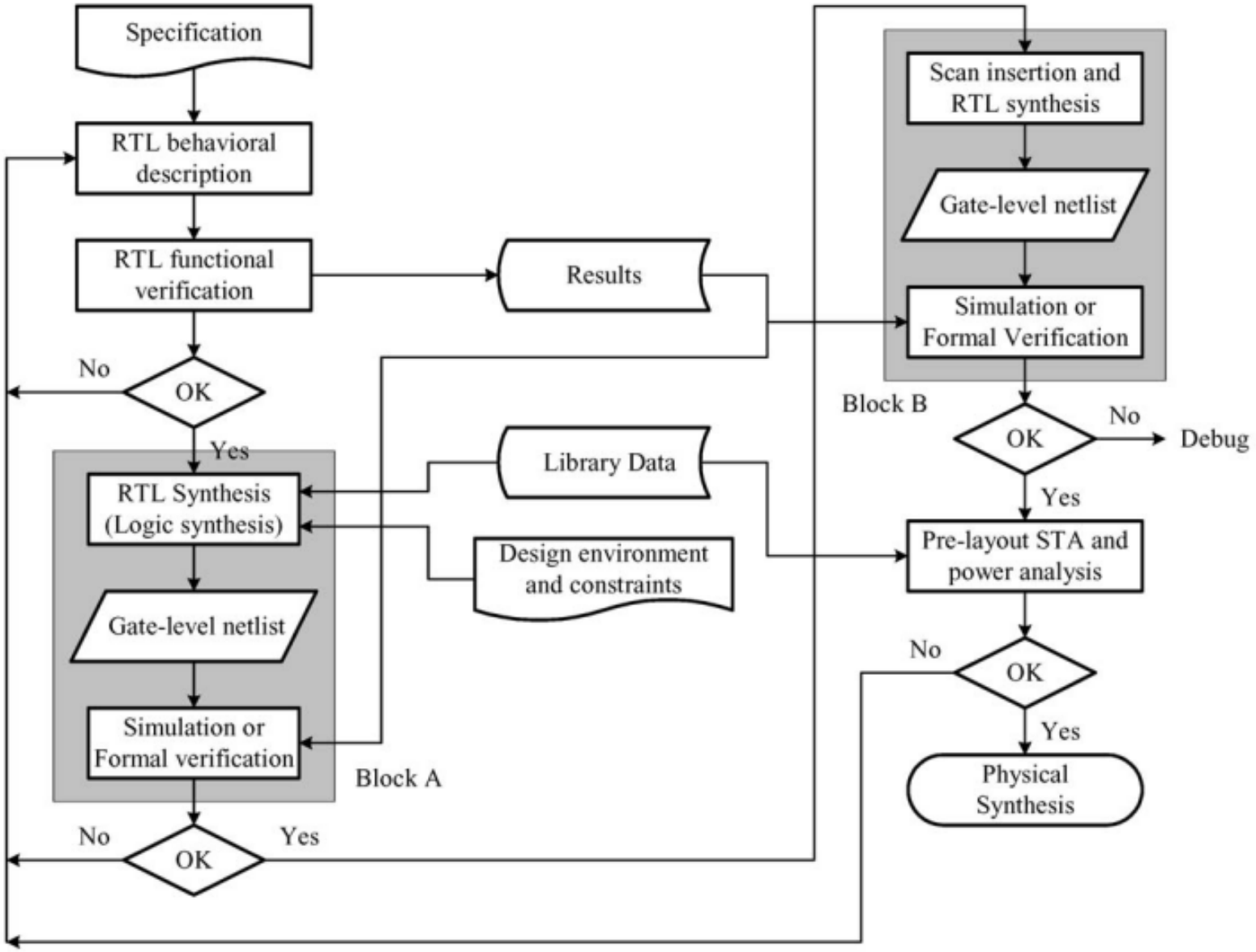
Advantages of HDLs

- **Allows designers to talk about what the hardware should do without actually designing the hardware itself, or in other words HDLs allow designers to separate behavior from implementation at various levels of abstraction**
  - Designers can develop an executable functional specification that documents the exact behavior of all the components and their interfaces
  - Designers can make decisions about cost, performance, power, and area earlier in the design process
  - Designers can create tools which automatically manipulate the design for verification, synthesis, optimization, etc.

