Imperial College
London

Lecture 8

SystemVerilog HDL

Peter Cheung
Imperial College London

URL: www.ee.imperial.ac.uk/pcheung/teaching/EE2_CAS/
E-mail: p.cheung@imperial.ac.uk

This lecture is about Verilog HDL, which, together with another language VHDL, are the most popular hardware languages used in industry.

Verilog is only a tool; this course is about digital electronics. Therefore, I will NOT be going through Verilog as in a programming course - it would have been extremely boring for both you and me if I did. Instead, you will learn about Verilog through examples. I will then point out various language features along the way. What it means is that the treatment of Verilog is NOT going to be systematic or comprehensive – there will be lots of features you won't know about Verilog. However, you will learn enough to specify and design reasonably sophisticated digital circuits, and you should gain enough confidence to learn the rest by yourself.

There are many useful online resources available on details of Verilog syntax etc.. Look it up as you need to and you will learn how to design digital circuit using Verilog through designing real circuits.

.

Lecture Objectives

- By the end of this lecture, you should understand:
  - The basic structure of a module specified in SystemVerilog HDL
  - Commonly used syntax of SystemVerilog HDL
  - Continuous vs Procedural Assignments
  - **always** block in SystemVerilog and sensitivity list
  - The use of arithmetic and logic operations in SystemVerilog
  - The danger of **incomplete specification**
  - How to specify **clocked circuits**
  - Differences between **blocking** and **nonblocking** assignments

Here is a list of lecture objectives. They are provided for you to reflect on what you are supposed to learn, and not as an introduction to this lecture.

I want, by the end of this lecture, to give you some idea about the basic **structure** and **syntax** of Verilog. I want to convince you that schematic capture is NOT a good way to design digital circuits. Finally, I want you to appreciate how to use Verilog to specify a piece of hardware at **different levels of abstraction**.

Schematic vs HDL

Schematic	HDL
✓ Good for multiple data flow	✓ Flexible & parameterisable
✓ Give overview picture	✓ Excellent input to optimisation & synthesis
✓ Relate directly to hardware	✓ Direct mapping to algorithms
✓ Don't need good programming skills	✓ Excellent for datapaths
✓ High information density	✓ Easy to handle electronically (only needing a text editor)
✓ Easy back annotations	
✓ Useful for mixed analogue/digital	
× Not good for algorithms	× Serial representation
× Not good for datapaths	× May not show overall picture
× Poor interface to optimiser	× Need good programming skills
× Poor interface to synthesis software	× Divorce from physical hardware
× Difficult to reuse	
× Difficult to parameterise	

You are very familiar with schematic capture. However modern digital design methods in general DO NOT use schematics. Instead an engineer would specify the design requirement or the algorithm to be implemented in some form of computer language specially designed to describe hardware. These are called "Hardware Description Languages" (HDLs).

The most important advantages of HDL as a means of specifying your digital design are: 1) You can make the design take on parameters (such as number of bits in an adder); 2) it is much easier to use compilation and synthesis tools with a text file than with schematic; 3) it is very difficult to express an algorithm in diagram form, but it is very easy with a computer language; 4) you can use various datapath operators such as +, * etc.; 5) you can easily edit, store and transmit a text file, and much hardware with a schematic diagram.

For digital designs, schematic is NOT an option. Always use HDL. In this lecture, I will demonstrate to you why with an example.

SystemVerilog HDL

- Similar to C language to describe/specify hardware
- Description can be at different levels:
  - **Behavioural level**
  - **Register-Transfer Level (RTL)**
  - **Gate Level**
- Not only a specification language, also with associated **simulation environment**
- Easier to learn and "lighter weight" than its competition: VHDL
- Very popular with chip designers

- For this lecture, we will:
  - Learn through examples and practical exercises
  - Use examples: e.g. 2-to-1 multiplexer and 7 segment decoder

I have chosen to use Verilog HDL as the hardware description language for this module. Verilog is very similar to the C language, which you should already know from last year. However, you must always remember that YOU ARE USING IT TO DESCRIBE HARDWARE AND NOT AS A COMPUTER PROGRAMME.

You can use Verilog to describe your digital hardware in three different level of abstraction:

1) Behavioural Level – you only describe how the hardware should behave without ANY reference to digital hardware.

2) Register-Transfer-Level (RTL) – Here the description assumes the existence of registers and these are clocked by a clock signal. Therefore digital data is transferred from one register to the next on successive clock cycles. Timing (in terms of clock cycles) is therefore explicitly defined in the Verilog code. This is the level of design we use most frequently in this course.

3) Gate Level – this is the lowest level description where each gate and its interconnection are explicitly specified.

