Imperial College
London

Lecture 2

Hardware Design with SystemVerilog

Prof Peter YK Cheung
Imperial College London

URL: www.ee.ic.ac.uk/pcheung/teaching/EIE2-IAC/
E-mail: p.cheung@imperial.ac.uk

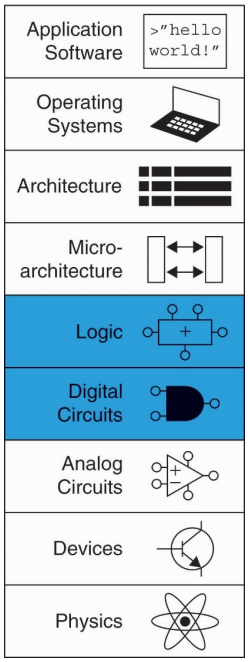
This lecture will focus on the Hardware Description Language (HDL), SystemVerilog (SV). You will not learn this HDL by listening at lectures. Instead, you will be learning SV by designing actual circuits and testing them.

Learning outcome for week 2

- Modules in SystemVerilog (SV)
- Syntax, operators, number formats in SV
- Behavioural vs structural description
- Combinational circuit description
- Sequential circuit description
- Blocking and non-blocking assignment

Slides in this lecture are derived and modified from:

"*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H), Morgan Kaufmann, 2022



Here is a list of things that you will learn in this lecture. Some of the slides are derived or modified from the slides provided by the publisher of the Harris & Harris book. This is another recommended textbook for this module. It is much thinner than the Patterson & Hennessy book, and is a more suitable textbook in many ways to support this module. Unfortunately, our library does not have an electronic copy of H&H for student to borow. If you can acquire a copy of this textbook, I recommend it highly.

Hardware description Languages

❖ Hardware description language (HDL):

- Specifies logic function only
- Computer-aided design (CAD) tool produces or synthesizes the optimized gates

❖ Most commercial designs use HDLs

❖ Two leading HDLs:

- **SystemVerilog**
  - ➢ Developed in 1984 by Gateway Design Automation (Verilog)
  - ➢ IEEE standard (1364) in 1995
  - ➢ Extended in 2005 (IEEE STD 1800-2009)
- **VHDL 2008**
  - ➢ Developed in 1981 by the Department of Defense
  - ➢ IEEE standard (1076) in 1987
  - ➢ Updated in 2008 (IEEE STD 1076-2008)

These are the pros and cons of using a HDL instead of schematic to specify digital hardware:

✓ Flexible & parameterisable

✓ Excellent input to optimisation & synthesis

✓ Direct mapping to algorithms

✓ Excellent for datapaths

✓ Easy to handle electronically (only needing a text editor)

× Serial representation

× May not show overall picture

× Need good programming skills

× Divorce from physical hardware

No modern digital integrated circuits or FPGA based designs that are not specified in some sort of HDL from which the final design is synthesized.

For this module, you will learn a particular level of **abstraction** of the processor hardware known as **Register Transfer Level** (RTL). In RTL specifications, all combinational logic are sandwiched between registers controlled by one or more clock signals.

HDL to Gates

- **Simulation**
  - Inputs applied to circuit
  - Outputs checked for correctness
  - Millions of dollars saved by debugging in simulation instead of hardware
- **Synthesis**
  - Transforms HDL code into a netlist describing the hardware (i.e., a list of gates and the wires connecting them)
- **Physical design**
  - Placement, routing, chip layout, …… – not considered in this module

IMPORTANT:

When using an HDL, think of the **hardware** the HDL should produce, then write the appropriate idiom that implies that hardware.

Beware of treating HDL like software and coding without thinking of the hardware.

