Digital Systems Design and Laboratory
[Lab 1. Combinational Circuit Design]

Chung-Wei Lin

cwlin@csie.ntu.edu.tw

CSIE Department

National Taiwan University

Spring 2020

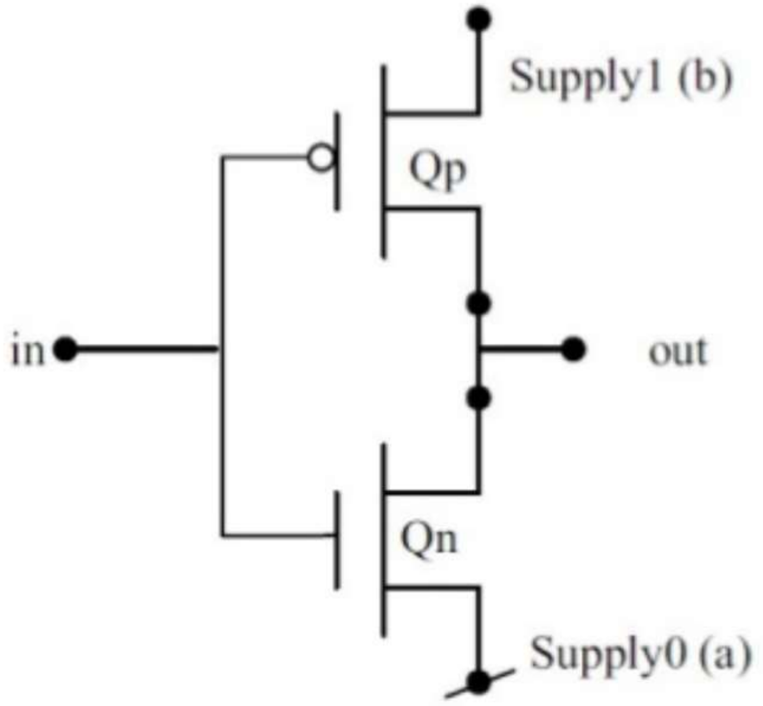
Outline

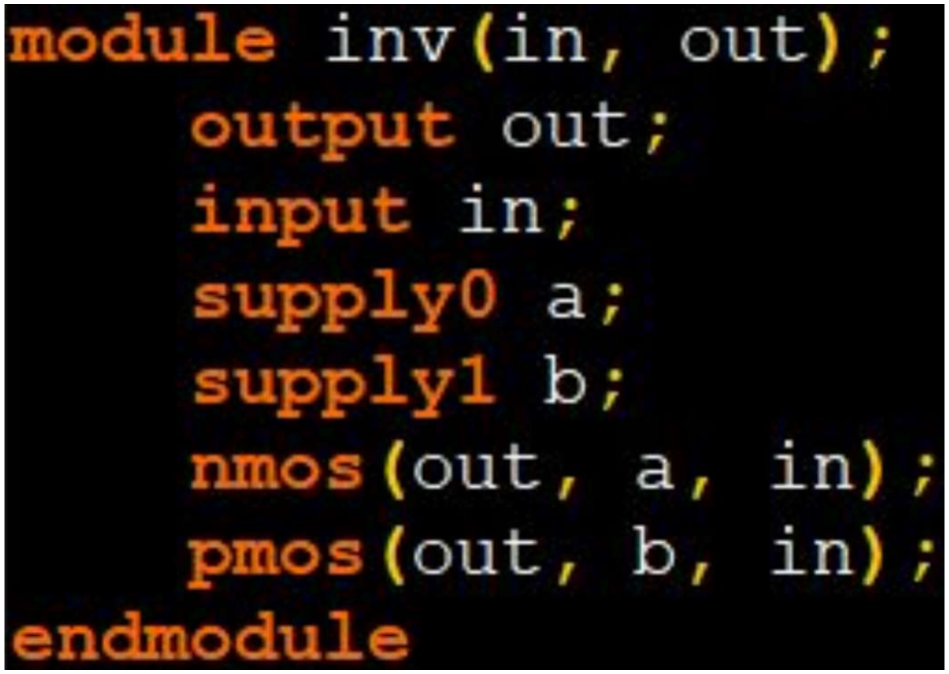
Hardware Description Language (HDL)

- Different from programming languages, they do not 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
  - 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 Level Modeling