Verilog is not only a specification language which tells the CAD system what hardware is supposed to do, it also includes a complete simulation environment. A Verilog compiler does more than mapping your code to hardware, it also can **simulate** (or execute) your design to predict the behaviour of your circuit. It is the predominant language used for chip design.

You will learn Verilog through examples and exercises, not through lecture. However, I will spend just this lecture to cover the basics of Verilog.

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.

After specifying your hardware in System Verilog 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.)

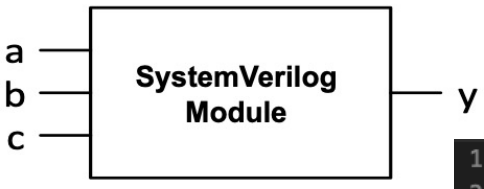
After simulation, the design is "**translated**" to low level building blocks (such as gates and flops) through a special type of **hardware compiler** to perform **synthesis**. This is the stage at which circuits can be optimized. For example, redundant gates (such as a 2-input NAND gate with one input 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 System Verilog 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.

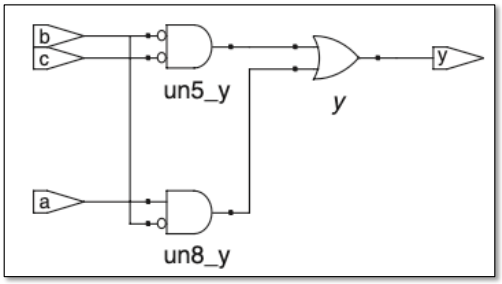
There are many different level of abstractions in specifying a module:

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.

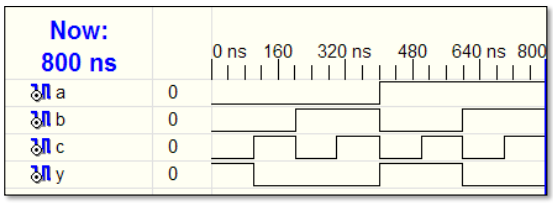
System Verilog: Behavioural Description

```
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),

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.

System Verilog: 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 */**

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

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

System Verilog: 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
```

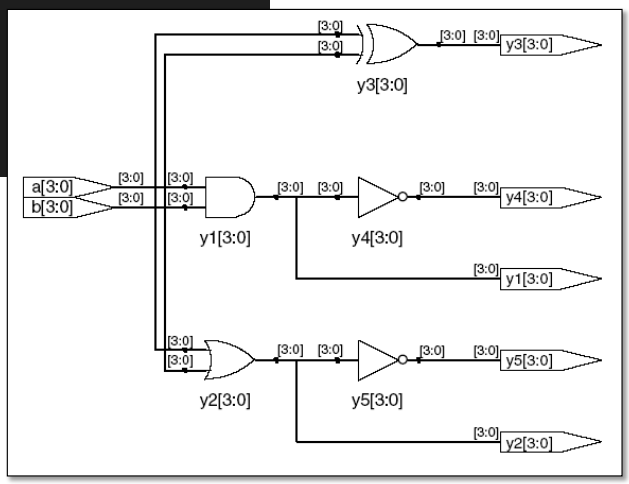

Combinational circuit is easiest to specify using behavioural specification with Boolean operators. You can also choose to provide structural description with interconnected gates as shown on the right.

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.

System Verilog: 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
```



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

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:

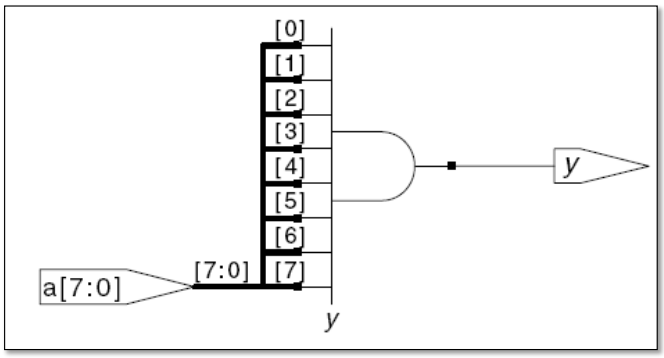
assign y1 = a & b;

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
```

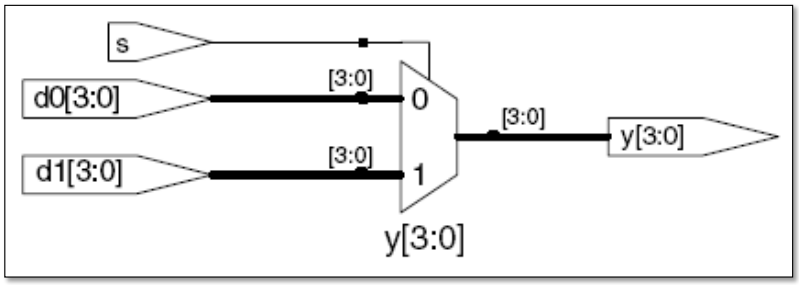


Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

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.