H&H 171-173

After specifying your hardware in SystemVerilog HDL, you need to make sure that your design works according to specification. Simulation tools such as circuit simulators, Matlab, Mathematica etc. allow users to predict circuits and systems behaviour WITHOUT having to implement the actual electronic system. This saves both time and money. Furthermore, it is very hard to find a bug in a million or billion transistor circuit on a physical chip because there is no easy way to access internal signals. (This statement is not entirely true. There is a technique used called "**scan chain**" or **JTAG,** which allows such internal access, but it is not easy to use.)

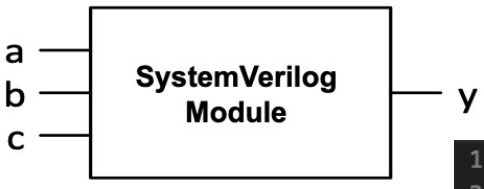
After simulation, the design is "**translated**" to low level building blocks (such as gates and flops) through a special type of **hardware compiler** to perform **synthesis**. This is the stage at which circuits can be optimized. For example, redundant gates (such as a 2-input NAND gate with one input always 0) are eliminated. Synthesis produces a network of interconnected building blocks, known as the **netlist**. At this stage, the design is still not necessary linked to any technology for implementation.

The netlist is then further processed to produce the final physical design. This final stage involves many steps such as **technology mapping**, **placement**, **routing**, **timing analysis, test vector generation**, **test coverage analysis** etc. We will NOT be considering any part of this stage of design in this module.

SystemVerilog: Module Declaration

❖ **Two types of Modules:**

- **Behavioral**: describe what a module does
- **Structural**: describe how it is built from simpler modules




```
1  module example(input  logic a, b, c,
2                 output logic y);
3    // module body goes here
4  endmodule
```

❖ **module/endmodule: required to begin/end module**

❖ **example: name of the module**

A SystemVerilog design consists of basic units called "**modules**". Each module, like a C function, provides specific functionality. Unlike C functions, modules are not "called" but "**instantiated**". That means that each time you use a module in SV, you "**clone**" a separate entity – the clone has a totally separate existence.

SV is entirely hierarchical. Modules can instantiate other modules.

All modules have inputs and outputs as shown on the slide.

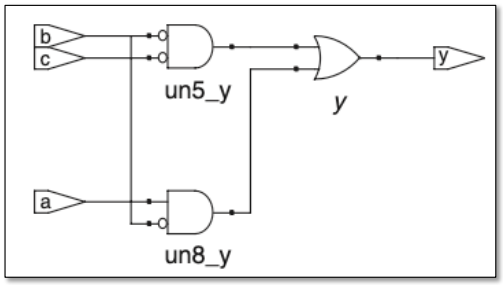
There are many different level of abstractions in specifying a module:

1. You can specify something at a **behavioural level** where the SV syntax allows you to describe the abstract functional behaviour rather than physical structure of the hardware.
2. Alternatively, you may describe a module in a **structural form**. For example, a top-level (chip level) module may consists of numerous lower-level modules interconnected together.

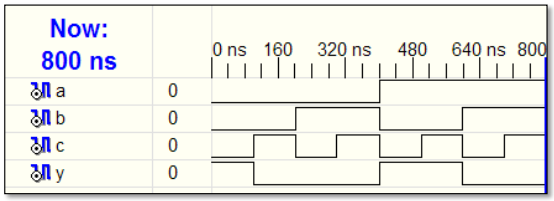
SystemVerilog: Behavioural Description

```
module example(input  logic a, b, c,
               output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

synthesis



simulation



Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

H&H 174

Here is a simple example of a combinational circuit consisting of many Boolean operations described in SV as a Boolean equation. We use the "**assign**" keyword to specify combinational circuit. We then use ~, & and | for NOT, AND and OR Boolean operations respectively.

Sythesis will produce optimized logic as shown in the schematic. Simulation will produce a trace file (i.e. a file contains signal values over time), which can be plotted as timing diagrams.

SystemVerilog: Syntax

- **Case sensitive**
  - **e.g.: reset and Reset are not the same signal.**
- **No names that start with numbers**
  - **e.g.: 2mux is an invalid name**
- **Whitespace ignored**
- **Comments:**
  - **// single line comment**
  - **/* multiline**
  - **comment */**

Here are some basic rules about naming variables in SystemVerilog. It is very much like C or C++.

SystemVerilog: Structural Description

Behavioural