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

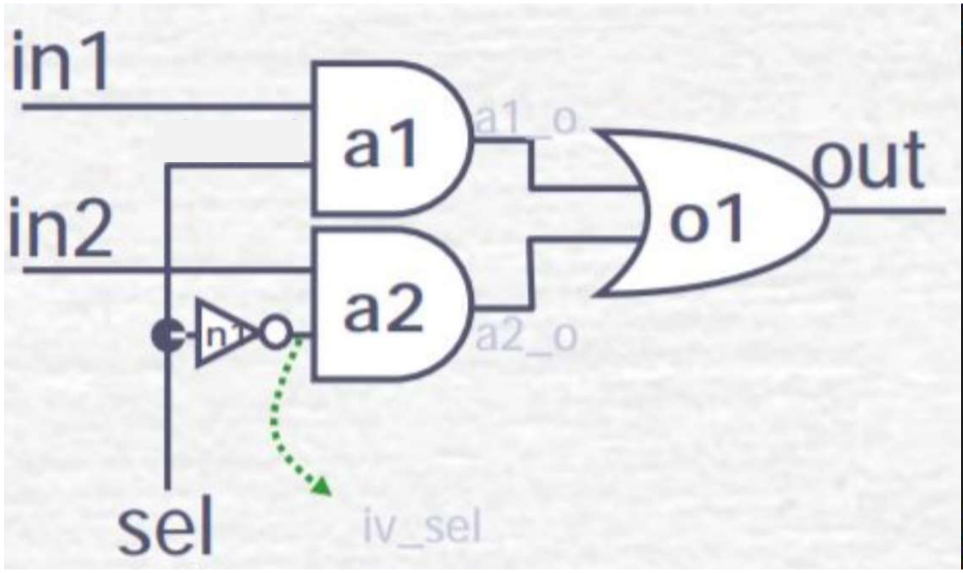


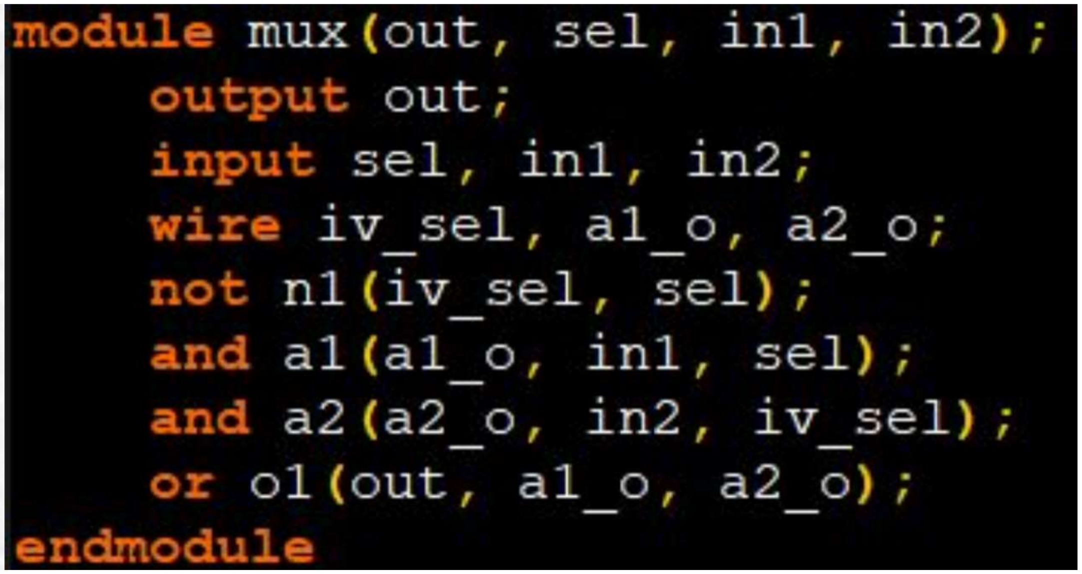


```
module inv(in, out);
    output out;
    input in;
    supply0 a;
    supply1 b;
    nmos(out, a, in);
    pmos(out, b, in);
endmodule
```

Verilog: Gate Level Modeling

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

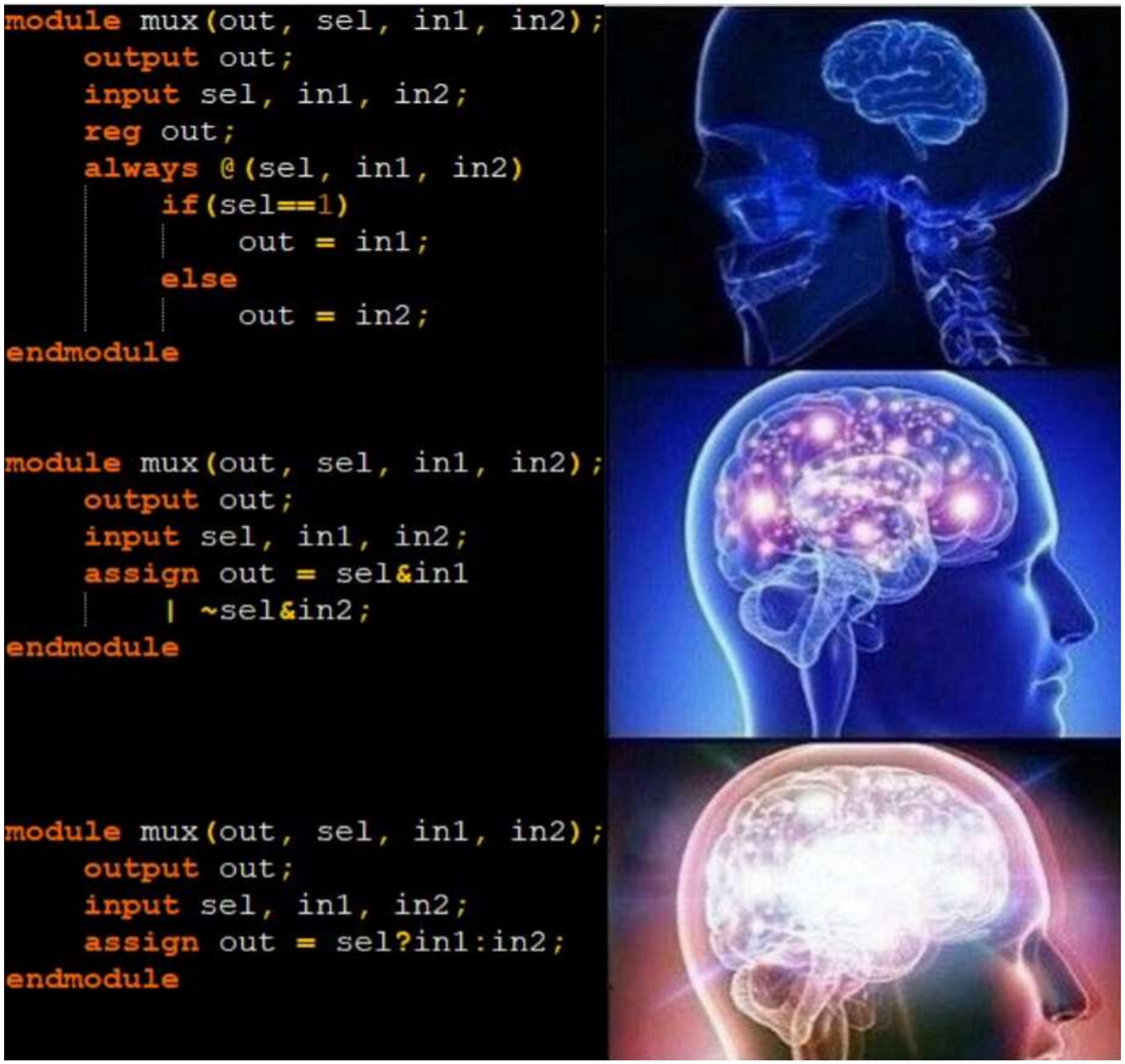




```
module mux(out, sel, in1, in2);
    output out;
    input sel, in1, in2;
    wire iv_sel, a1_o, a2_o;
    not n1(iv_sel, sel);
    and a1(a1_o, in1, sel);
    and a2(a2_o, in2, iv_sel);
    or o1(out, a1_o, a2_o);
endmodule
```

Verilog: Behavioral Level Modeling

- Register-Transfer Level (RTL)



```
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
```

Outline

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]`**
- ❑ More details
  - ➢ 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`
- More details
  - http://web.engr.oregonstate.edu/~traylor/ece474/beamer_lectures/verilog_number_literals.pdf