System Verilog: Conditional Assignment

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



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

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

cond ? True_value : False_value

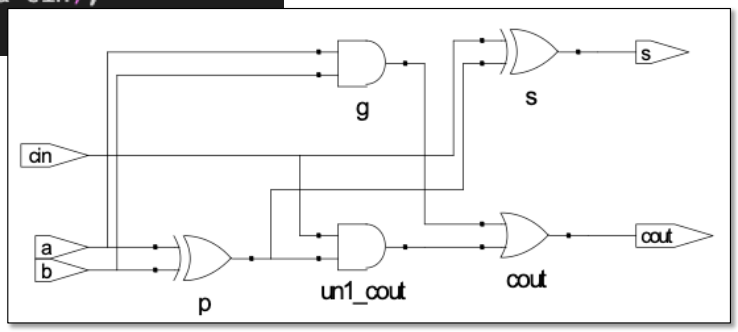
Therefore, assign y = s ? d1 : d0;

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

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

System Verilog: 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
```



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

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).

System Verilog: 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

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

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

System Verilog: 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

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

PYKC 28 Oct 2024 EE2 – Circuits & Systems Lecture 8 Slide 15

When using System Verilog 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.

System Verilog: 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. **System Verilog ignores them**.

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

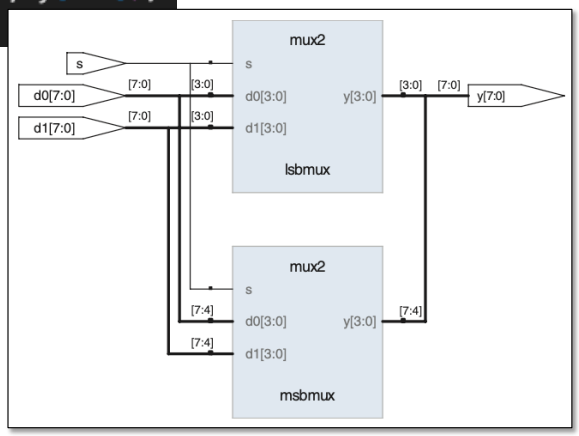
{ . } is called a **concatenation** operation. { 1, 0, 1, 1} forms a 4-bit number 4'b1011.

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.

System Verilog: 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
```



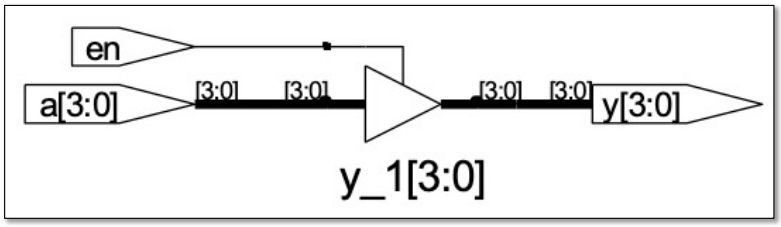
Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

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?

System Verilog: 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

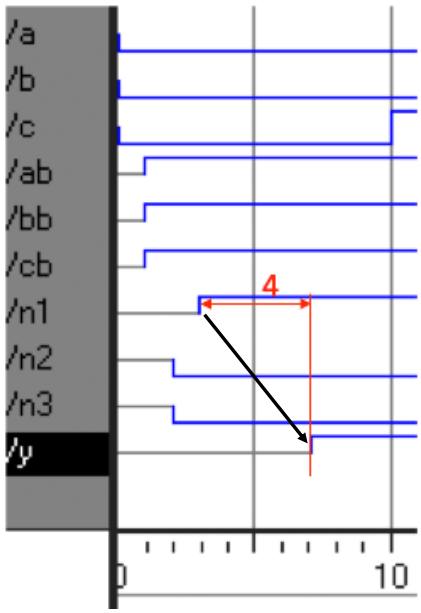
Based on: "*Digital Design and Computer Architecture (RISC-V Edition)*" by Sarah Harris and David Harris (H&H),

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 System Verilog 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.

System Verilog: 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.

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

Digital circuits have **delays**. System Verilog 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.

System Verilog: Sequential Logic

- System Verilog 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

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

Sequential logics are specified using the pattern:

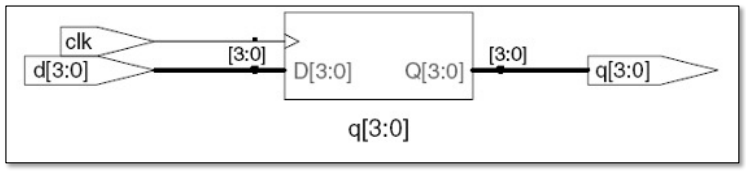
```
always @(sensitivity list)
        statement;
```

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.