```
module and3(input  logic a, b, c,
            output logic y);
  assign y = a & b & c;
endmodule
```

```
module inv(input  logic a,
           output logic y);
  assign y = ~a;
endmodule
```

Structural

```
module nand3(input  logic a, b, c
             output logic y);
  logic n1;                  // internal signal

  and3 andgate(a, b, c, n1); // instance of and3
  inv  inverter(n1, y);      // instance of inv
endmodule
```

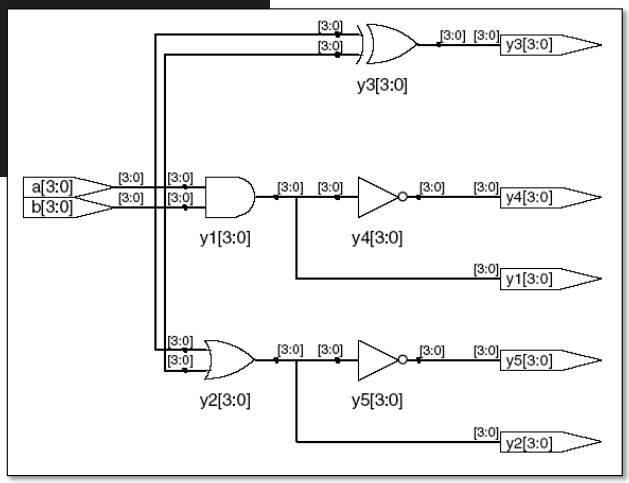

Combinational circuit is easiest to specify using behavioural specification with Boolean operators. You can also choose to provide structural description with interconnected gates as shown on the right.

It is NOT advisable to describe low-level modules in a structural way. It is both tedious, prone to error and not easy to read.

We normally only use structural description when we connect large modules together at a higher level of the design hierarchy.

SystemVerilog: Bitwise Operators

```
module gates(input  logic [3:0]  a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
  /* Five different two-input logic
     gates acting on 4 bit busses */
  assign y1 = a & b;    // AND
  assign y2 = a | b;    // OR
  assign y3 = a ^ b;    // XOR
  assign y4 = ~(a & b); // NAND
  assign y5 = ~(a | b); // NOR
endmodule
```



H&H 177

Here is an example where signals are bundled into multi-bit bus. In this case, they are 4-bit wide as [3:0]. SV does not restrict you to name the bus from bit 3 to bit 0. You could declare the signals as, say, [4:1] instead. However, we adapt the notation that LSB is bit 0, and MSB is WIDTH-1, in this case 3.

Now the continuous assignment keyword “assign” results in bit-wise operation. For example:

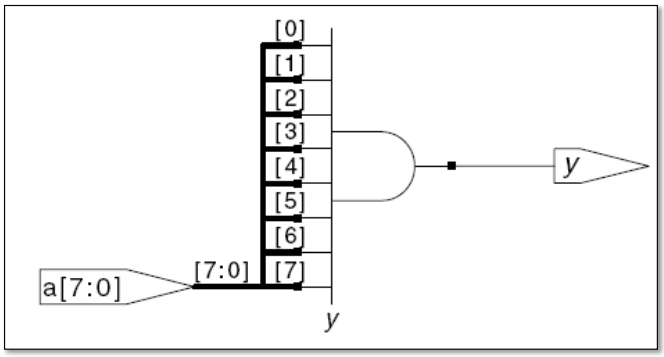
assign y1 = a & b;

Means:

y1[3] = a[3] & b[3], y1[2] = a[2] & b[2]. ……

SysytemVerilog: Reduction Operators

