

1

Computer Organization

Lab8 Verilog &

EDA tools (Vivado+Minisys Board, Icarus Verilog + GTKwave)

2021 Spring term

wangw6@sustech.edu.cn

Topics

- **Verilog**
 - A kind of **H**ardware **D**escription **L**anguage
- **EDA tools**
 - vivado
 - Icarus Verilog + GTKwave
- **Practice**

verilog

- ▶ **Design-Under-Test** vs **Test-Bench**
- ▶ Structured Design(top **module**, instance **module**)
- ▶ Block (Combinational, Sequential)
- ▶ Statement
 - ▶ Continuous assignment
 - ▶ Unblock assignment vs Block assignment
 - ▶ If else ,case, loop
- ▶ Variable vs Constant
 - ▶ Reg vs Wire, Splicing { , }, Number system

DUT vs Testbench

- **DUT is a designed module with input and output ports**
 - While do the design, **non-synthesizable grammar** means can't be convert to circuit, **is NOT suggested!**
 - DUT may be a top module using structured design which means the sub module is instanced and connected in the top module
- **Testbench is used for test DUT with NO input and output ports**
 - Instance the DUT, bind its ports with variable, set the states of variable which bind with inputs and check the states of variable which bind with outputs
 - **Testbench is NOT a part of Design.** It only runs in FPGA/ASIC EDA, so the un-synthesizable grammar can be used in testbench

Module (Structured Design vs TestBench)

```

module multiplexer_153(out,c0,c1,c2,c3,a,b,g1n);
input c0,c1,c2,c3;
input a,b;
input g1n;
output reg [3:0] out;

always @(*)
if(1'b0==g1n)
    case({b,a})
        2'b00:out=4'b1110;
        2'b01:out=4'b1101;
        2'b10:out=4'b1011;
        2'b11:out=4'b0111;
    endcase
else
    out = 4'b1111;
endmodule

```

```

module multiplexer_153_2(out1,out2,c10,c11,c12,c13,a1,b1,g1n,
    c20,c21,c22,c23,a2,b2,g2n);
input c10,c11,c12,c13,a1,b1,g1n,c20,c21,c22,c23,a2,b2,g2n;
output out1,out2;

    multiplexer_153 m1(
        .g1n(g1n),
        .a(a1),
        .b(b1),
        .c0(c10),
        .c1(c11),
        .c2(c12),
        .c3(c13),
        .out(out1)
    );

    multiplexer_153 m2(
        .g1n(g2n),
        .a(a2),
        .b(b2),
        .c0(c20),
        .c1(c21),
        .c2(c22),
        .c3(c23),
        .out(out2)
    );

endmodule

```

```

module lab3_df_sim( );
    reg simx,simy;
    wire simq1,simq2,simq3;
    lab3_df u_df(
        .x(simx), .y(simy), .q1(simq1), .q2(simq2), .q3(simq3) );

    initial
    begin
        simx=0;
        simy=0;

        #10
        simx=0;
        simy=1;

        #10
        simx=1;
        simy=0;

        #10
        simx=1;
        simy=1;

    end
endmodule

```

Module Design

➤ Gate Level

- Implementation from the **perspective of gate-level structure** of the circuit, **using gates as components, connecting pins of gates**
- Using **logical and bitwise operators or original primitive**(not , or , and , xor , xnor ..)
 - For example: `not n1(na,a); xor xor1(c,a,b);`

➤ Data Streams

- Implementation from the **perspective of data processing and flow**
- Using **continuous assignment**, pay attention to the correlation between signals, the difference between logical and bitwise operators
 - For example: `assign z = (x | y) ^ (a&b);`

➤ Behavior Level

- Implementation from the **perspective of the Behavior** of Circuits
- Implemented in the **always** statement block
- The variable which is assigned in the always block Must be **Reg type**.