System Verilog: 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
```



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

System Verilog 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.

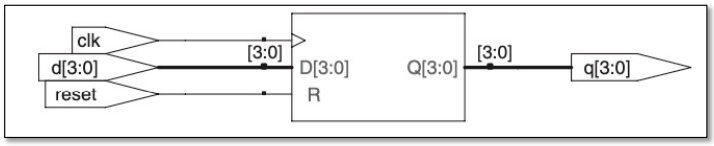
System Verilog: Resettable D Flip-Flop

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
```



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

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

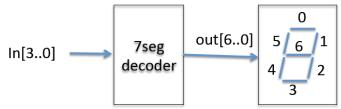
```
// combinational logic using an always statement
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
  always_comb        // need begin/end because there is
    begin            // more than one statement in always
      y1 = a & b;    // AND
      y2 = a | 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.

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

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.

Putting everything together – 7 seg decoder



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

In3 : in2

In1 : in0 \ out6	00	01	11	10
00	1	0	1	0
01	1	0	0	0
11	0	1	0	0
10	0	0	0	0

out6 = /in3*/in2*/in1 + in3*in2*/in1*/in0 + /in3*in2*in1*in0

out5 = /in3*/in2*in0 + /in3*/in2*in1 + /in3*in1*in0 + in3*in2*/in1*in0

out4 = /in3*in0 + /in3*in2*/in1 + in3*/in2*/in1*in0

out3 = /in3*in2*/in1*/in0 + /in3*/in2*/in1*in0 + in2*in1*in0 + /in2*in1*/in0

out2 = /in3*/in2*in1*/in0 + in3*in2*/in0 + in3*in2*in1

out1 = in3*in2*/in0 + /in3*in2*/in1*in0 + in3*in1*in0 + in2*in1*/in0

out0 = /in3*/in2*/in1*in0 + /in3*in2*/in1*/in0 + in3*in2*/in1*in0 + in3*/in2*in1*in0

Here is a simple example: the design of a 4-bit hex code to 7 segment decoder. You can express the function of this 7-segment decoder in three forms: 1) as a truth table (note that the segments are low active); 2) as 7 separate K-maps (shown here is for **out[6]** segment only); 3) as Boolean equations.

This is probably the last time you see K-maps. In practical digital design, you would rely heavily on CAD tools. In which case, logic simplifications are done for you automatically – you never need to use K-maps to do Boolean simplification manually!

Method 1: Schematic Entry Implementation

out6 = /in3*/in2*/in1 + in3*in2*/in1*/in0 + /in3*in2*in1*in0

out5 = /in3*/in2*in0 + /in3*/in2*in1 + /in3*in1*in0 + in3*in2*/in1*in0

out4 = /in3*in0 + /in3*in2*/in1 + in3*/in2*/in1*in0

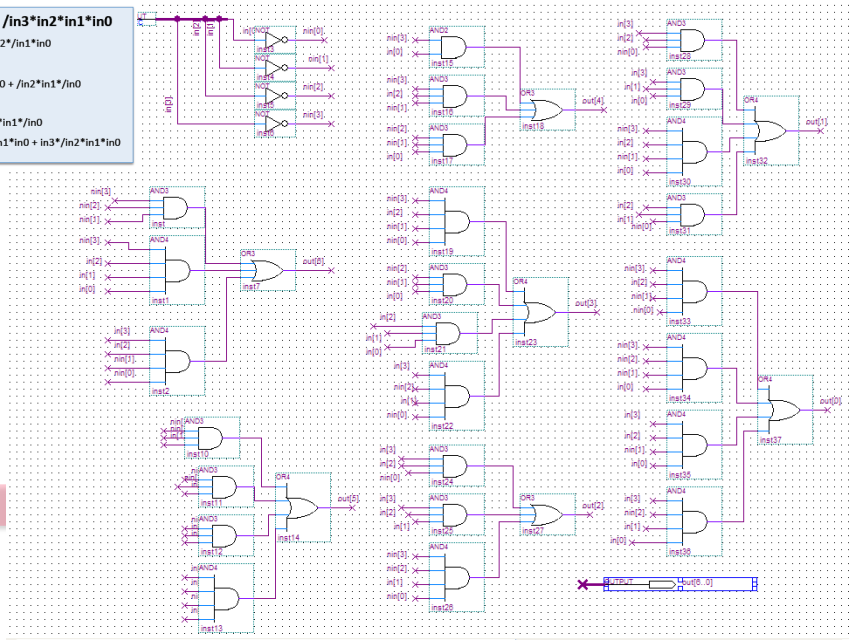
out3 = /in3*in2*/in1*/in0 + /in3*/in2*/in1*in0 + in2*in1*in0 + /in2*in1*/in0

out2 = /in3*/in2*in1*/in0 + in3*in2*/in0 + in3*in2*in1

out1 = in3*in2*/in0 + /in3*in2*/in1*in0 + in3*in1*in0 + in2*in1*/in0

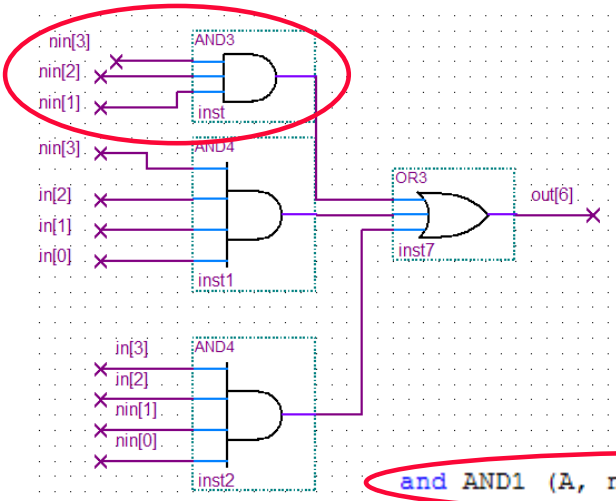
out0 = /in3*/in2*/in1*in0 + /in3*in2*/in1*/in0 + in3*in2*/in1*in0 + in3*/in2*in1*in0

TEDIOUS!!!!!



Here is a tedious implementation in the form of schematic diagram of the 7 segment decoder as interconnected gates. Very hard to do and very prone to errors.

Method 2: Use primitive gates in Verilog



Equally TEDIOUS!!!!!

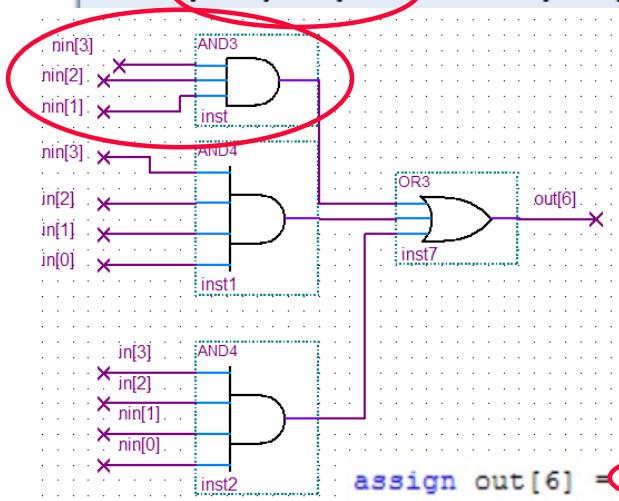
Direct mapping of gates to primitives