ASIC Design Flow



RTL Synthesis Flow



HDLs

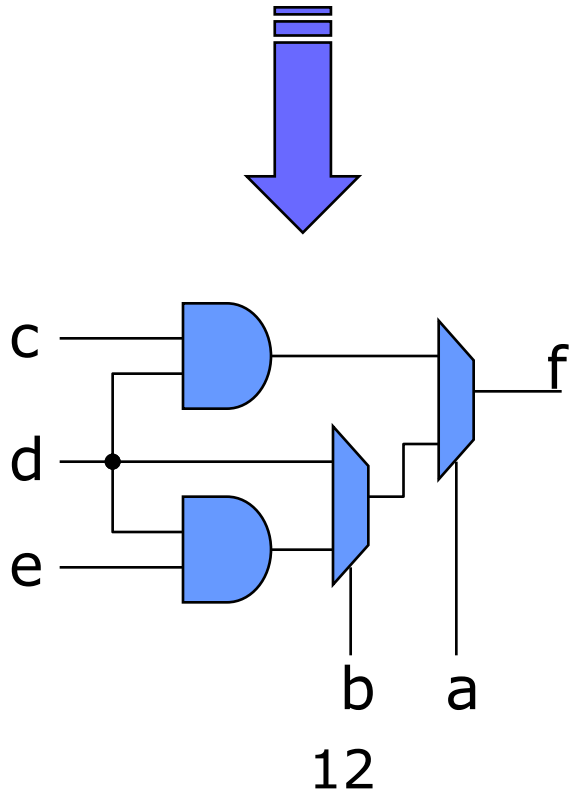
- A Hardware Description Language (HDL) is a language used to describe a digital system, for example, a computer or a component of a computer.
  - Textual descriptions of digital logic
  - Description languages, not Programming languages
  - Allow modeling and simulating the functional behavior and timing of digital hardware
  - Synthesis tools take an HDL description and generate a technology-specific netlist (real hardware representation)
- Two main HDLs used by industry
  - Verilog (C-based, industry-driven)
  - VHDL (Ada-based, defense/industry/university-driven)

- Describe hardware using code
  - Document logic functions
  - Simulate logic before building
  - Synthesize code into gates and layout

Describing Hardware, not Software!

- Hardware is created during synthesis
  - Even if `a` is true, still performs `d&e`
  - HDLs are inherently parallel
- Learn to understand how descriptions translated to hardware

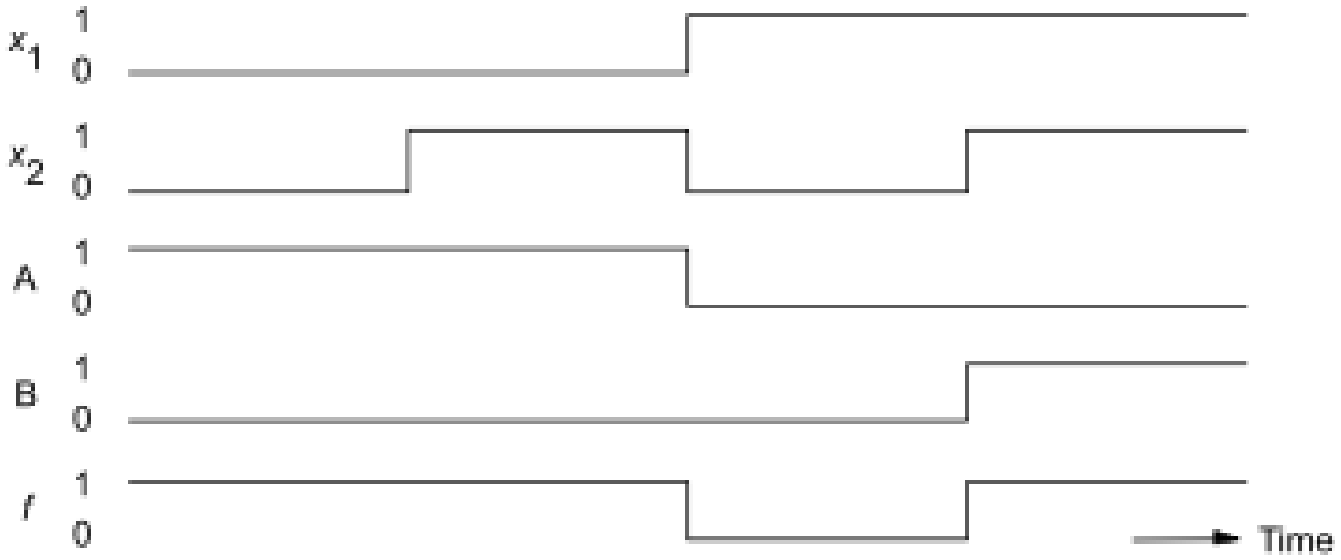
```
if (a) f = c & d;
else if (b) f = d;
else f = d & e;
```



