Digital Systems Design and Laboratory Spring 2019 Lab 1

Tzu-Hsu Yu
e841018@gmail.com
2019/04/15

Sample code:
https://drive.google.com/open?id=1NHDGHVTfOcuYkMBfPBUKFsD-87m5EqO-

Agenda

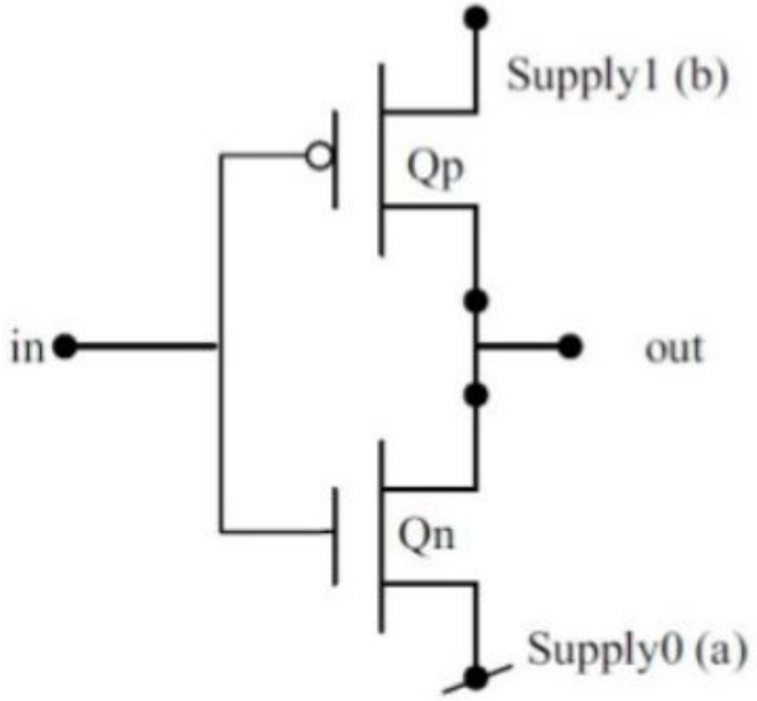
- Introduction
- Verilog syntax
- Simulator installation
- Demonstration
- Assignment

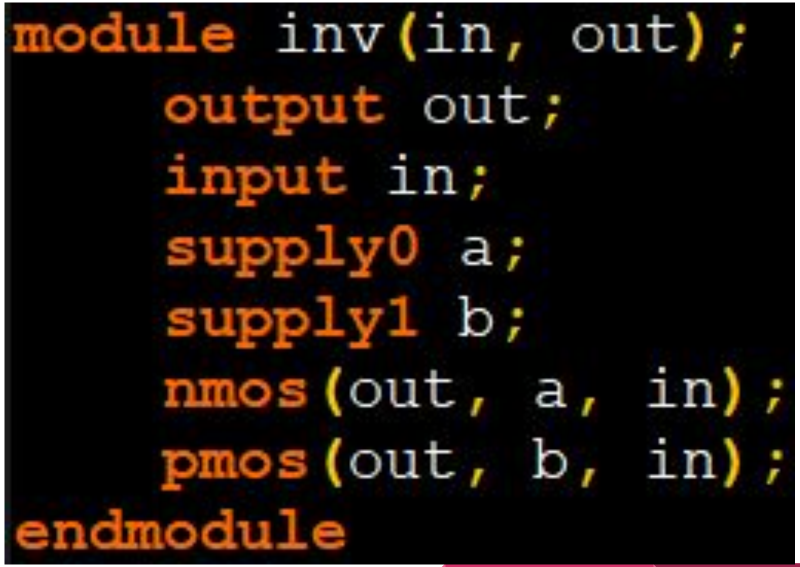
Introduction

Hardware description language (HDL)

- Different from programming languages, they don’t really “execute” at run time
- They describe hardware at different abstract levels
- Popular languages: Verilog, VHDL
- Logic synthesis: First convert HDL code into a netlist, and then place and route them to generate a set of masks (for IC) or a list of mapping and interconnections (for FPGA/CPLD).
- Simulation: use a test bench to check if the behavior meets your requirements