```
and AND1 (A, nin[3], nin[2], nin[1]);
and AND2 (B, nin[3], in[2], in[1], in[0]);
and AND3 (C, in[3], in[2], nin[1], nin[0]);
or  OR1  (out[6], A, B, C);
```


One could take a group of gates and specify the gates in Verilog gate primitives such as **and**, **or** etc. Still very tedious. Here is the implementation for the **out[6]** output.

Method 3: Use continuous assignment in Verilog

out6 = /in3*/in2*/in1 + in3*in2*/in1*/in0 + /in3*in2*in1*in0

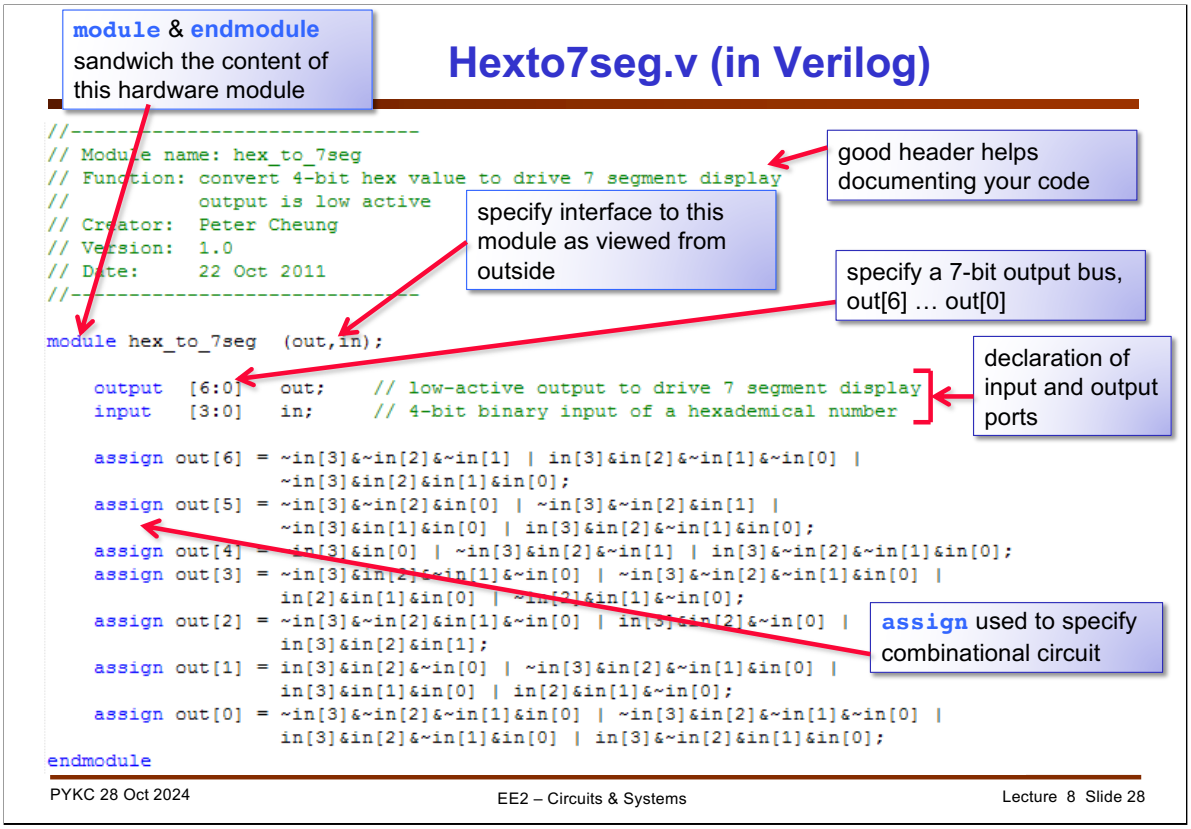


Much Better?

Direct mapping of Boolean equation using continuous assignment..