How to verify the functionality?

Validation

- The function of a logic network can also be described by a timing diagram (gives dynamic behavior of the network)



Timing diagram

Activity

- Draw timing diagram for a 3-input (A, B and C inputs) XOR gate. Draw for a total duration of 80ns (clock high for 20ns and low for 20ns). Inputs A & B are high from 10 to 50ns, and low otherwise. C input is high from 40 to 70ns.

What is Verilog?

- What is Verilog?
  - It is a Hardware Description Language
- A language used for simulation and synthesis of digital logic
- Verilog HDL provides:
  - Mixed level modeling:
    - Behavioral: algorithmic
    - Dataflow: register transfer
    - Structural: gates, switches, modules
  - Built-in primitives, logic functions, and data types

- Official Language Document: **"IEEE Standard for Verilog Hardware Description Language", IEEE Std 1364-2005, IEEE. (older versions 1995, 2001)**
- Development continues with System Verilog. Most recent standard IEEE Std 1800-2009. Working group: http://www.vhdl.org/sv-ieee1800/

Module

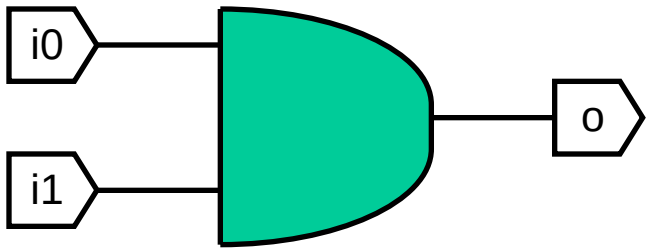
- Basic Unit – A module
  - Module is the basic building block in Verilog
  - A module definition describes a component in a circuit
    - Describes the functionality of the design
    - States the input and output ports
  - Modules can be interconnected to describe the structure of a digital system
  - Modules start with keyword **module** and end with keyword **endmodule**

```
module AND <port list>
    .
    .
    .
endmodule
```

```
module CPU <port list>
    .
    .
    .
endmodule
```

Ports

- Module Ports
  - Similar to pins on a chip
  - Provide a way to communicate with outside world
  - Ports can be input, output or inout




```
module AND (i0, i1, o);
   input  i0, i1;
   output o;









endmodule
```

Declaring A Module

- Name the module
  - Can't use Verilog keywords as module, port or signal names
  - Choose a *descriptive* module name

- List the port names (module interface)
  - Choose *descriptive* port names
  - Declare the type and size of ports

- Declare any internal signals
- Write the internals of the module (functionality)

Declaring Ports

- Only the ports are accessible from outside the module!
- A signal is attached to every port

- Declare type of port
  - **input**
  - **output**
  - **inout** (bidirectional)
- Scalar (single bit) - don't specify a size
  - **input** cin;
- Vector (multiple bits) - specify size using range
  - Range is MSB to LSB (left to right)
  - Don't have to include zero if you don't want to… (**D[2:1]**)
  - **output** [7:0] OUT; ← most common to use high:low
  - **input** [0:4] IN;

Verilog Module Styles

- Modules can be specified different ways
  - *Structural* – connect primitives and modules
  - *Data Flow* – use continuous assignments to specify combinational logic
  - *Behavioral* – use initial and always blocks to describe the behavior of the circuit, not its implementation
- A single module can (and often does) use more than one method.