Verilog: switch(transistor) level modeling

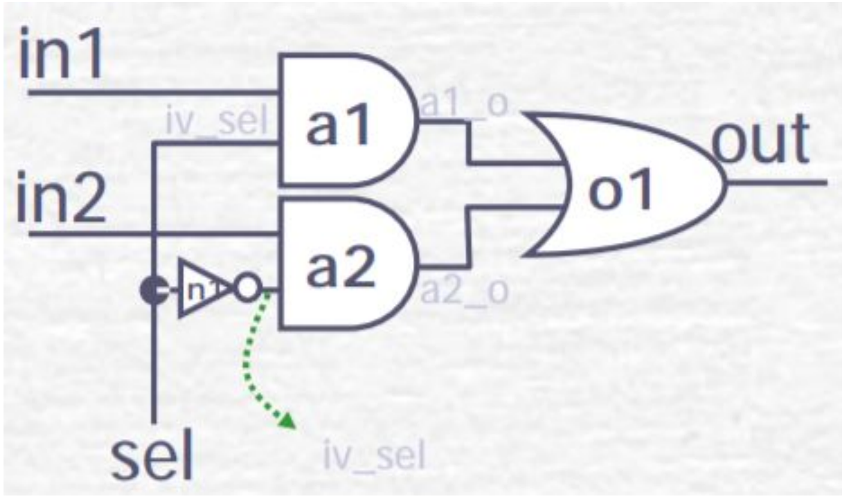


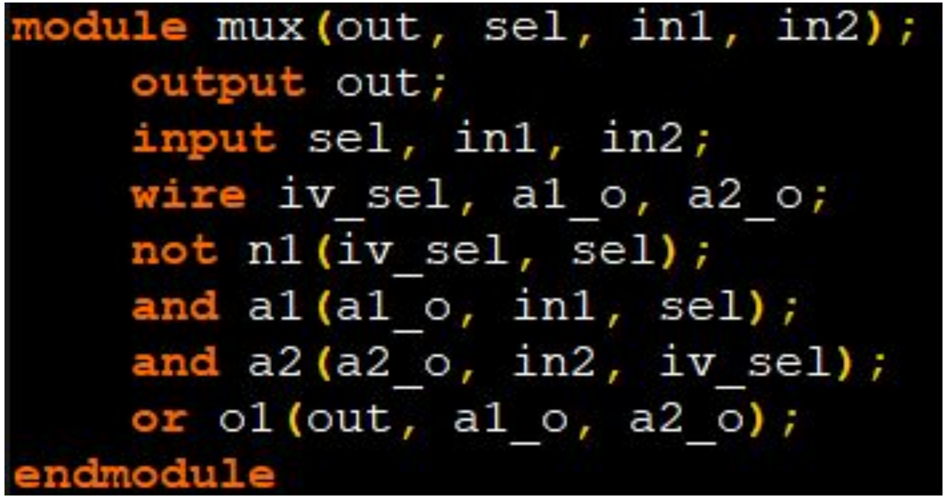


```
module inv(in, out);
    output out;
    input in;
    supply0 a;
    supply1 b;
    nmos(out, a, in);
    pmos(out, b, in);
endmodule
```

https://www.slideshare.net/pradeepdevip/switch-level-modeling

Verilog: gate level modeling



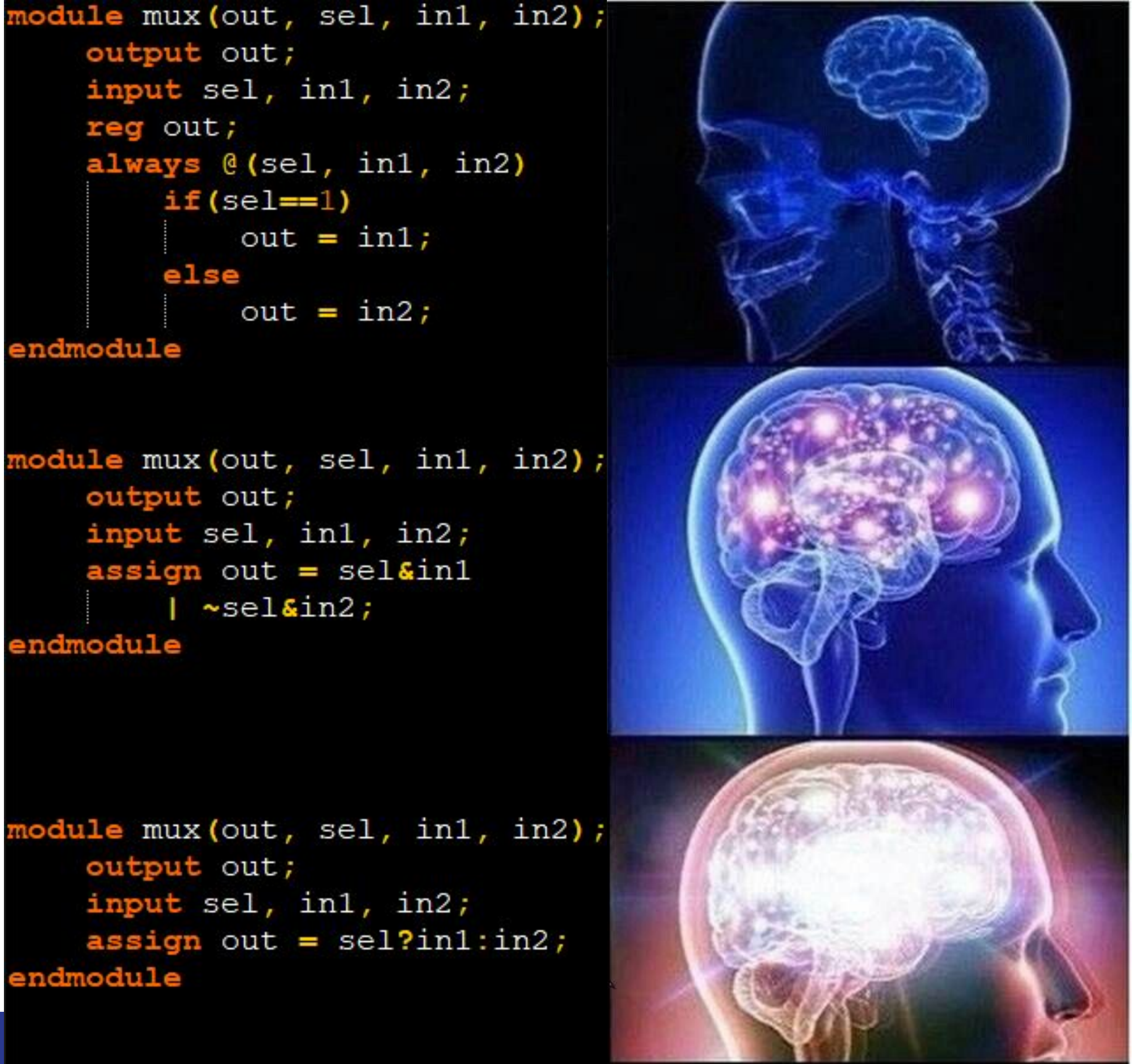

```
module mux(out, sel, in1, in2);
    output out;
    input sel, in1, in2;
    wire iv_sel, a1_o, a2_o;
    not n1(iv_sel, sel);
    and a1(a1_o, in1, sel);
    and a2(a2_o, in2, iv_sel);
    or o1(out, a1_o, a2_o);
endmodule
```