Concurrent and Sequential Statements

- Concurrent statement
  - "Executed" at the same time

    ```
    wire a, b, c; 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`**] $\rightarrow$ [**`reg b`**] $\rightarrow$ [**`reg c`**]
    - If output of register **a** changes, **b** changes before **c**
  - Sequential statements are only allowed in a process block

Assignments

- Continuous assignment
  - Outside a process block

```
assign x = y & z; // continuous assignment
// same as and a0(x, y, z), x cannot be a reg
```

- Blocking and non-blocking assignment
  - 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
```

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

Indexing Arrays

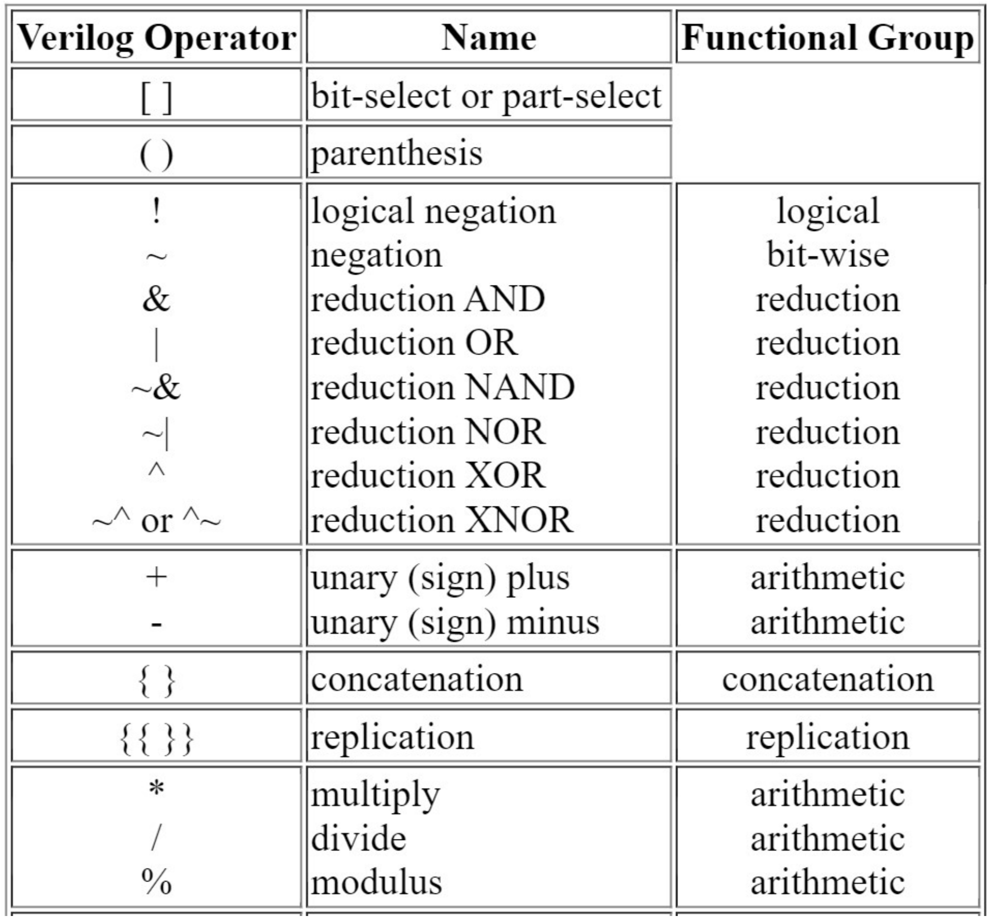
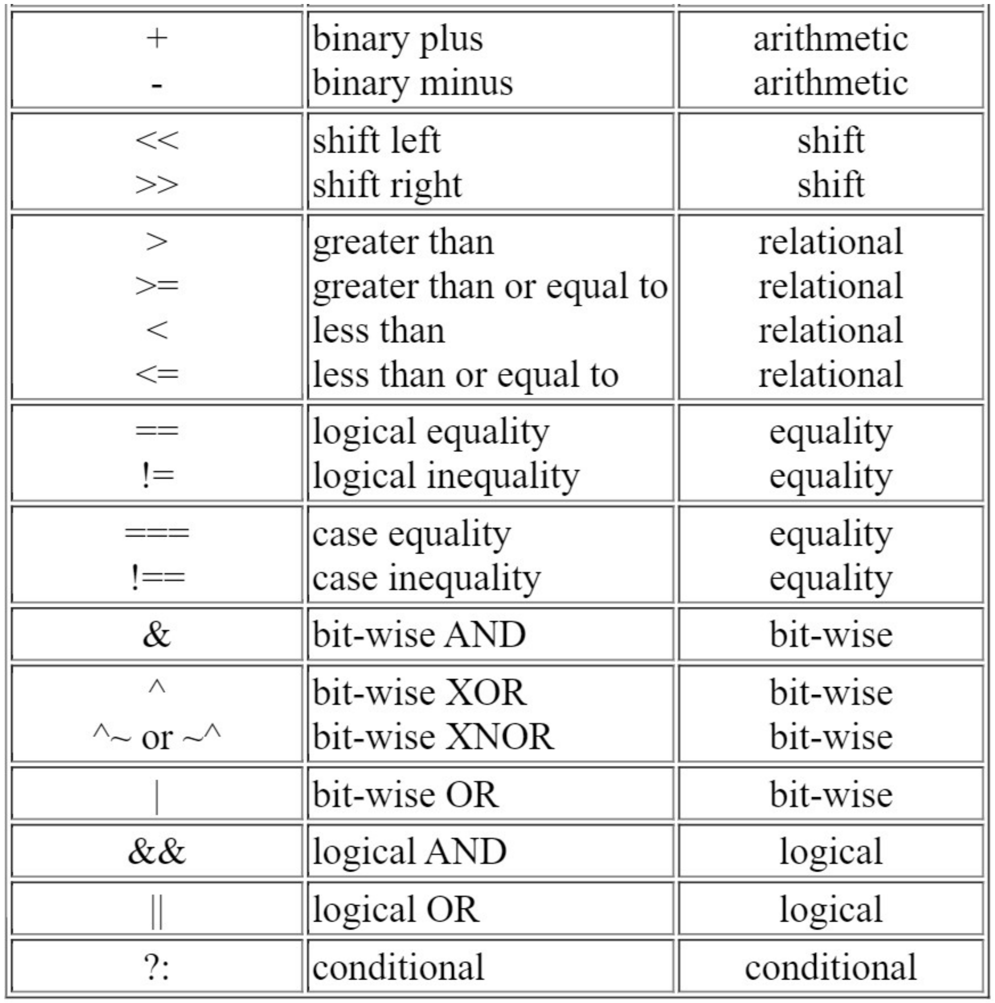
- Example