- We will cover Structural first, then Data Flow, and finally Behavioral.

Structural Verilog

- Text description of a schematic
- Build up a circuit from gates/flip-flops
  - Gates are primitives (part of the language)
  - No flip-flop primitive

- Structural design
  - Create module interface
  - Instantiate the gates in the circuit
  - Declare the internal wires needed to connect gates
  - Put the names of the wires in the correct port locations of the gates to make connections
    - For primitives, outputs always come first

Structural Basics: Primitives

- Build design up from the gate level
  - Flip-flops usually constructed using Behavioral Verilog
- Verilog provides a set of gate primitives
  - and, nand, or, nor, xor, xnor, not, buf, etc.
  - Combinational building blocks for structural design
  - Each primitive is a “black box”
    - and(f,a,b,…) f = (a · b …)
    - or(f,a,b,…) f = (a + b + …)
    - not(f,a) f = a'
    - nand(f,a,b,…) f = (a · b …)'
    - nor(f,a,b,…) f = (a + b + …)'
    - xor(f,a,b,…) f = (a ⊕ b ⊕ …)
    - xnor(f,a,b,…) f = (a b …)

Primitives

- No declarations - can only be instantiated
- Output port appears before input ports
- Optionally specify: instance name and/or delay (discuss delay later)

```
and N25 (Z, A, B, C);      // name specified
and #10 (Z, A, B, X),
        (X, C, D, E);      // 2 gates, delay specified
and #10 N30 (Z, A, B); // name and delay specified
```

Verilog Example - 2to4 Decoder with Enable input

module name

port names of module

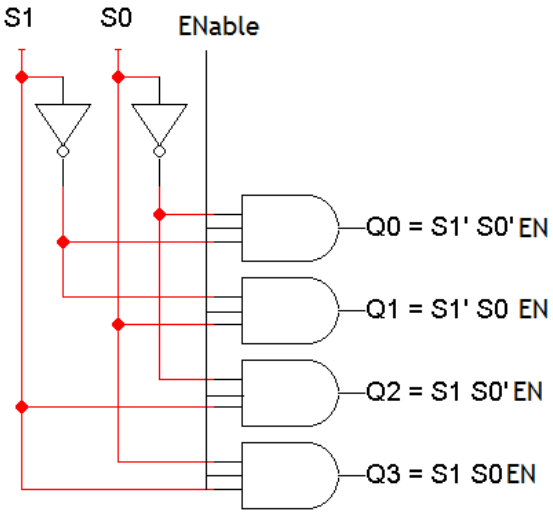
```
module decoder_2to4 (S1,S0,EN,Q);
    input S1,S0,EN;
    output [3:0]Q;

    wire S1not,S0not;
        not
            n1 (S1not,S1),
            n2 (S0not,S0),
        and
            n4 (Q[0],S1not,S0not,EN),
            n5 (Q[1],S1not,S0,EN),
            n6 (Q[2],S1,S0not,EN),
            n7 (Q[3],S1,S0,EN);
endmodule
```

port types

port sizes

module contents (what it "does")

keywords underlined



Activity

Write Verilog code for a 3-input XOR gate

Declaration vs. Instantiation

- Declaration is when you define a module
  - Its interface
  - Its functionality

xor_2 is **declared** here

```
module xor_2 (Y, A, B) ;
input  A, B ;
output Y ;
assign Y = A ^ B;
endmodule
```