http://access.ee.ntu.edu.tw/course/logic_design_94first/941%20Verilog%20HDL(Gate%20Level%20design).pdf

Verilog:

behavioral level modeling (Register-Transfer Level, RTL)

http://access.ee.ntu.edu.tw/course/logic_design_94first/941%20Verilog%20HDL(Gate%20Level%20design).pdf



```
module mux(out, sel, in1, in2);
    output out;
    input sel, in1, in2;
    reg out;
    always @(sel, in1, in2)
        if(sel==1)
            out = in1;
        else
            out = in2;
endmodule
```

```
module mux(out, sel, in1, in2);
    output out;
    input sel, in1, in2;
    assign out = sel&in1
          | ~sel&in2;
endmodule
```

```
module mux(out, sel, in1, in2);
    output out;
    input sel, in1, in2;
    assign out = sel?in1:in2;
endmodule
```

Verilog Syntax

Module

- Module: basic functional unit, can be instantiated in other modules

-
  ```
  module <name>(<signal0>, <signal1>, <signal2>);
      output <signal0>;
      input <signal1>, <signal2>;
      // implementation
  endmodule
  ```

-
  ```
  module <name>(output <signal0>, input <signal1>);
      reg <signal0>;
      // implementation
  endmodule
  ```

Data types

- `reg`: driven in a process block, may become **a net or a register** after synthesis
- `wire`: driven outside a process block, becomes **a net** after synthesis
- Type of `input` and `output` is `wire` unless specified
- `integer`: default to be 32 bit **signed**, usually used in test bench
- `time`: equivalent to `reg[63:0]`

https://en.wikibooks.org/wiki/Programmable_Logic/Verilog_Data_Types

Integer literals

● Binary(2):	`4’b1011`
● Octal(8):	`4’o13`
● Hexadecimal(16):	`4’hb`
● Decimal(10):	`4’d11 == -4’d5`
● Concatenation:	`{2’b10, 2’b11} == 4’b1011`

http://web.engr.oregonstate.edu/~traylor/ece474/beamer_lectures/verilog_number_literals.pdf

Concurrent and Sequential

- Concurrent statement: “executed” at the same time
  `wire a, b; assign b = a; assign c = b;`
  Connect wire `b` to wire `a`, and then connect wire `c` to wire `b`
  If `a` changes from `0` to `1`, `b` and `c` will change at the same instant
- Sequential statement: “executed” one by one like programming languages, but actually converted into equivalent concurrent statements
  `reg a, b, c; b = a; c = b;`
  [reg a]-->[reg b]-->[reg c], if output of register `a` changes, `b` changes before `c`
- Sequential statements are only allowed in a **process block**

Assignment: continuous assignment, blocking and non-blocking assignment

- Outside a process block:
```
assign x = y & z; // continuous assignment
// same as and a0(x, y, z), x cannot be a reg
```
- Inside a process block:
```
x = y;  // blocking assignment
a <= b; // non-blocking assignment
b <= a; // a and b exchanges their values
```

Arrays

- Unpacked array (array):

```
reg an_unpacked_array[2:0];
```

- Packed array (vector):

```
reg[5:0] a_packed_array;
reg[0:5] a_packed_array; // same
reg[3:5] a_packed_array; // it's legal syntax!
```

- Icarus Verilog only supports 1-dimensional arrays

```
reg[5:0][4:0] two_dimensional_array[3:0][2:0]
// syntax error in Icarus Verilog
```

https://verificationacademy.com/forums/ovm/difference-between-packed-and-unpacked-arrays

Indexing arrays

```
bit[3:0][4:0] how_to_index[1:0][2:0];
      2                         0   1
```

```
how_to_index[0][1][2] = 5'b00010;
```