```
module and8(input  logic [7:0] a,
            output logic       y);
  assign y = &a;
  // &a is much easier to write than
  // assign y = a[7] & a[6] & a[5] & a[4] &
  //            a[3] & a[2] & a[1] & a[0];
endmodule
```

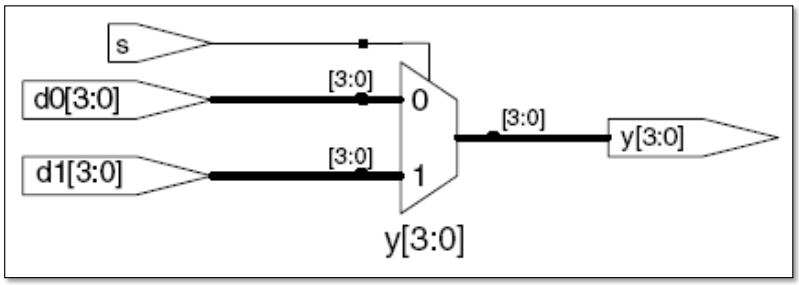


H&H 178

The '&' operator can also be used with a single operand as shown here. This is called a "**reduction**" operator. It reduces multiple bits of a[7:0] to a single bit y. It basically ANDs all bits of a[7:0] together as shown in the slide.

SystemVerilog: Conditional Assignment

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);
   assign y = s ? d1 : d0;
endmodule
```



H&H 179

The conditional assignment operator (as found in C or C++) is:

cond ? True_value : False_value

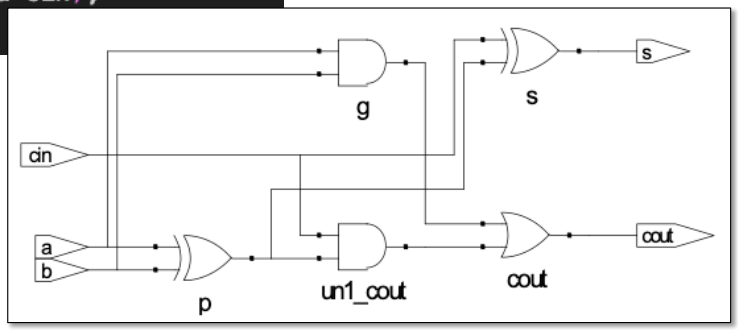
Therefore, assign y = s ? d1 : d0;

Is the same as: If s is true, y = d1, else y = d0.

This effectively produces a **multiplexer** as shown here.

SystemVerilog: Internal Signals

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;   // internal nodes

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```



H&H 182

For most modules, there are internal signals which are neither inputs nor outputs. The module here is a single bit full adder. There are two internal signals p, g.

These signals are not “visible” outside the module and are declared as local signals (similar to local variables in C++ functions).

SystemVerilog: Precedence of operators

Highest

~	NOT
*, /, %	mult, div, mod
+, -	add, sub
<<, >>	shift
<<<, >>>	arithmetic shift
<, <=, >, >=	comparison
==, !=	equal, not equal
&, ~&	AND, NAND
^, ~^	XOR, XNOR
|, ~|	OR, NOR
?:	ternary operator

Lowest

H&H 183

Here are all the operators that SystemVerilog understands. They are listed here with their precedence.

SystemVerilog: Number Format

Format: N'Bvalue

N = number of bits, **B** = base

N'B is optional but recommended (default is decimal)

Number	# Bits	Base	Decimal Equivalent	Stored
3'b101	3	binary	5	101
'b11	unsized	binary	3	00…0011
8'b11	8	binary	3	00000011
8'b1010_1011	8	binary	171	10101011
3'd6	3	decimal	6	110
6'o42	6	octal	34	100010
8'hAB	8	hexadecimal	171	10101011
42	Unsized	decimal	42	00…0101010

H&H 184

When using SystemVerilog to describe hardware, always remember that you are NOT writing a program. All “variables” are in fact signals. So, when specifying number, beware that you are using physical wire.