- Instantiation is when you "insert" the module into your design

xor_3 is **declared** here

xor_2 is **instantiated** twice here (two copies)

```
module xor_3 (F, C,D,E) ;
input  C, D, E ;
wire G;
xor_2 x1 (G, C, D) ;
xor_2 x2 (F, G, E) ;
endmodule
```

Stuctural Verilog - Example

Module name *Module ports*

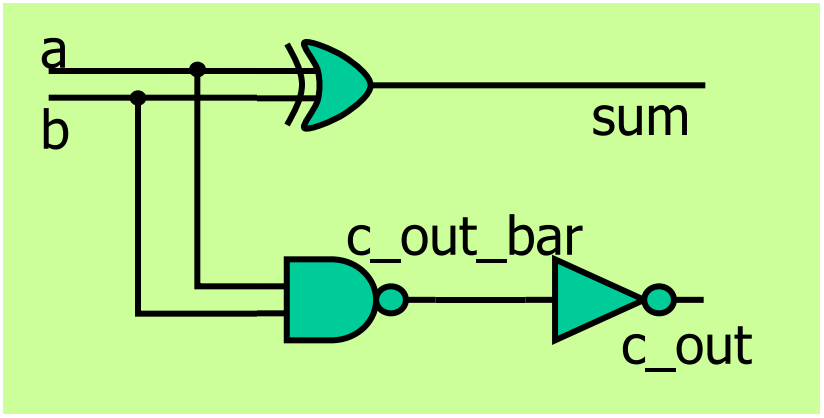
```
module Half_Add ( sum, c_out, a, b );
    input       a, b;
    output      sum, c_out;
    wire        c_out_bar;

    xor (sum, a, b);
    nand (c_out_bar, a, b);
    not (c_out, c_out_bar);
endmodule
```

Verilog keywords

Declaration of port modes

Declaration of internal signal

Instantiation of primitive gates



Port Declarations

- Syntax:

```
port_direction [port_size] port_name, port_name,... ;
```

- `port_direction` is declared as:
  - `input` for scalar or vector input ports.
  - `output` for scalar or vector output ports.
  - `inout` for scalar or vector bi-directional ports.
- `port_size` is a range from `[ msb : lsb ]`

Examples	Notes
input a,b,sel;	3 scalar ports
output [7:0] result;	little endian convention
inout [0:15] data_bus;	big endian convention
input [15:12] addr;	msb:lsb may be any integer
parameter word = 32; input [word-1:0] addr;	constant expressions may be used

Data Types (Signal Classification)

- There are two kinds of variables in Verilog
  - *nets*: used to represent structural connectivity
  - *registers*: used as abstract storage elements

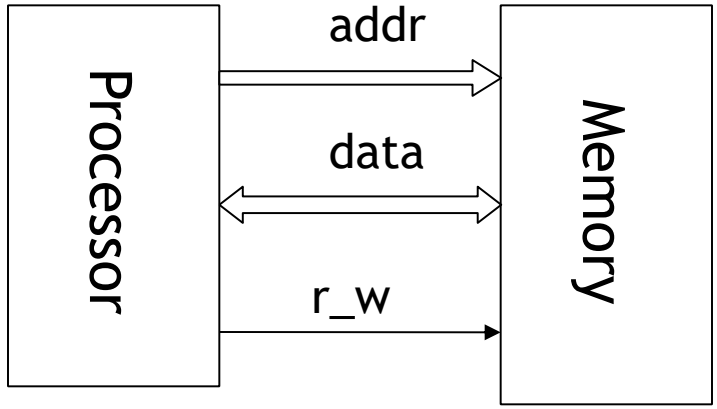
- Net Data Type
  - Nets are physical connections between devices (physical wire)
  - Nets always reflect the logic value of the driving device
  - Many types of nets, but most of the time we use **`wire`**
  - A *net* may be assigned a value **explicitly** only by a *continuous assignment* statement or **implicitly** as an output of a primitive or module.
- Register Data Type
  - Register: Implicit storage – unless variable of this type is modified it retains previously assigned value
    - Does not necessarily imply a hardware register
    - A register is like a variable in programming languages. It keeps its value until a new value is assigned to it.
    - Register type is denoted by **`reg`**
    - The register declaration explicitly specifies the size

By default data type is wire.