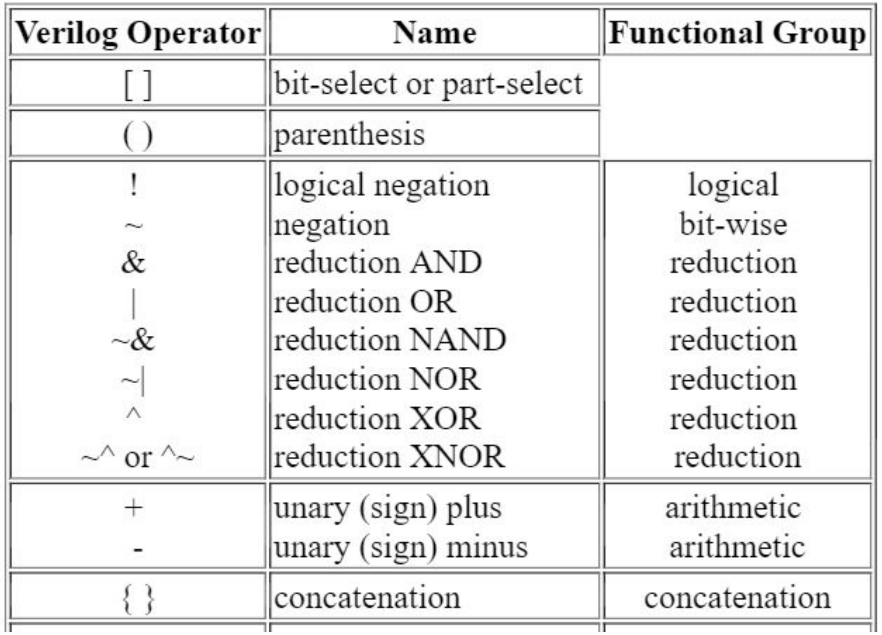
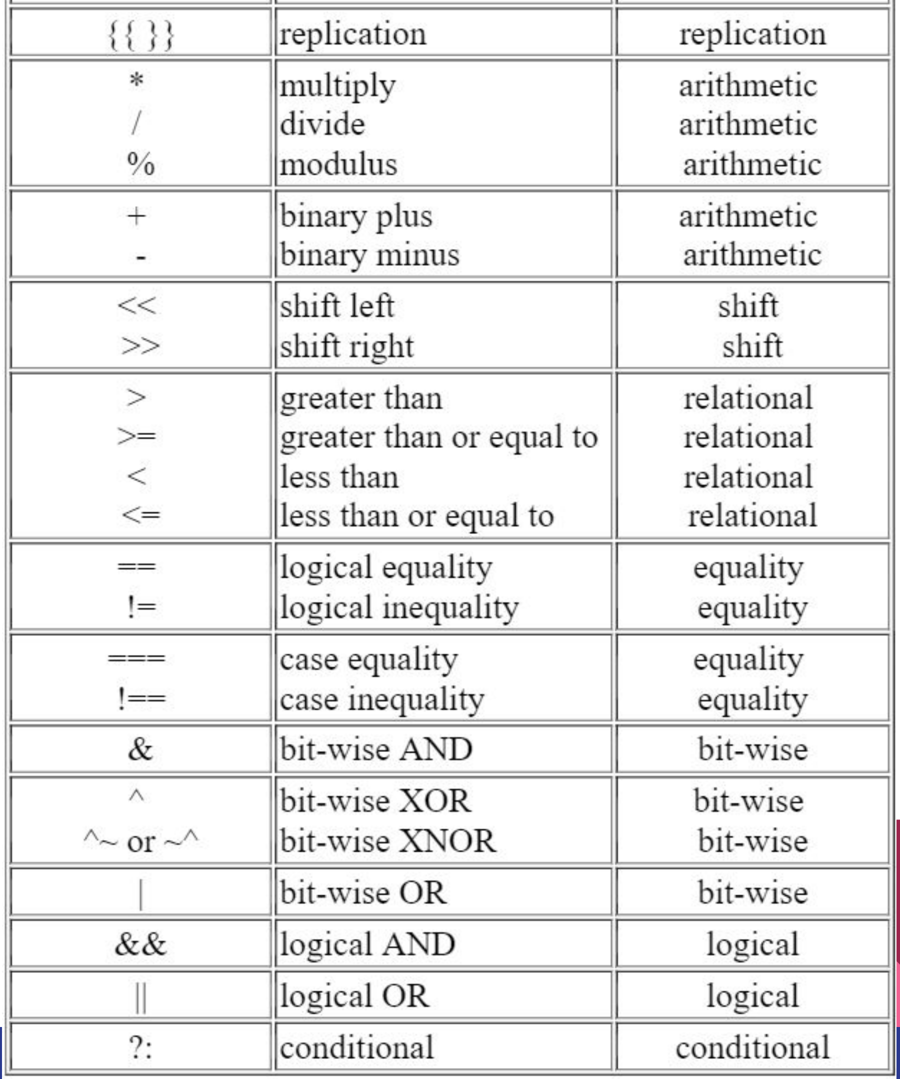
```
   0                     1                     2
0  00000000000000000000  00000000000000010000  00000000000000000000
   01234012340123401234  01234012340123401234  01234012340123401234

1  00000000000000000000  00000000000000000000  00000000000000000000
   01234012340123401234  01234012340123401234  01234012340123401234
```

https://verificationacademy.com/forums/ovm/difference-between-packed-and-unpacked-arrays

Operators

`a+=1`, `a++`:

no such syntax, use a = a+1 instead

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
|	reduction OR	reduction
~&	reduction NAND	reduction
~|	reduction NOR	reduction
^	reduction XOR	reduction
~^ or ^~	reduction XNOR	reduction
+	unary (sign) plus	arithmetic
-	unary (sign) minus	arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic
+	binary plus	arithmetic
-	binary minus	arithmetic
<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal to	relational
<	less than	relational
<=	less than or equal to	relational
==	logical equality	equality
!=	logical inequality	equality
===	case equality	equality
!==	case inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
^~ or ~^	bit-wise XNOR	bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

https://class.ece.uw.edu/cadta/verilog/operators.html

Process block: always and initial

-
  ```
  always begin
      <statements>
  end
  ```

-
  ```
  always
      <single statement>
  ```
- Statements in `always` will be executed from time 0 and repeated forever
- `initial` is the same as `always` except that it only executes once