```
assign out[6] = ~in[3]&~in[2]&~in[1] |
                in[3]&in[2]&~in[1]&~in[0] |
                ~in[3]&in[2]&in[1]&in[0];
```


Instead of specifying each gate separately, here is using continuous assignment statement, mapping the **Boolean equation** direction to a single Verilog statement. This is better.



Here is the complete specification of the hex_to_7seg module using **continuous assignment** statements. It shows how one should write Verilog code with good comments and clear documentation of input and output ports.

Method 4: Power of behavioural abstraction

```
module hexto7seg    (
    output logic [6:0]  out,    // low-active
    input  logic [3:0]  in      // 4-bit binar
);
    always_comb
      case (in)
        4'h0: out = 7'b1000000;
        4'h1: out = 7'b1111001;     // -- 0 --
        4'h2: out = 7'b0100100;     // |
        4'h3: out = 7'b0110000;     // 5
        4'h4: out = 7'b0011001;     // |
        4'h5: out = 7'b0010010;     // -- 6 --
        4'h6: out = 7'b0000010;     // |
        4'h7: out = 7'b1111000;     // 4
        4'h8: out = 7'b0000000;     // |
        4'h9: out = 7'b0011000;     // -- 3 --
        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;
        default: out = 7'b0000000; // de
      endcase
endmodule
```

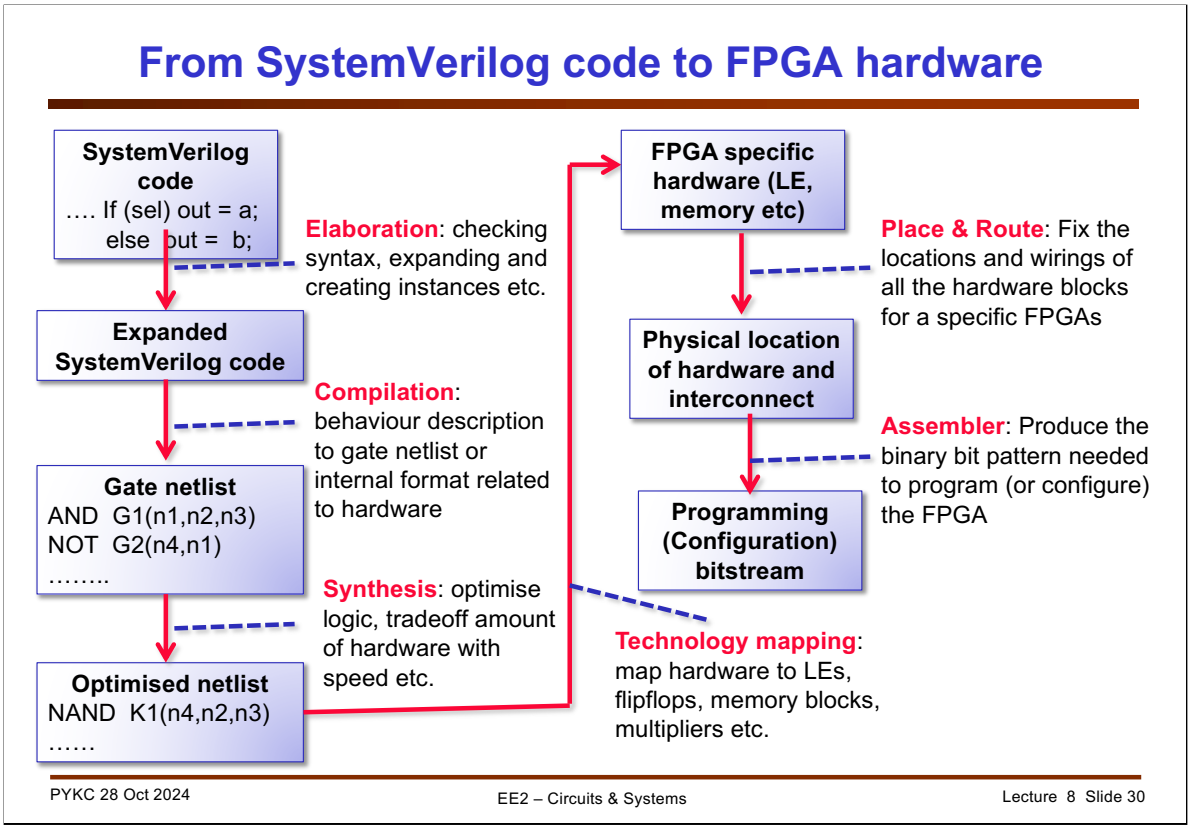
BEAUTIFUL !!!

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
1000	0000000	8
1001	0010000	9
1010	0001000	A
1011	0000011	b
1100	1000110	C
1101	0100001	d
1110	0000110	E
1111	0001110	F

- Direct mapping of truth table to case statement
- Close to specification, not implementation

Finally the 4th method is the best. We use the **case** construct to specify the behaviour of the decoder. Here one directly maps the truth table to **the case statement** – easy and elegant.

Instead of using: always @ (in), you could also use always @*



How is a Verilog description of a hardware module turned into FPGA configuration? This flow diagram shows the various steps taken inside the Quartus Prime CAD system.

Power of SystemVerilog: Integer Arithmetic

- Arithmetic operations make computation easy:

```
module  add32 (
    input  logic [31:0]     a,
    input  logic [31:0]     b,
    output logic [31:0]     sum
);
    assign sum = a + b;
endmodule
```

- Here is a 32-bit adder with carry-in and carry-out:

```
module  add32_carry (
    input  logic [31:0]     a,
    input  logic [31:0]     b,
    input  logic            cin,
    output logic [31:0]     sum,
    output logic            cout
);
    assign {cout, sum} = a + b + cin;
endmodule
```


Verilog is very much like C. However, the declaration of **a, b** and **sum** in the **module add32** specifies the data width (i.e. number of bits in each signal **a**, **b** or **sum**). This is often known as a "**vector**" or a "**bus**". Here the data width is 32-bit, and it is ranging from bit 31 down to bit 0 (e.g. **sum[31:0]**).

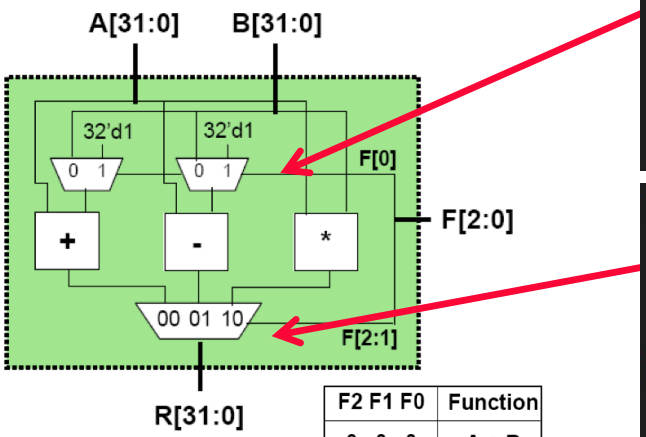
You can refer to individual bits using the index value. For example, the least-significant bit (LSB) of sum is **sum[0]** and the most-significant bit (MSB) is **sum[31]. sum[7:0]** refers the the least-significant byte of **sum**.

The '+' operator can be used for signals of any width. Here a 32-bit add operation is specified. **sum** is also 32-bit in width. However, if **a** and **b** are 32-bit wide, the sum result could be 33-bit (including the carry out). Therefore this operation could result in a wrong answer due to **overflow** into the carry bit. The 33th bit is truncated.

The second example **module add32_carry** shows the same adder but with carry input and carry output. Note the LHS of the **assign** statement. The **{cout, sum}** is a **concatenation** operator – the contents inside the brackets **{ }** are concatenated together, with **cout** is assigned the MSB of the 33th bit of the result , and the remaining bits are formed by **sum[31:0]**.

A larger example – 32-bit ALU in SV

- Here is an 32-bit ALU with 5 simple instructions:



F2 F1 F0	Function
0 0 0	A + B
0 0 1	A + 1
0 1 0	A - B
0 1 1	A - 1
1 0 X	A * B