Behavior Modeling(if – else)

“if else” block can represent the **priority** among signals

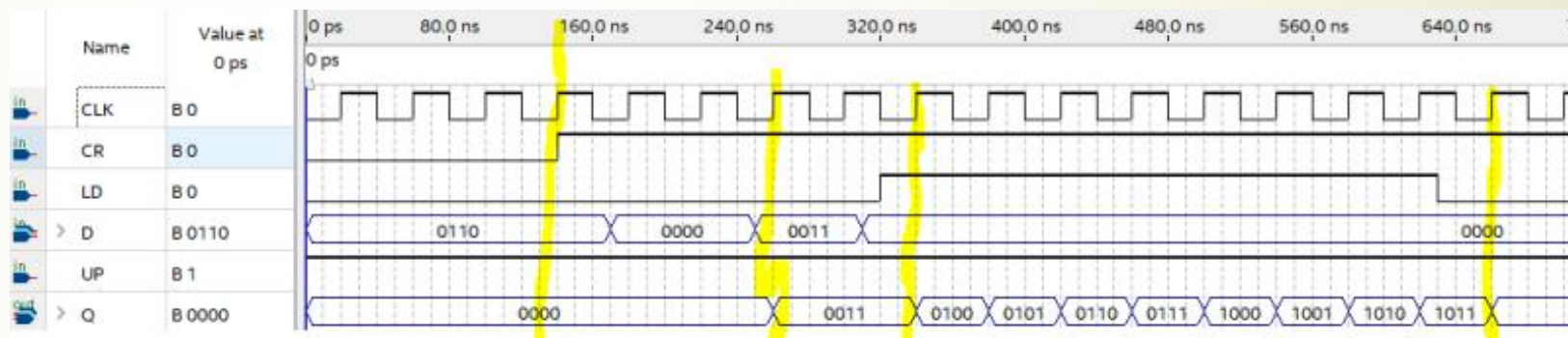
From the overall structure, from top to bottom, priority decreases in turn

```

module updown_counter(D,CLK,CR,LD,UP,Q)
input [3:0]D;
input CLK,CR,LD,UP;
output reg [3:0] Q;

always @(posedge CLK )
if(!CR)
    Q=0;
else if(!LD)
    Q=D;
else if(UP)
    Q=Q+1;
else
    Q=Q-1;
endmodule

```



NOTIC:

- 1) If there is no 'else' branch in the statement, latches will be generated while doing the synthesis.
- 2) Nested 'if-else' is NOT suggested, 'case' is suggested as an alternative.

Behavior Modeling(case)

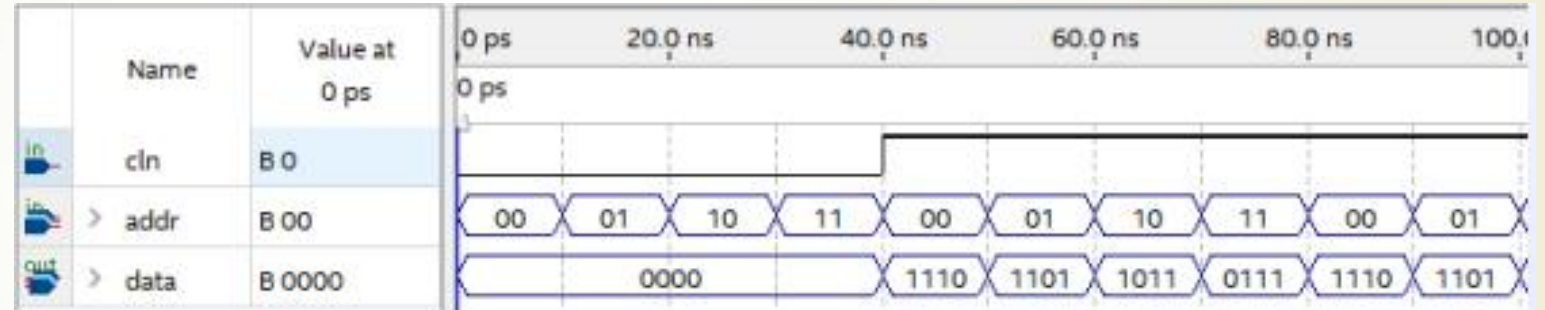
```

module decoder(cIn,data,addr);
input cIn;
input [1:0] addr;
output reg [3:0] data;

always @(cIn or addr )
begin
if(0==cIn)
data=4'b0000;
else
case(addr)
2'b00:data=4'b1110;
2'b01:data=4'b1101;
2'b10:data=4'b1011;
2'b11:data=4'b0111;
endcase
end
endmodule