Process block: sensitivity list

- If `always` is followed by a `@`, the process will be executed once whenever the expressions in the list changes, instead of repeating from time 0
- `always @(a or b) // old syntax`
- `always @(a, b)`
- `always @(a, posedge clk, negedge enable)`
- `always @a`
- `always @*`

Process block: flow control

-
  ```
  if(condition) begin
      <statements>
  end
  else begin
      <statements>
  end
  ```

-
  ```
  for() begin
      <statements>
  end
  ```
- `while, case also supported`

Delay

Outside a process block:

- `assign #10 a = b+c;`

  The adder has 10 time units of propagation delay

Inside a process block:

- `#10;`

  Delay 10 units
- `#10 a = b+c;`

  Delay 10 units of time and evaluate `b+c`, assign it to `a`
- `a = #10 b+c;`

  Evaluate `b+c` and execute the next statement, assign 10 time units later

http://content.inflibnet.ac.in/data-server/eacharya-documents/53e0c6cbe413016f23443704_INFIEP_33/7/LM/33-7-LM-V1-S1__delay_modeling.pdf

Test bench: macros and system tasks

- `include <module>
- `define <parameter> <value>
- `timescale <unit>/<precision>
- $dumpfile("some_file.vcd");
- $dumpvars(<level>, <module>);
  level=0: variables in all levels
  level=1: variables in <module> only
  level=2: variables in <module> and one level below it
- $display(), $write(), $monitor()

Quick guides for Verilog

- Summary of Synthesisable Verilog 2001 (2 pages) https://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/files/verilogcheatsheet.pdf
- Quick Reference for Verilog HDL (25 pages) http://ece.eng.umanitoba.ca/undergraduate/ECE3610/Verilog%20Notes/VerilogQuickRef.pdf
- Language reference: http://verilog.renerta.com/

Simulator Installation

Icarus Verilog and GTKWave

Reference:
https://www.swarthmore.edu/NatSci/mzucker1/e15_f2014/iverilog.html

For Windows users

http://bleyer.org/icarus/iverilog-0.9.7_setup.exe

Reminder: Run the installer as administrator!

For Mac and Linux users

Some students successfully installed with the default package managers of their OSes.

Commands and file extensions

- Compile:

  ```
  > iverilog [-o compiled_file.vpp] source_file.v
  ```

  (if no -o option provided, a.out will be generated)
- Simulate:

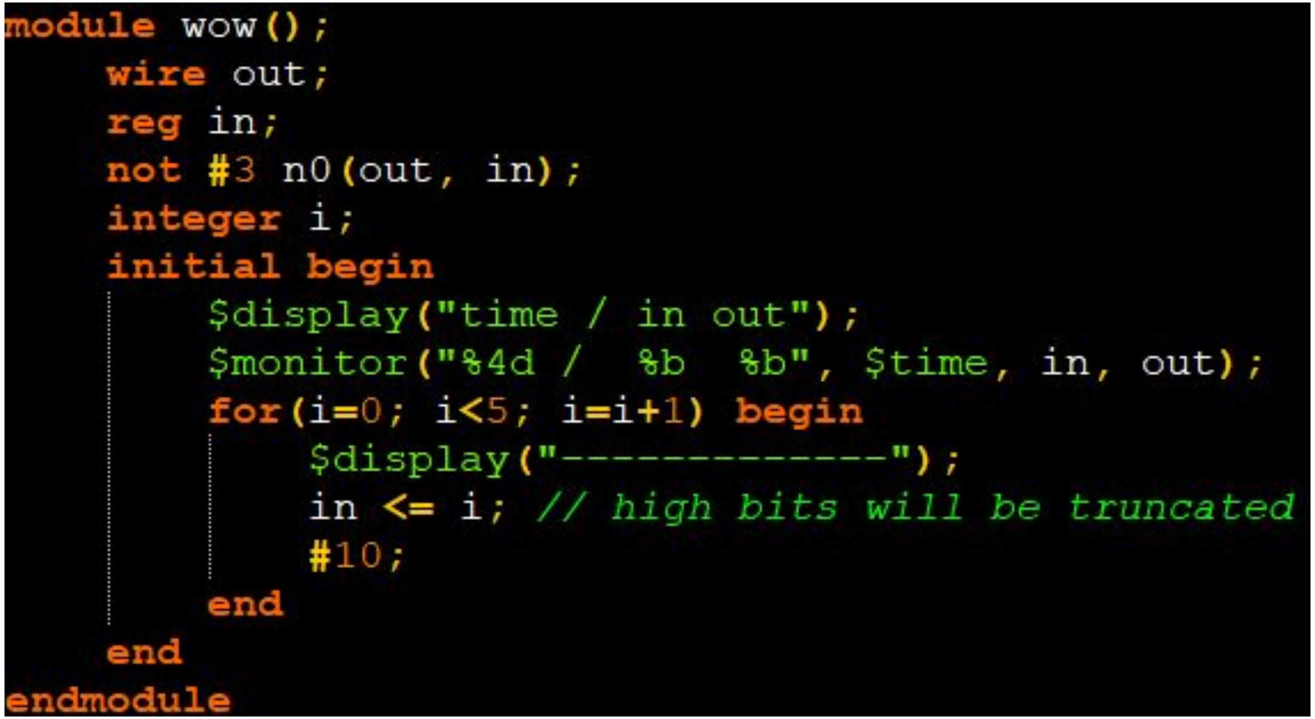
  ```
  > vvp compiled_file.vvp
  ```

  (a dump file will be generated if $dumpfile(“dump_file.**vcd**”) is called)
  (vcd stands for Value Change Dump)
- View waveform (optional):
  Open a new cmd window

  ```
  > gtkwave
  ```