Therefore numbers are specified with number of bits explicitly stated. The general format is N’Bxxxx.

N is the number of bits. **B** is the base: b = binary, d = decimal, h = hexadecimal.

See above. If you don’t provide bit and base specification, the number is assumed to be 32 bits and in decimal by default. Not specifying the size (i.e. number of bits) of a signal in a design is not recommended.

SystemVerilog: Bit Manipulations (1)

```
assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};
```

- If y is a 12-bit signal, the above statement produces:

```
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0
```

- Underscores (_) are used for formatting only to make it easier to read. **SystemVerilog ignores them**.

The syntax shown here is very unlike C or C++, and is particularly important to specification of hardware.

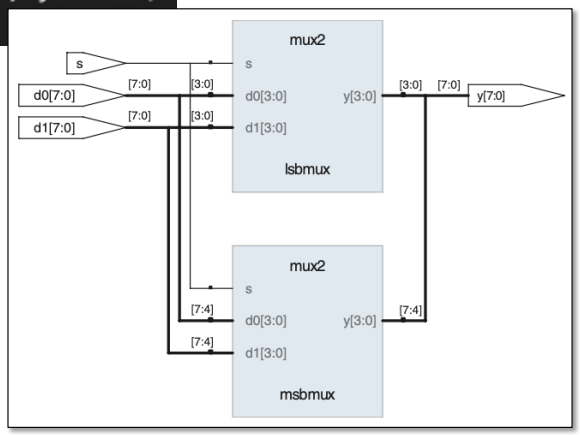
{ . } is called a **concatenation** operation. { 1, 0, 1, 1} forms a 4-bit number 4'b1011.

In the example above, a[2:1] is a two bit number a[2] and a[1].

{ 3 {b[0]} } forms a three bit number with b[0] repeated 3 times.

SystemVerilog: Bit Manipulations (2)

```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```



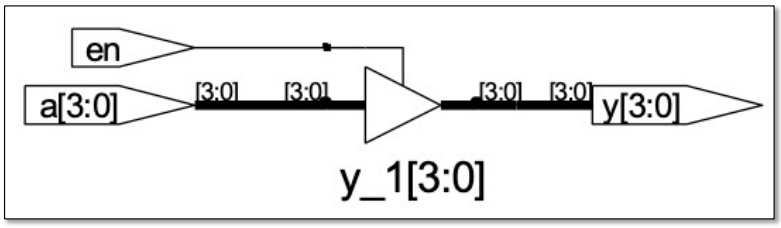
H&H 190

This is an example of slicing and merging different bits of signals d0 and d1 to form an 8-bit output y.

If d0 = 8'b10110101, and d1 = 8'h5A, work out what is y for s = 0, and s = 1?

SystemVerilog: Floating Output Z

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);
   assign y = en ? a : 4'bz;
endmodule
```



- Note that Verilator does not handle floating output Z

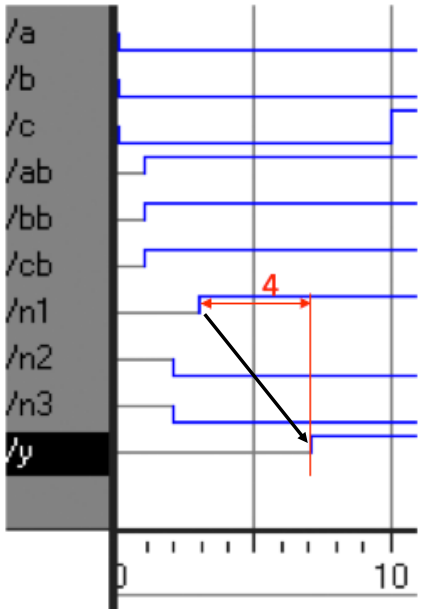
H&H 185

We normally use "**logic**" to specify a signal to be a signal which has values of 0 or 1. However, there is a signal type **tri** which can take on three values: 0, 1, or z, where z is high impedance. This allows SystemVerilog to **describe tri-state outputs**.