```

case	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

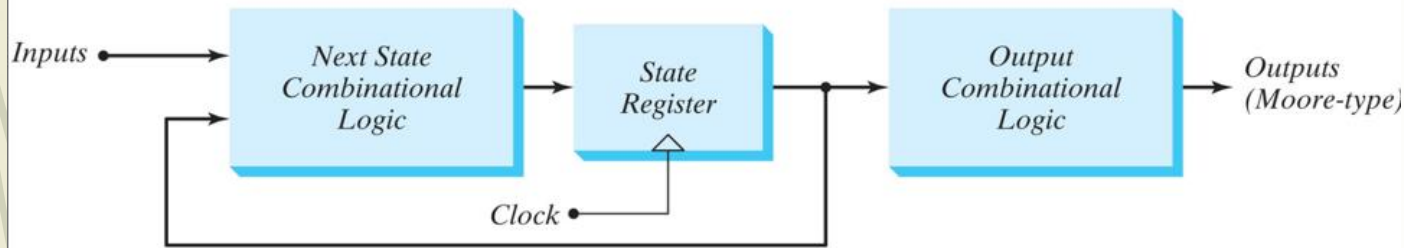


NOTIC:

Without defining 'default' branches and declaring all situations under the "case", latches will be generated while doing the synthesis.

Sequential Circuit: FSM

Moore Machine



```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module moore_2b(input clk,rst_n,x_in,output[1:0] state,next_state);
  reg [1:0] state,next_state;
  parameter S0=2'b00,S1=2'b01,S2=2'b10,S3=2'b11;
  always @(posedge clk,negedge rst_n) begin
    if(~rst_n
      state <= S0;
    else
      state <= next_state;
    end
  end
  always @(state,x_in) begin
    case(state)
      S0: if(x_in) next_state = S1; else next_state = S0;
      S1: if(x_in) next_state = S2; else next_state = S1;
      S2: if(x_in) next_state = S3; else next_state = S2;
      S3: if(x_in) next_state = S0; else next_state = S3;
    endcase
  end
end
endmodule
  
```

Constant

➤ Expression

➤ `<bit width>'<numerical system expression><number in the numerical system >`

➤ Numerical system expression

➤ B / b : Binary

➤ O / o : Octal

➤ D / d : decimal

➤ H / h : hexadecimal

➤ `'<numerical system expression><number in the numerical system >`

➤ The default value of bit width is based on the machine-system(at least 32 bit)

➤ `<number>` : default in decimal

➤ The default value of bit width is based on the machine-system(at least 32 bit)

Constant continued

- ▶ **X**(uncertain state) and **Z**(High resistivity state)
 - ▶ The default value of a wire variable is **Z** before its assignment
 - ▶ The default value of a reg variable is **X** before its assignment
- ▶ Negative value
 - ▶ Minus sign must be ahead of bit-width
 - ▶ -4'd3 (is ok) while 4'd-3 is illegal
- ▶ Underline
 - ▶ Can be used between number but can NOT be in the bit width and numerical system expression
 - ▶ 8'b0011_1010 (is ok) while 8'_b_0011_1010(is illegal)
- ▶ Parameter (symbolic constants)
 - ▶ Used for improving the readability and maintainability
 - ▶ Declare an identifier on a constant
 - ▶ Parameter p1=expression1,p2=expression2,...;

Variable

➤ Wire

- Net
- Can 't store info ,must be driven (such as continuous assignment)
- The input and output port of module is wire by default
- Can NOT be the type of left-hand side of assignment in initial or always block

```
wire a;  
wire [7:0] b;  
wire [4:1] c,d;
```