  Open dump_file.**vcd** in GUI

Hello world


```
module wow();
    wire out;
    reg in;
    not #3 n0(out, in);
    integer i;
    initial begin
        $display("time / in out");
        $monitor("%4d /  %b  %b", $time, in, out);
        for(i=0; i<5; i=i+1) begin
            $display("-------------");
            in <= i; // high bits will be truncated
            #10;
        end
    end
endmodule
```

save as wow.v

```
> iverilog -o wow.vvp wow.v && vvp wow.vvp
```

Trouble shooting

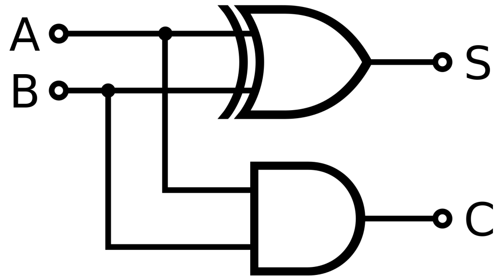
For windows users:

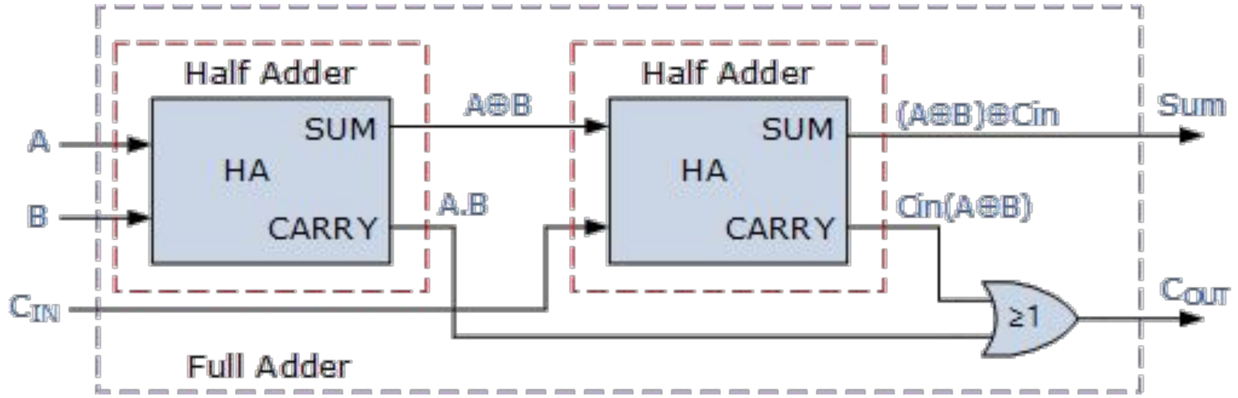
- If you can’t run `iverilog` on command line, uninstall and run the installer as administrator.
- If an error message pop up saying that some dll is missing, or the compiler executed but didn’t generate any output file, try the latest version: http://bleyer.org/icarus/iverilog-v11-20190327-x64_setup.exe

Demonstration

Half adder and full adder

HA: `assign {C, S} = A+B;` FA: `assign {Cout, S} = A+B+Cin;`



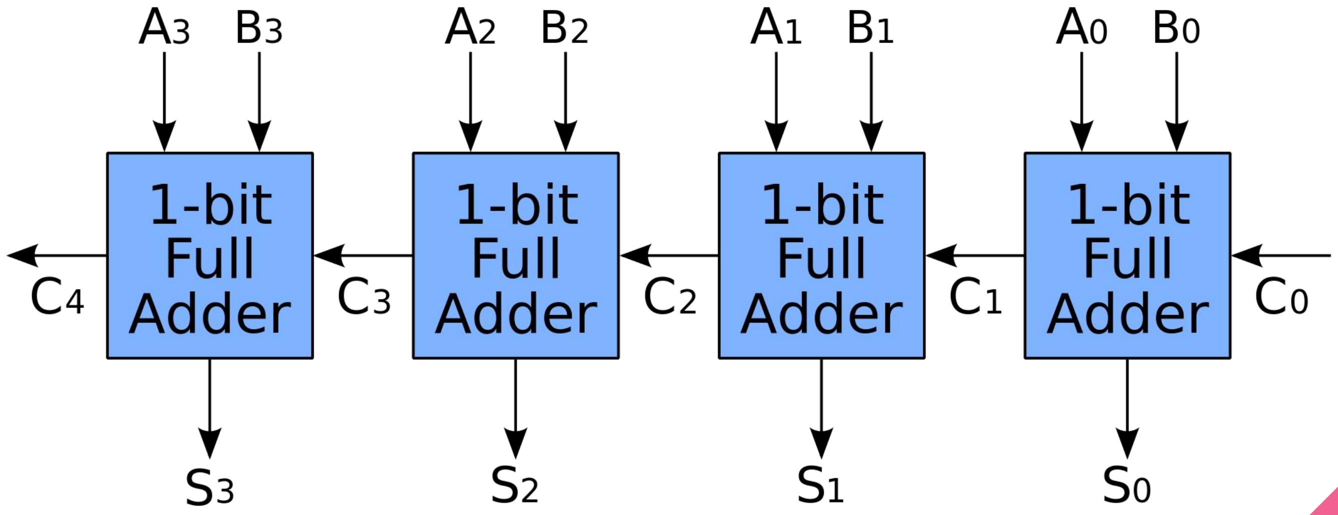


https://en.wikipedia.org/wiki/Adder_(electronics)
https://www.electronics-tutorials.ws/combination/comb_7.html