```
module  mux2to1 (
    input  logic [31:0]     i0,
    input  logic [31:0]     i1,
    input  logic            sel,
    output logic [31:0]     out
);
    assign out = sel ? i1 : i0;
endmodule
```

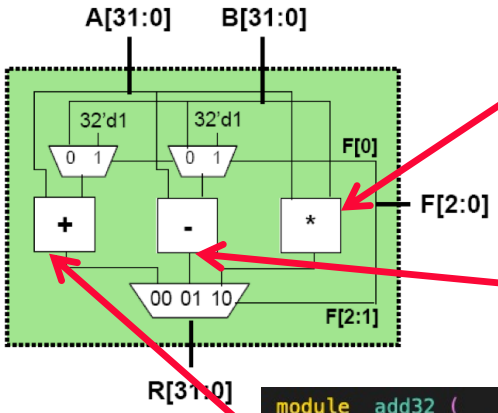
```
module  mux3to1 (
    input  logic [31:0]     i0,
    input  logic [31:0]     i1,
    input  logic [31:0]     i3,
    input  logic [1:0]      sel,
    output logic [31:0]     out
);
    always_comb
        case (sel)
            2'b00:  out = i0;
            2'b01:  out = i1;
            2'b10:  out = i2;
            default:  out = 32'bx;
        endcase
endmodule
```


Now let us put all you have learned together in specifying (or designing) a 32-bit ALU in Verilog.

There are five operators in this ALU. We assume that there are three arithmetic blocks, and three multiplexers (two 2-to-1 MUX and one 3-to-1 MUX).

The arithmetic modules

- Here is an 32-bit ALU with 5 simple instructions:




```
module  mul16 (
    input  logic [15:0]     i0,
    input  logic [15:0]     i1,
    output logic [31:0]     pro
);
    assign  prod = i0 * i1;
endmodule
```

```
module  sub32 (
    input  logic [31:0]     i0,
    input  logic [31:0]     i1,
    output logic [31:0]     diff
);
    assign diff = i0 - i1;
endmodule
```

```
module  add32 (
    input  logic [31:0]     i0,
    input  logic [31:0]     i1,
    output logic [31:0]     sum
);
    assign sum = i0 + i1;
endmodule
```


Each hardware block is defined as a Verilog module. So we have the following modules:

mux32two – a 32-bit multiplexer that has TWO inputs

mux32three – a 32-bit multiplexer that has THREE inputs

mul16 – a 16-by-16 binary multiplier that produces a 32-bit product

add32 – a 32-bit binary adder

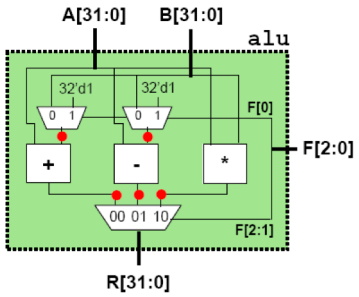
sub32 – a 32-bit binary subtractor

Top-level module – putting them together

- Given submodules:

```
module mux2to1 (i0, i1, sel, out);
module mux3to1 (i0, i1, i2, sel, out);
module add32 (i0, i1, sum);
module sub32 (i0, i1, diff);
module mul16 (i0, i1, prod);
```

```
module alu (
    input  logic [31:0]     a,
    input  logic [31:0]     b,
    input  logic [2:0]      f,
    output logic [31:0]     r
);
    logic [32:0]    addmux_out, submux_out;
    logic [32:0]    add_out, sub_out, mul_out;

    mux2to1 adder_mux (b, 32'd1, f[0], addmux_out);
    mux2to1 sub_mux   (b, 32'd1, f[0], submux_out);
    add32   our_adder (a, addmux_out, add_out);
    sub32   out_sub   (a, submux_out, sub_out);
    mul16   our_mult  (a[15:0], b[15:0], mul_out);
    mux3to1 output_mux(add_out, sub_out, mul_out, f[2:1], r);
endmodule
```



Now let us put all these together.

Note that **mxu32two** is being used twice and therefore this **is instantiated** two times with two different **instance names**: **adder_mux** and **sub_mux**.

Connections between modules are implicit through the use of **signal names**. For example, the 16-bit inputs to the multiplier are taken from the lower 16-bits of **a** and **b** inputs (i.e. **a[15:0]** and **b[15:0]**).