```
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 00000000000001000000 00000000000000000000
  01234012340123401234 01234012340123401234 01234012340123401234
1 00000000000000000000 00000000000000000000 00000000000000000000
  01234012340123401234 01234012340123401234 01234012340123401234
```

Operators

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

===
!==
	case equality
	case inequality
		equality
		equality

&	bit-wise AND	bit-wise
^ ^~ or ~^	bit-wise XOR bit-wise XNOR	bit-wise bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

- Note: no **a++** and **a+=1**
  - Use **a=a+1** instead
  - More details: https://class.ece.uw.edu/cadta/verilog/operators.html

Process Block (1/3)

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

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

  -
    ```
    always
      <single statement>
    ```

Process Block (2/3)

- 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
- Example
  - **`always @(a or b)`** **//old syntax**
  - **`always @(a, b)`**
  - **`always @(a, posegde clk, negedge enable)`**
  - **`always @a`**
  - **`always @*`**

Process Block (3/3)

- Flow control
- Example

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

  -
    ```
    for() begin
      <statements>
    end
    ```
  - `while`, `case` are 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
- More details
  - 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 only
    - Level = 2: variables in and one level below it
  - `$display(), $write(), $monitor()`

References

- 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/

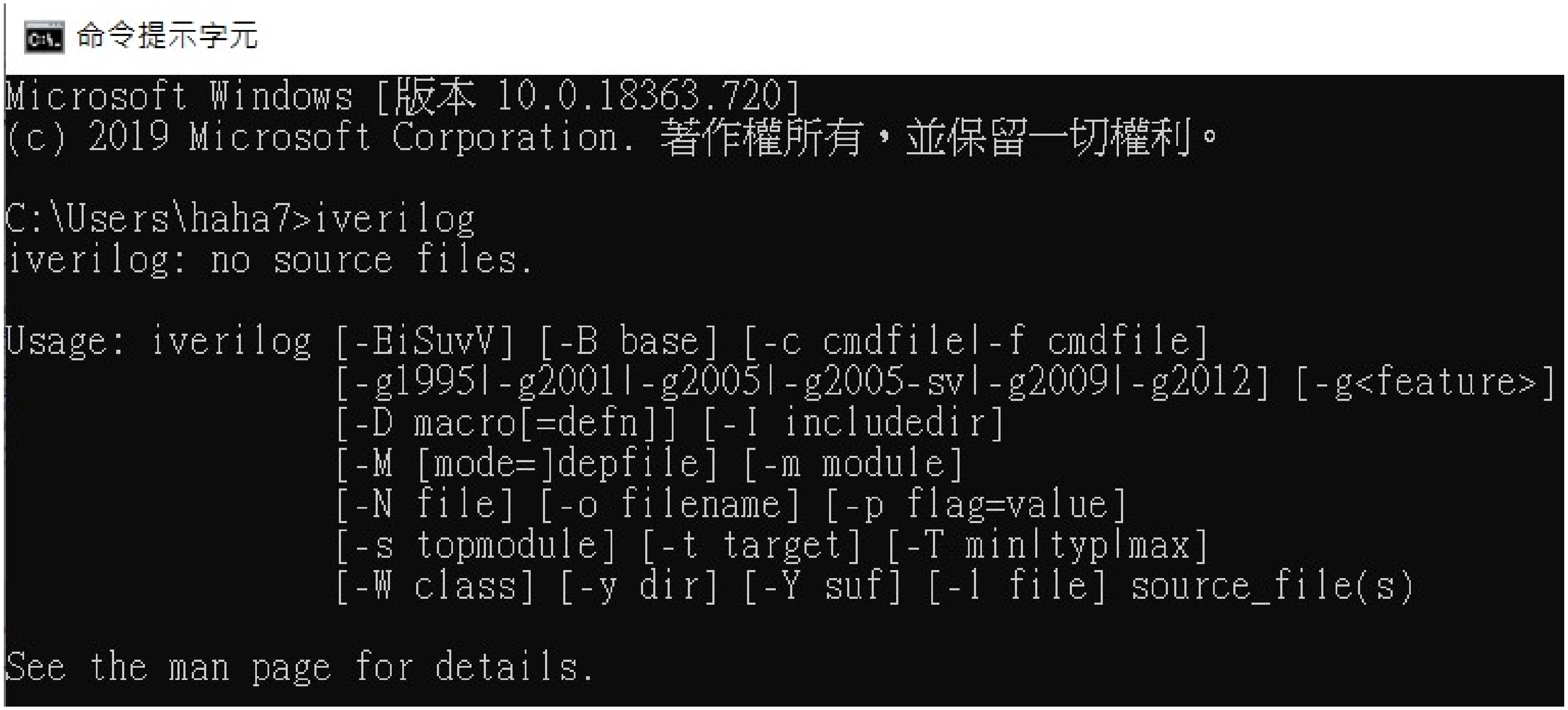
Outline

- ❑ Introduction
- ❑ Verilog Syntax
- ❑ **Simulator Installation**
- ❑ Demonstration
- ❑ Assignment

Icarus Verilog and GTKWave

- What is Icarus Verilog?
  - http://iverilog.icarus.com/
- For Windows users
  - https://bleyer.org/icarus/
  - http://bleyer.org/icarus/iverilog-0.9.7_setup.exe (recommended)
    - Reminder: Run the installer as administrator!
- For Linux-like OS users
  - `sudo apt-get update`
  - `sudo apt-get install iverilog`
  - `sudo apt-get install gtkwave`
- For Mac users
  - Use **`brew install`** instead

After Downloading Successfully

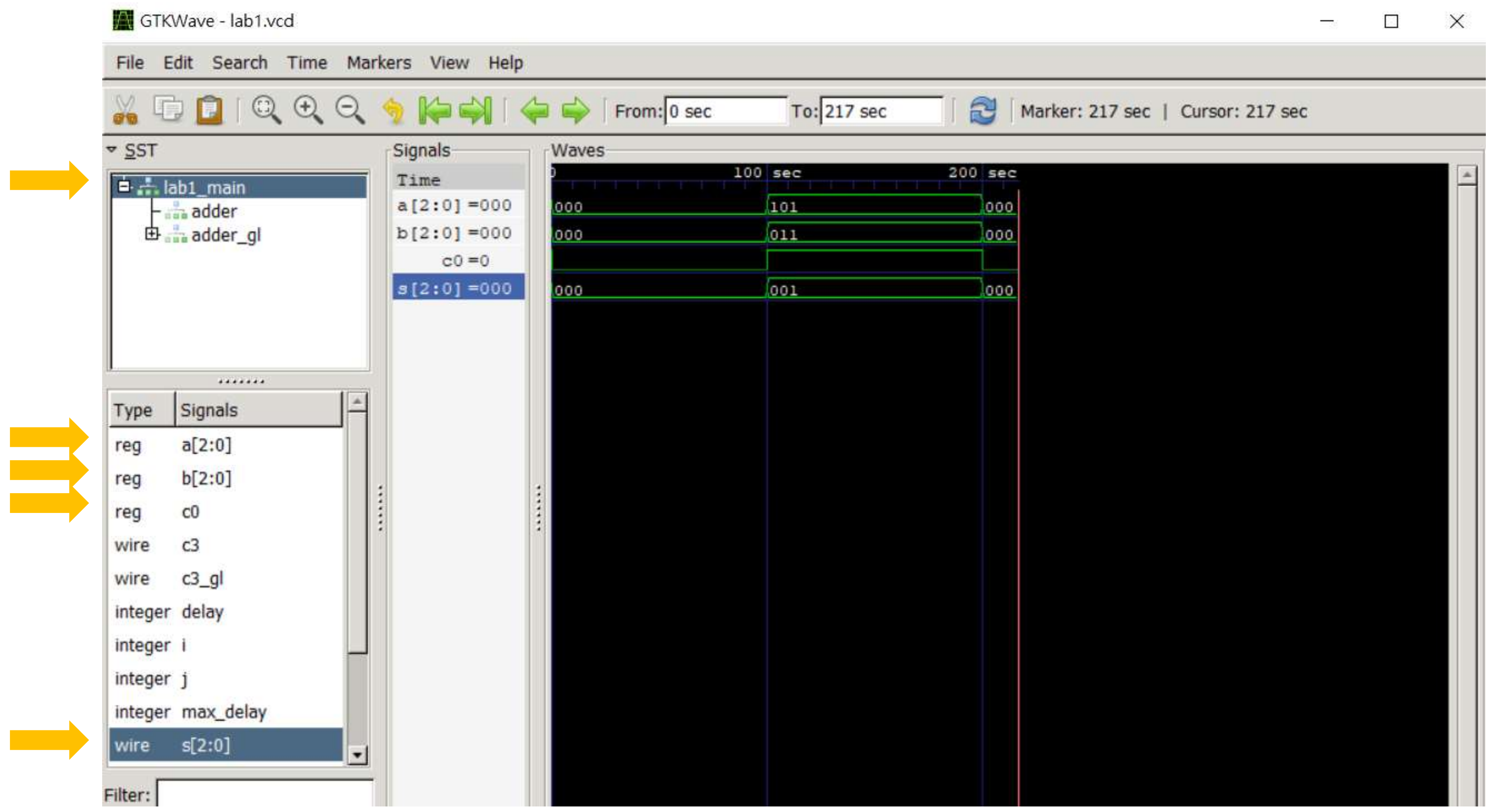