Ripple-carry adder

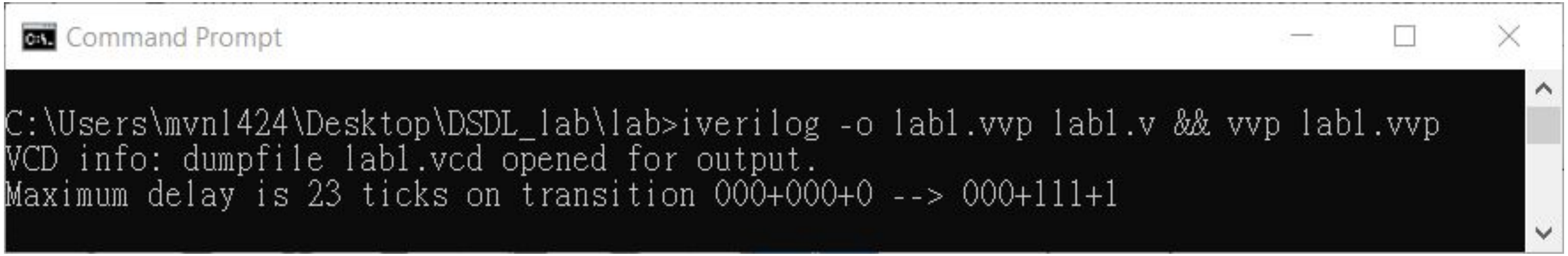
Notice that carry_n depends on carry_{n-1}

Where is the longest propagation path?



https://en.wikipedia.org/wiki/Adder_(electronics)

Ripple-carry adder: maximum delay

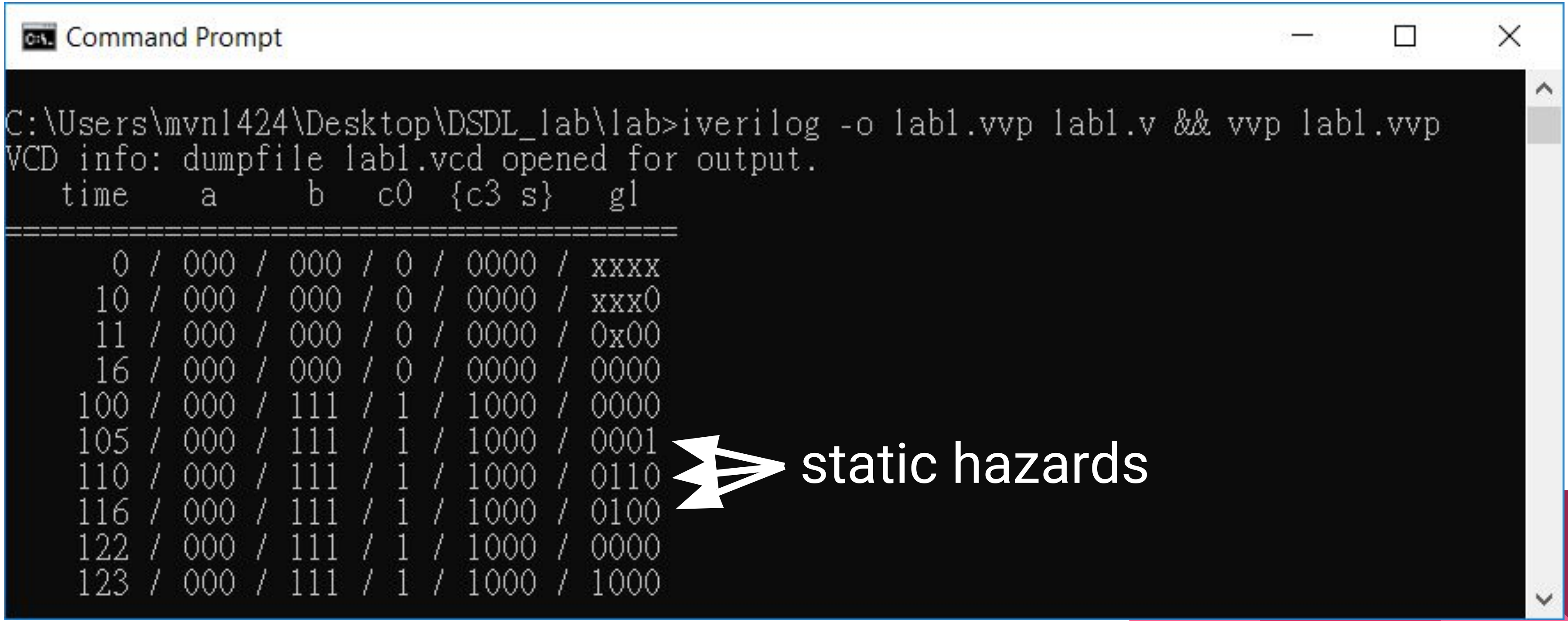


```
C:\Users\mvn1424\Desktop\DSDL_lab\lab>iverilog -o lab1.vvp lab1.v && vvp lab1.vvp
VCD info: dumpfile lab1.vcd opened for output.
Maximum delay is 23 ticks on transition 000+000+0 --> 000+111+1
```