➤ Reg

- MUST be the type of **left-hand** side of assignment in initial or always block
- The default initial value of reg is an indefinite value X. Reg data can be assigned positive values and negative values.
- When a reg data is an operand in an expression, its value is treated as an unsigned value, that is, a positive value.
- For example, when a 4-bit register is used as an operand in an expression, if the register is assigned -1. When performing operations in an expression. It is considered to be a complement representation of + 15 (- 1)

Variable continued

```
module sub_wr();
  input reg in1,in2;
  output out1;
  output out2;
endmodule
```

Error: Port in1 is not defined
 Error: Non-net port in1 cannot be of mode input
 Error: Port in2 is not defined
 Error: Non-net port in2 cannot be of mode input

```
module sub_wr(in1,in2,out1,out2);
  input in1,in2;
  output out1;
  output reg out2;

  assign in1 = 1'b1;

  initial begin
    in2 = 1'b1;
  end

endmodule
```

Error: procedural assignment to a non-register in2 is not permitted, left-hand side should be reg/integer/time/genvar

```
23 module test_wire_reg(
24   );
25   wire i1,i2;
26   reg o1,o2;
27   sub_wr s1(i1,i2,o1,o2);
28 endmodule
29
30 module sub_wr(in1,in2,out1,out2);
31   input in1,in2;
32   output out1;
33   output reg out2;
34
```

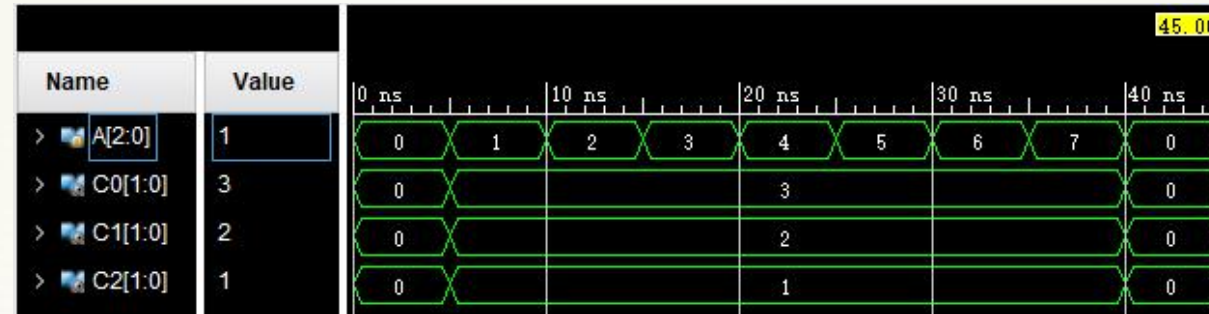
[Synth 8-685] variable 'o1' should not be used in output port connection [test_wire_reg.v:27]

Memory

```

module test(
    A, C0, C1, C2
);
    input [2:0] A;
    output [1:0] C0, C1, C2;
    reg [1:0] B [2:0];
    assign {C0, C1, C2} = {B[0], B[1], B[2]};
    always @(A)
    if(A)
    begin
        B[0] = 2'b11;
        B[1] = 2'b10;
        B[2] = 2'b01;
    end
    else
    begin
        B[0] = 2'b00;
        B[1] = 2'b00;
        B[2] = 2'b00;
    end
end
endmodule

```



Does the waveform belongs to the two test?
If not, which one does it belong to?

```

module test(
    A, C0, C1, C2
);
    input [2:0] A;
    output [1:0] C0, C1, C2;
    reg [1:0] B [2:0];
    assign {C0, C1, C2} = {B[0], B[1], B[2]};
    always @(A)
    if(A)
    begin
        {B[0], B[1], B[2]} = 6'b011011;
        /*B[0] = 2'b11;
        B[1] = 2'b10;
        B[2] = 2'b01;*/
    end
    else
    begin
        {B[0], B[1], B[2]} = 6'b0;
        /*B[0] = 2'b00;
        B[1] = 2'b00;
        B[2] = 2'b00;*/
    end
end
endmodule

```


Operator

highest	! ~
priority	* / %
	+ -
	<< >>
	< <= > >=
	== != === !==
	&
	^ ^~
	&&
lowest	
priority	? :

Bit splicing operator { }

Multiple data or bits of data are separated by commas in order, then using braces to splice them as a whole.