In this module, and if en=1, then y = a. If en=0, the output y is tri-state and is therefore not driven by this module.

SystemVerilog: Delays

```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



- Delays are for simulation only! They do not determine the delay of your hardware.
- **Verilator similator ignores delays** – it is cycle accurate without timing.

H&H 187

Digital circuits have **delays**. SystemVerilog provides constructs to specify such delays (default in ns). However, Verilator ignores all such specifications: Verilator assumes that all combinational logic output changes immediately with inputs. As such, Verilator is NOT suitable to verify physical digital circuits – it can only be used for functional verification.

SystemVerilog: Sequential Logic

- SystemVerilog uses **idioms** (or special keywords or groups of words) to describe latches, flip-flops and FSMs
- Other coding styles may simulate correctly but produce incorrect hardware
- GENERAL STRUCTURE:

```
always @(sensitivity list)
        statement;
```

- Whenever the event in **`sensitivity list`** occurs, **`statement`** is executed

H&H 191

Sequential logics are specified using the pattern:

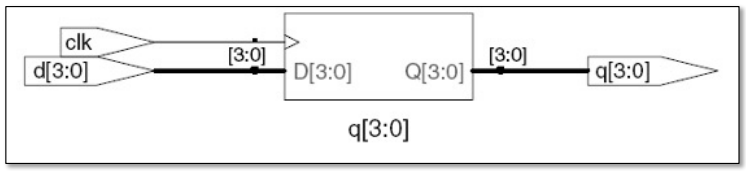
```
always @(sensitivity list)
        statement;
```

The "**always**" followed by **@(sensitivity list)** means that when any signal in the sensitivity list is asserted, "statement" is executed.

All sequential circuits are described in this form.

SystemVerilog: D Flip-Flop

```
module flop(input  logic       clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;                 // pronounced "q gets d"

endmodule
```



H&H 192

SystemVerilog has a specific syntax for D flip-flops.

aways_ff @(posedge clk)

will synthesize one or more registers that are triggered on **positive edge** of the signal **clk**.

Note that you can call your clock signal anything, e.g. **fred** would do equally well. There is NO SIGNIFICANCE in the name itself. However, it is of course advisable to use a signal name that is meaningful.

Note also that the statement to execute in this case is:

q <= d;

This is called **non-blocking assignment** (but don't worry about what it is called for now). The effect of this module is: on rising edge of clk, the 4-bit value of d is transferred to q.

This will synthesize to 4-bit D flip-flop.

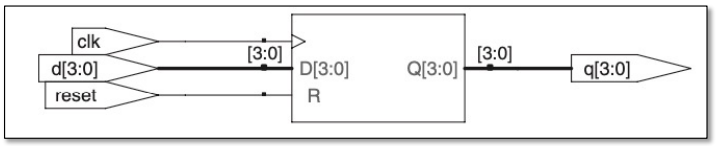
SystemVerilog: Resettable D Flip-Flop

Asynchronous reset

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

// asynchronous reset
always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else q <= d;
endmodule
```

Synchronous reset

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // synchronous reset
  always_ff @(posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```



H&H 193

You should **ALWAYS** add a **reset** control to your flops. Otherwise, your digital system may power up in a random state.

Reset can be implemented as **synchronous** or **asynchronous**. Synchronous reset means that reset happens only on the active edge of the clock signal. Asynchronous reset can happen anytime whenevert the reset signal is asserted and is independent of the clock.

The slide shows the two forms of reset description. For asychonrous case, it also shows how the sensitivity list can **contain multiple conditions**.