Net declaration

- A net declaration starts with keyword **wire**

```
.....
wire          r_w;     // scalar signal
wire  [7:0]   data;    // vector signal
wire  [9:0]   addr;    // vector signal
.....
```

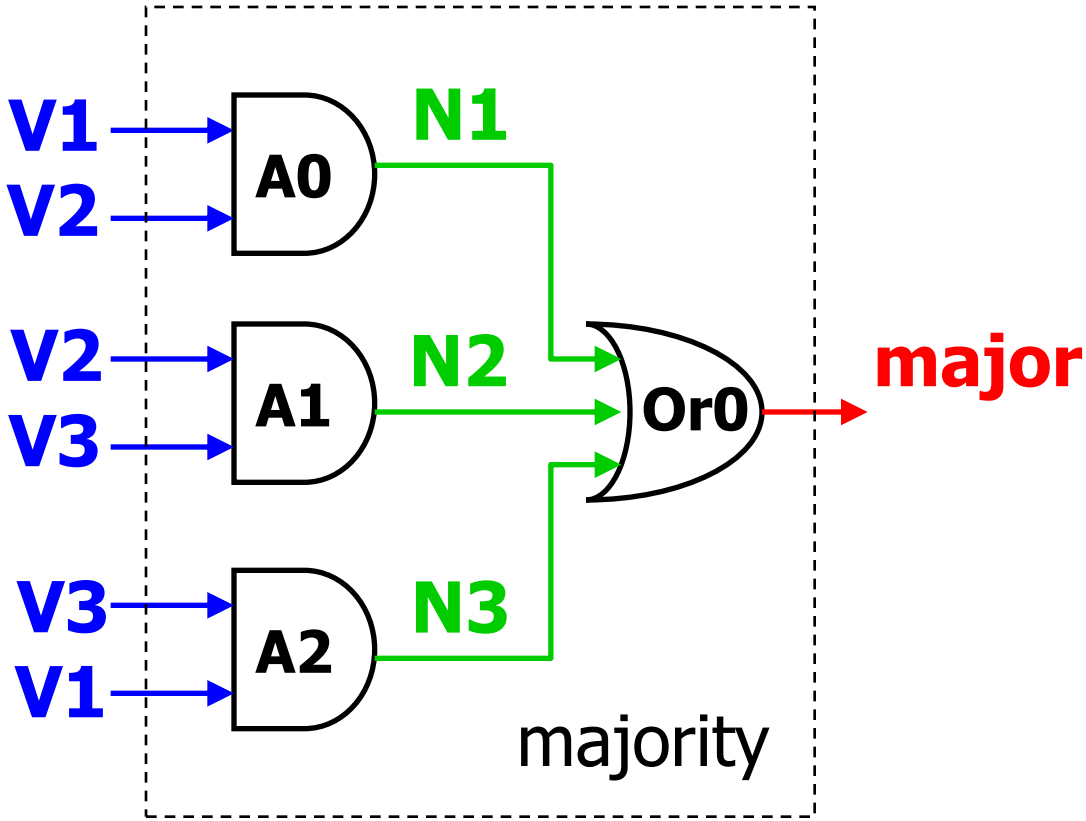


- Selecting a single bit or a portion of vector signals
  - data[2] single bit
  - data [5:3] 3 bits

Structural Example: Majority Detector

- Structural models specify interconnections of primitives and modules.
- Synthesis tools may still optimize your design!

```
module majority (major, V1, V2, V3) ;

output major ;
input V1, V2, V3 ;

wire N1, N2, N3;

and A0 (N1, V1, V2),
    A1 (N2, V2, V3),
    A2 (N3, V3, V1);

or  Or0(major, N1, N2, N3);

endmodule
```