```
Microsoft Windows [版本 10.0.18363.720]
(c) 2019 Microsoft Corporation. 著作權所有，並保留一切權利。

C:\Users\haha7>iverilog
iverilog: no source files.

Usage: iverilog [-EiSuvV] [-B base] [-c cmdfile|-f cmdfile]
                [-g1995|-g2001|-g2005|-g2005-sv|-g2009|-g2012] [-g<feature>]
                [-D macro[=defn]] [-I includedir]
                [-M [mode=]depfile] [-m module]
                [-N file] [-o filename] [-p flag=value]
                [-s topmodule] [-t target] [-T min|typ|max]
                [-W class] [-y dir] [-Y suf] [-l file] source_file(s)

See the man page for details.
```

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
  - Open a new cmd window
  - `> gtkwave`
  - Open **dump_file.vcd** in GUI
  - You can also use **`gtkwave dump_file.vcd`** in the same line

After Opening **dump_file.vcd**



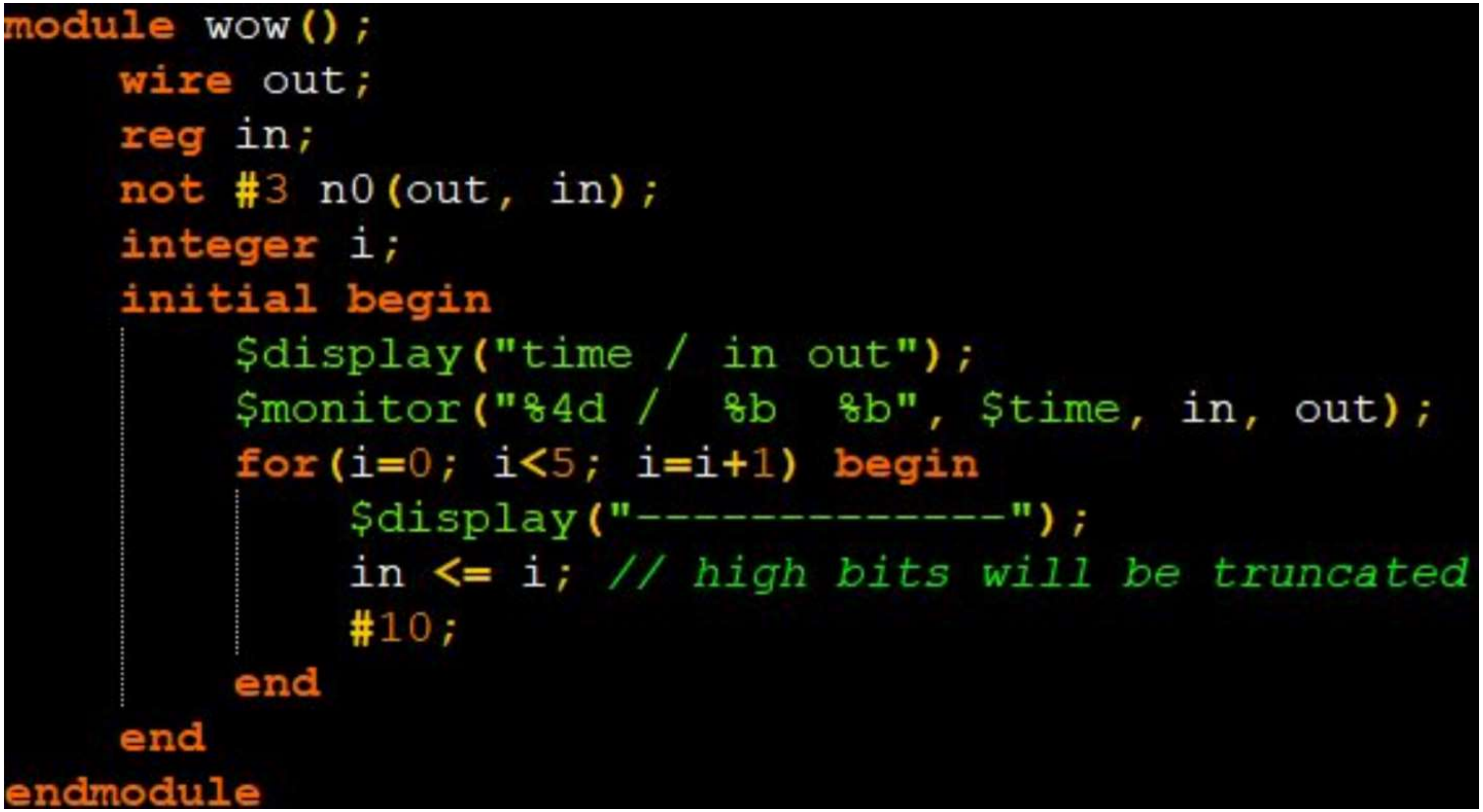
Type	Signals
reg	a[2:0]
reg	b[2:0]
reg	c0
wire	c3
wire	c3_gl
integer	delay
integer	i
integer	j
integer	max_delay
wire	s[2:0]