Combinational Logic using always

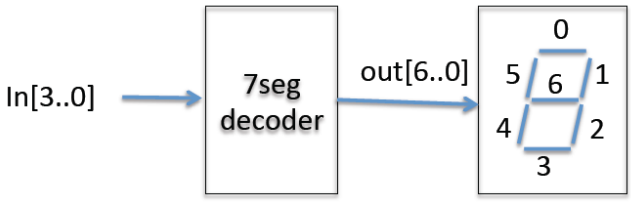
```
// combinational logic using an always statement
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
  always_comb        // need begin/end because there is
    begin            // more than one statement in always
      y1 = a & b;    // AND
      y2 = a | b;    // OR
      y3 = a ^ b;    // XOR
      y4 = ~(a & b); // NAND
      y5 = ~(a | b); // NOR
    end
endmodule
```

This hardware could be described with **`assign` statements using fewer lines** of code, so it's better to use **`assign`** statements in this case.

H&H 198

There is a form of **always block** which allows the specification of **combinational circuits**. However, there is no advantage in this form of specification as compare to multiple assign statements.

Combinational Logic using always-case



in[3..0]	out[6:0]	Digit
0000	1000000	0
0001	1111001	1
0010	0100100	2
0011	0110000	3
0100	0011001	4
0101	0010010	5
0110	0000010	6
0111	1111000	7

in[3..0]	out[6:0]	Digit
1000	0000000	8
1001	0010000	9
1010	0001000	A
1011	0000011	b
1100	1000110	C
1101	0100001	d
1110	0000110	E
1111	0001110	F

H&H 199

```
module hex_to_7seg (
    output  logic [6:0]   out,
    input   logic [3:0]   in);

    always_comb
        case (in)
        4'h0:   out = 7'b1000000;
        4'h1:   out = 7'b1111001;
        4'h2:   out = 7'b0100100;
        4'h3:   out = 7'b0110000;
        4'h4:   out = 7'b0011001;
        4'h5:   out = 7'b0010010;
        4'h6:   out = 7'b0000010;
        4'h7:   out = 7'b1111000;
        4'h8:   out = 7'b0000000;
        4'h9:   out = 7'b0011000;
        4'ha:   out = 7'b0001000;
        4'hb:   out = 7'b0000011;
        4'hc:   out = 7'b1000110;
        4'hd:   out = 7'b0100001;
        4'he:   out = 7'b0000110;
        4'hf:   out = 7'b0001110;
        endcase
    endmodule
```


A more common use of the **always_comb** statement is when it is used with the **case** statement.

Here is a 7-segment decoder specification. A 4-bit binary input in[3:0] is decoded to provide 7 output signals to drive a 7-segment display. The outputs are assumed to be **low-active**, i.e. the segments turns ON when the output signals [6:0] are driven LOW.

The function of the decoder can be specified in the truth table shown.

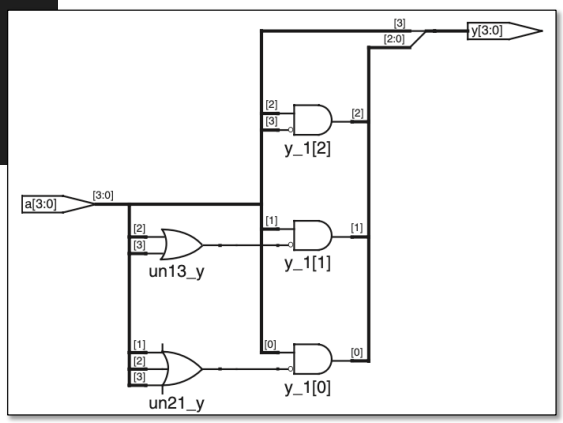
The **case statement** here shows a direct way to specify such a decoder.

In general, any truth tables or ROMs can be specified in this way.