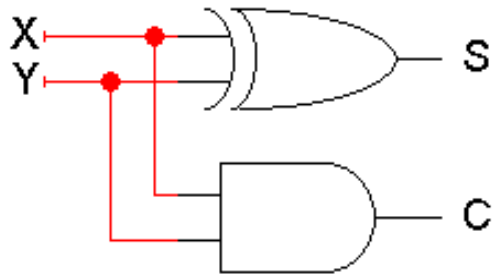


Acitivity

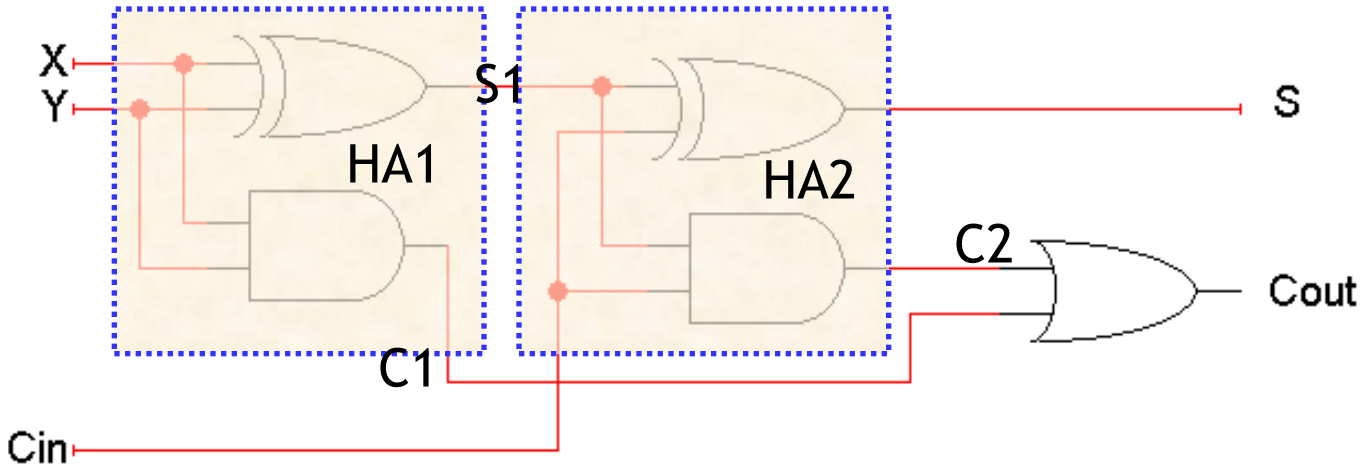
- Modify the same majority function for higher bit width: inputs and outputs are all 3 bits. Output bits are defined by corresponding input bits.

Half Adder - Structural Verilog Design

```
module ADD_HALF (S,C,X,Y);
    output S,C;
    input X,Y;
    wire S,C,X,Y;
    // this line is optional since nodes default to wires

    xor G1 (S,X,Y); // instantiation of XOR gate
    and G2 (C,X,Y); // instantiation of AND gate
endmodule
```



Full Adder - Structural Verilog Design




```
module ADD_FULL (S,Cout,X,Y,Cin);
    output S,Cout;
    input X,Y,Cin;
    //internal nodes also declared as wires
    wire Cin,X,Y,S,Cout,S1,C1,C2;
    ADD_HALF HA1(S1,C1,X,Y);
    ADD_HALF HA2(S,C2,Cin,S1);
    or (Cout,C1,C2);
endmodule
```

Using parameters

- Constants: Declared using the keyword **parameter**
- The use of parameters make code easy to read and modify
  - **parameter** byte_size = 8 ; // integer
- Example:

```
……
parameter  bussize = 8;
reg  [bussize-1 : 0]  databus1;
reg  [bussize-1 : 0]  databus2;
……
```

Open Source Simulators

- https://www.edaplayground.com/x/B
- https://www.tutorialspoint.com/compile_verilog_online.php
- https://www.jdoodle.com/execute-verilog-online