```
Time
a[2:0] =000
b[2:0] =000
c0 =0
s[2:0] =000
```

```
0            100 sec        200 sec
000          101            000
000          011            000
000          001            000
```

Hello World

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



```
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
```

Trouble Shooting

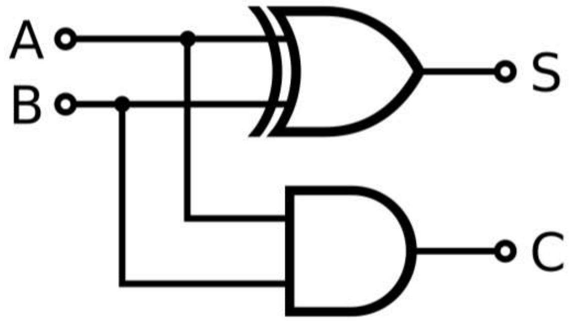
- For Windows users
  - If you cannot run iverilog on command line, uninstall and run the installer as administrator
  - If an error message pop up saying that some dll is missing, or the compiler executed but didn't generate any output file, try the latest version:
    - http://bleyer.org/icarus/iverilog-v11-20190327-x64_setup.exe

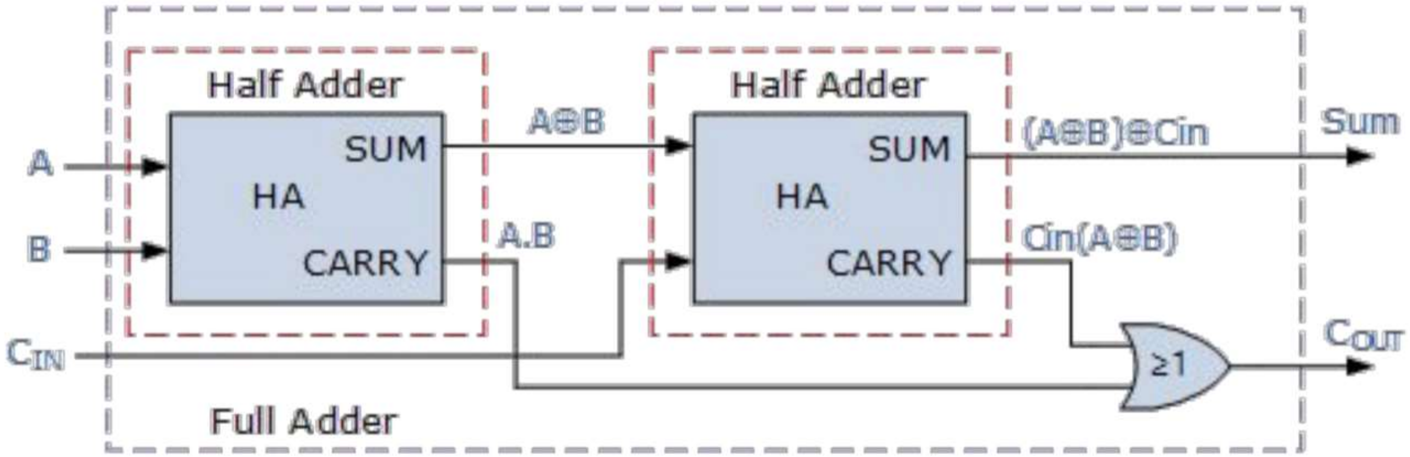
Outline

- ❑ Introduction
- ❑ Verilog Syntax
- ❑ Simulator Installation
- ❑ **Demonstration**
- ❑ Assignment

Half Adder and Full Adder

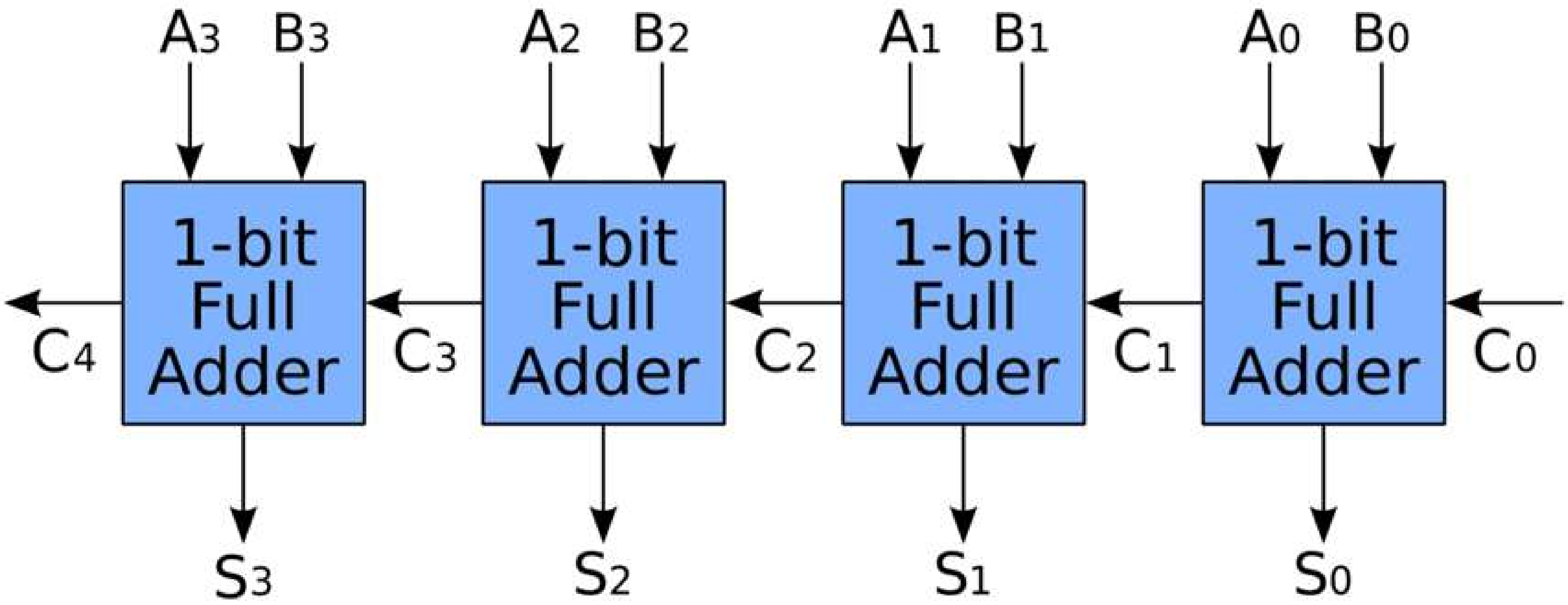
- HA
  - `assign {C, S} = A + B;`
- FA
  - `assign {Cout, S} = A + B + Cin;`
- More details
  - https://www.electronics-tutorials.ws/combination/comb_7.html





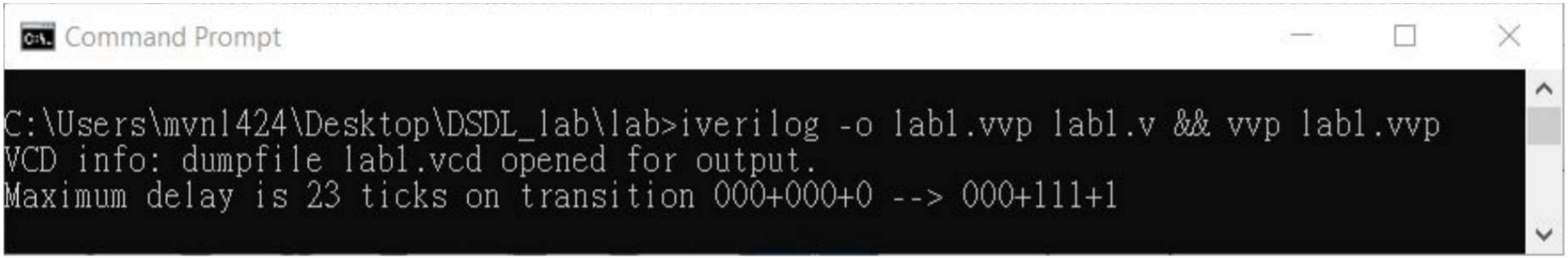
Ripple-Carry Adder (1/4)

- Notice that carry$_n$ depends on carry$_{n-1}$
  - Where is the longest propagation path?



Ripple-Carry Adder (2/4)

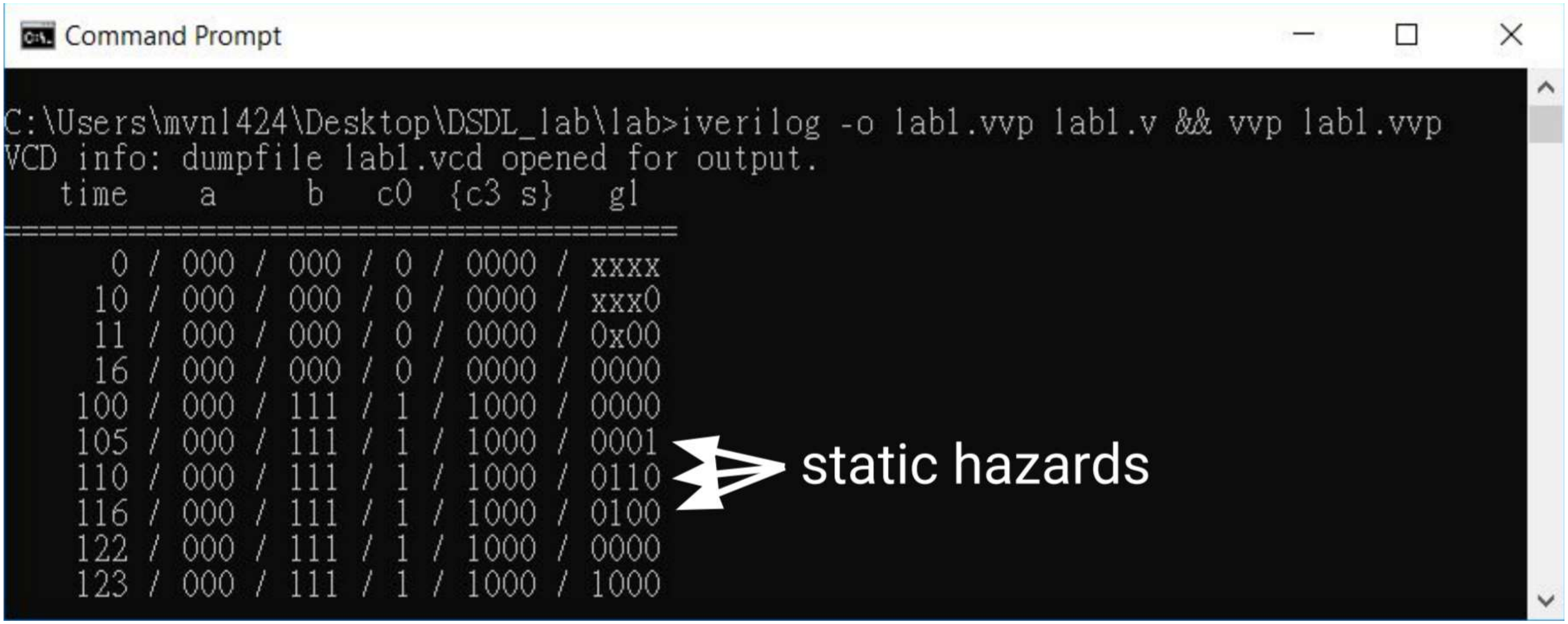
- ❑ Maximum delay
  - ➢ Notice that 000+000+0 → 111+000+1 should cause the same delays