Notice that 000+000+0 → 111+000+1 should cause the same delays.

https://en.wikipedia.org/wiki/Adder_(electronics)

Ripple-carry adder: transition



Command Prompt

```
C:\Users\mvn1424\Desktop\DSDL_lab\lab>iverilog -o lab1.vvp lab1.v && vvp lab1.vvp
VCD info: dumpfile lab1.vcd opened for output.
   time     a       b    c0   {c3 s}    gl
==========================================
      0 / 000 / 000 / 0 / 0000 /  xxxx
     10 / 000 / 000 / 0 / 0000 /  xxx0
     11 / 000 / 000 / 0 / 0000 /  0x00
     16 / 000 / 000 / 0 / 0000 /  0000
    100 / 000 / 111 / 1 / 1000 /  0000
    105 / 000 / 111 / 1 / 1000 /  0001
    110 / 000 / 111 / 1 / 1000 /  0110
    116 / 000 / 111 / 1 / 1000 /  0100
    122 / 000 / 111 / 1 / 1000 /  0000
    123 / 000 / 111 / 1 / 1000 /  1000
```

https://en.wikipedia.org/wiki/Adder_(electronics)

Assignment

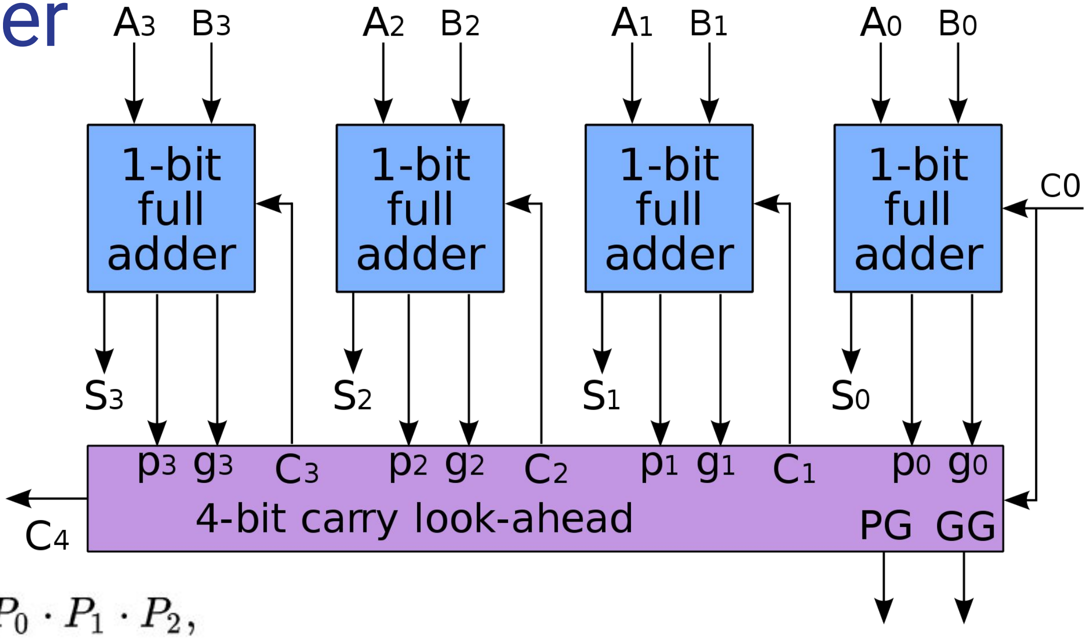
Carry-lookahead adder

$G_i = A_i \cdot B_i.$

Pi can be either:

$P_i = A_i \oplus B_i,$

$P_i = A_i + B_i.$



$C_1 = G_0 + P_0 \cdot C_0,$

$C_2 = G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1,$

$C_3 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2,$

$C_4 = G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3.$

PG and GG don’t have to be implemented in the assignment.

https://en.wikipedia.org/wiki/Carry-lookahead_adder

Requirements

- Implement a 3-bit adder **`adder_rtl`** using RTL modeling.
- Implement a 3-bit carry-lookahead adder **`cla_gl`** using gate-level modeling, only the gates provided in `gates.v` (with delays) are allowed to use.
- Print out your source codes and attach it with your Homework 3.
- Show the waveform of `cla_gl.S[2:0]` on input transitions:
  (1) from 000 + 000 + 0 to 001 + 000 + 1
  (2) from 111 + 000 + 0 to 000 + 111 + 0
- Find the maximum propagation delay of `cla_gl` and one of the corresponding input transitions.
- Assume that only 2-input gates are used. Derive the number of levels needed in an n-bit carry-lookahead adder as a function of n.

Hints

- **`adder_rtl`** should be simple to implement, I did it in one line.
- The outputs of **`adder_rtl`** and **`cla_gl`** should be the same at steady state. If they're different, maybe there's some mistake in **`cla_gl`** since it's complicated.
- Implement your adders in `adders.v`. The output and input signals are given.
- Do not modify `gates.v`.
- Only minor changes should be done to `lab1.v`.
- Use system tasks and GTKWave to debug.

- Feel free to email me for any questions. (address on the first page)
- And please remember to bring your laptop at Lab2!