Combinational Logic using if-else

❖ Prioity encoder circuit

```
module priorityckt( input logic  [3:0] a,
                    output logic [3:0] y);
always_comb
    if (a[3])       y = 4'b1000;
    else if (a[2])  y = 4'b0100;
    else if (a[1])  y = 4'b0010;
    else if (a[0])  y = 4'b0001;
    else            y = 4'b0000;
endmodule
```



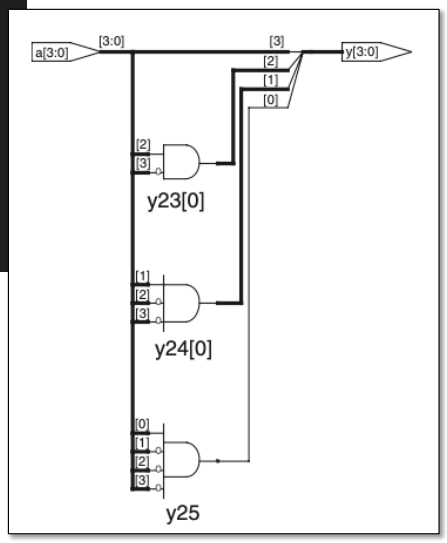
H&H 202

Here is another example called **priority encoder**. **always_comb** statement uses **if-else** constructs. The output y[3:0] reports the position of the first '1' in the input from MSB to LSB. So if a[3] is '1', then y[3] is '1' etc.

This is call a **priority encoder** because it detects the highest priority signal being set. The if-else statement is perfect for such description because it fully describes the behaviour of the circuit explicitly.

Combinational Logic using casez

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);
  always_comb
    casez(a)
      4'b1???: y = 4'b1000; // ? = don't care
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
    endcase
endmodule
```



- ? = don't-care
- Beware: MUST have default statement in case not all cases are covered!

Here is an alternatively method to do the same thing using the casez statement.

Here the conditions are specified with '?' meaning "**don't care**". Note that with 4-bits input, there are 16 possibilities. '?' allows these bit values to be either '0' or '1' (i.e. don't care).

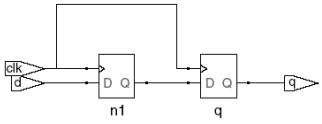
However, beware that not all cases may be covered by such specifidation. You MUST always specify the default case (i.e. when the input a value is not included in the case list).

Blocking vs. Nonblocking Assignment

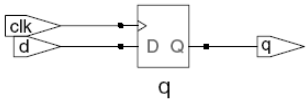
- <= is nonblocking assignment
  - Occurs simultaneously with others
- = is blocking assignment
  - Occurs in order it appears in file

```
// Good synchronizer using
// nonblocking assignments
module syncgood(input  logic clk,
                input  logic d,
                output logic q);
  logic n1;
  always_ff @(posedge clk)
    begin
      n1 <= d;  // nonblocking
      q  <= n1; // nonblocking
    end
endmodule
```

```
// Bad synchronizer using
// blocking assignments
module syncbad(input logic  clk,
               input  logic d,
               output logic q);
  logic n1;
  always_ff @(posedge clk)
    begin
      n1 = d;  // blocking
      q  = n1; // blocking
    end
endmodule
```



H&H 203



Inside any always block, you should use the **non-blocking assignement** "<=" instead of the **block assignment** "=".

With <=, all assignment statements take effectly ONLY at the end of the always block simultaneously.

With = assignment, assignment occurs sequentially. The synthesized results is a single flip-flop instead of a shift register.

ALWAYS USE "<=" IN YOUR SEQUENTIAL CIRCUIT SPECIFICATION.

Rules for Signal Assignment

- **Synchronous sequential logic, use:**

  `always_ff` and nonblocking assignments (<=)

```
always_ff @(posedge clk)
    q <= d; // nonblocking
```

- **Simple combinational logic, use continuous assignments (assign…)**

```
assign y = a & b;
```

- **More complicated combinational logic, use:**

  `always_comb and blocking assignments (=)`

- **Assign a signal in ONLY ONE `always` statement or continuous `assignment` statement.**

Here are some general rules about assingments.