```
C:\Users\mvnl424\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
```

Ripple-Carry Adder (3/4)

❑ Transition



Command Prompt

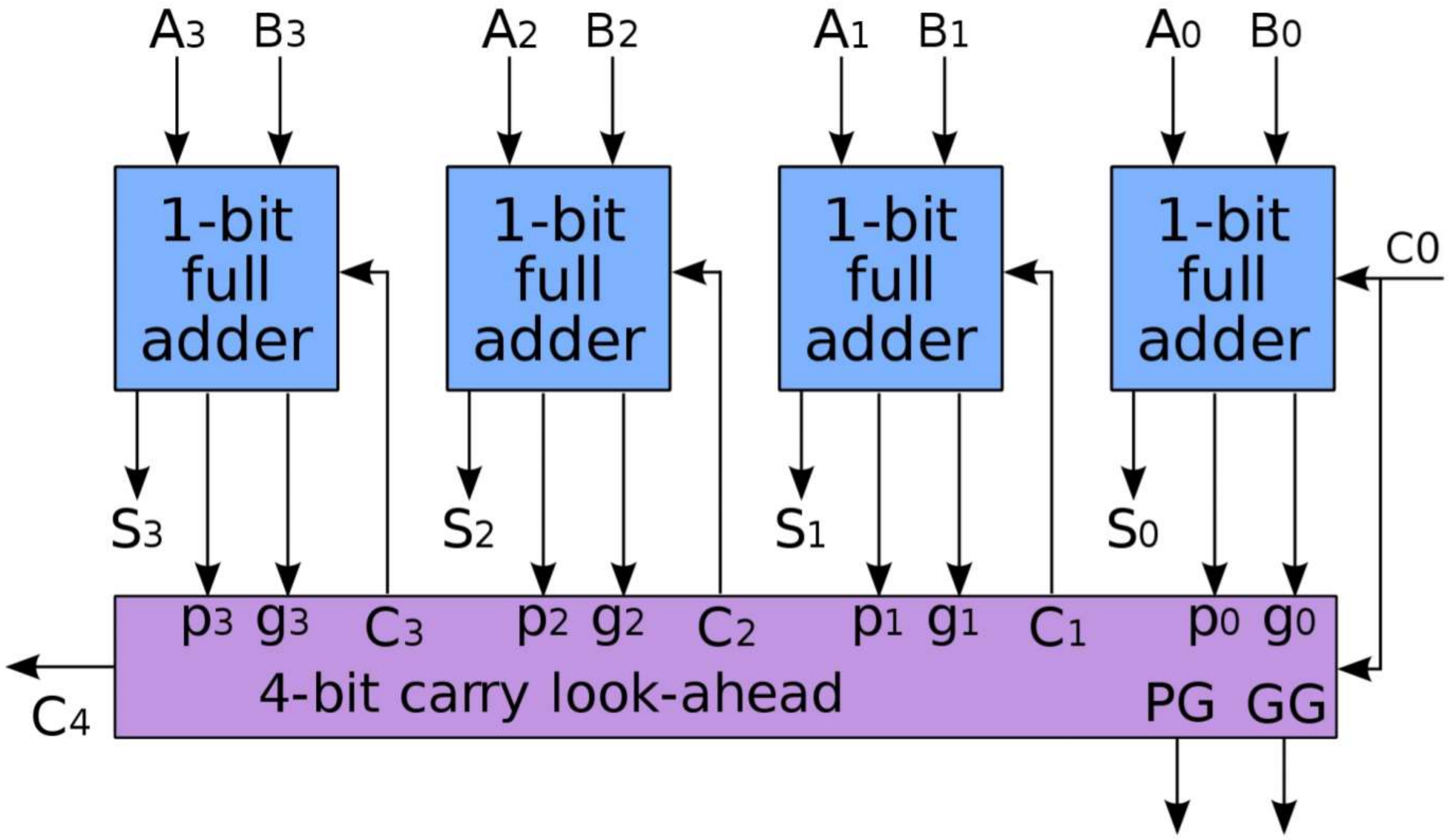
```
C:\Users\mvn1424\Desktop\DSDL_lab\lab>iverilog -o lab1.vvp lab1.v && vvp lab1.vvp
VCD info: dumpfile lab1.vcd opened for output.
   time     a      b    c0  {c3 s}   gl
=====================================
      0 / 000 / 000 / 0 / 0000 / xxxx
     10 / 000 / 000 / 0 / 0000 / xxx0
     11 / 000 / 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
```

Ripple-Carry Adder (4/4)

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

Outline

Carry-Lookahead Adder (1/2)



Carry-Lookahead Adder (2/2)

- Carry lookahead method
  - $G_i = A_i \cdot B_i$
  - P_i can be either:

    $P_i = A_i \oplus B_i$

    $P_i = A_i + B_i$
- Implementation details
  - $C_1 = G_0 + P_0 \cdot C_0$
  - $C_2 = G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1$
  - $C_3 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2$
  - $C_4 = G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3$
- Reference
  - https://en.wikipedia.org/wiki/Carry-lookahead_adder

Requirements (1/2)

- Implement a 3-bit adder **`adder_rtl`**
  - Use RTL modeling
- Implement a 3-bit carry-lookahead adder **`cla_gl`**
  - Use gate-level modeling
  - Only the gates provided in **gates.v** (with delays) are allowed to use

Requirements (2/2)

- Attach your source codes to your Homework 3
- Show the waveform of **`cla_gl.S[2:0]`** on input transitions
  - From 000 + 000 + 0 to 001 + 000 + 1
  - From 111 + 000 + 0 to 000 + 111 + 0
- Find the maximum propagation delay of **`cla_gl`**
  - And find 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
- Hand in along with Homework 3

Hints

- **adder_rtl** should be simple to implement
  - It can be implemented in one line
- The outputs of **adder_rtl** and **cla_gl** should be the same at steady state
  - If they are different, maybe there are some mistakes in **cla_gl** since it is more 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

Q&A