Such as:

{a, B [1:0], w, 2'b10} // Equivalent to {a, B [1], B [0], w, 1'b1, 1'b0}

Repetition can be used to simplify expressions

{4 {w}} // Equivalent to {w, w, w, w}

{b, {2 {x, y}}} // Equivalent to {b, x, y, x, y}

Operator continued

When numeric values are used for conditional judgment, non-zero values represent logical truth and zero values represent logical false.

```
module test_bool(A,C);
input [2:0]A;
output reg [2:0]C;

always @(A )
begin
if(A)
C=2'b11;
else
C=2'b00;
end
endmodule
```

	Name	Value at 0 ps	0 ps	20.0 ns	40.0 ns	60.0 ns	80.0 ns				
			0 ps								
ip	> A	B 000	000	001	010	011	100	101	110	111	000
out	> C	B 000	000			011					000

```
module test_bool(A,C);
input [2:0]A;
output reg [2:0]C;

always @(A )
begin
if(A==1)
C=2'b11;
else
C=2'b00;
end
endmodule
```

	Name	Value at 0 ps	0 ps	20.0 ns	40.0 ns	60.0 ns	80.0 ns	100.0 ns				
			0 ps									
ip	> A	B 000	000	001	010	011	100	101	110	111	000	001
out	> C	B 000	000	011			000					011

17

EDA 1. VIVADO

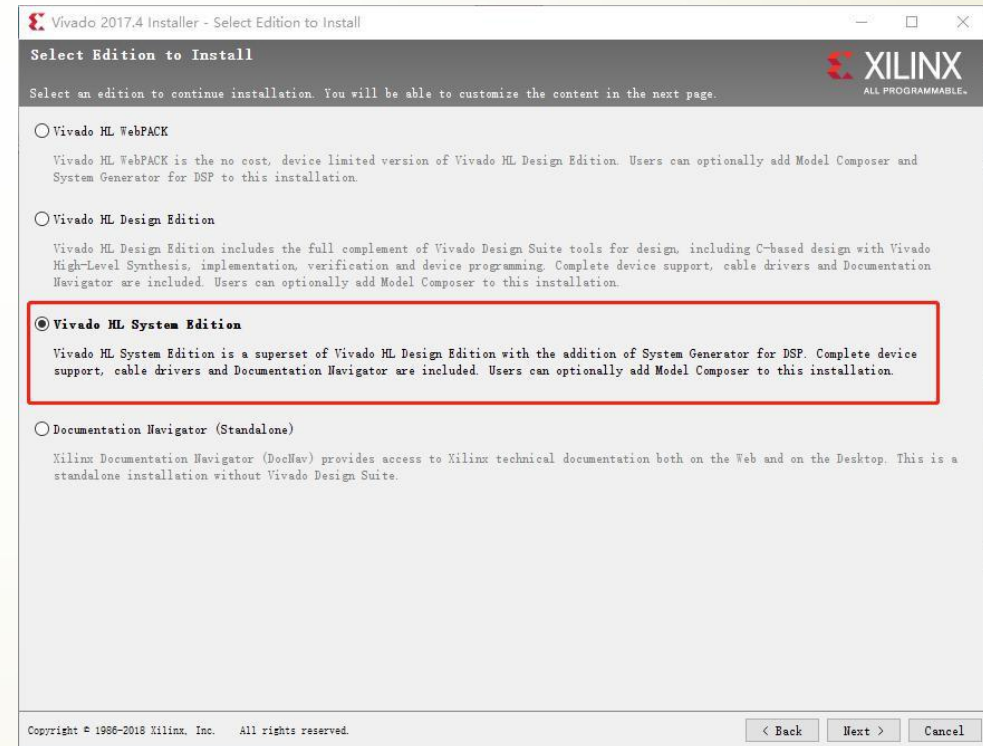
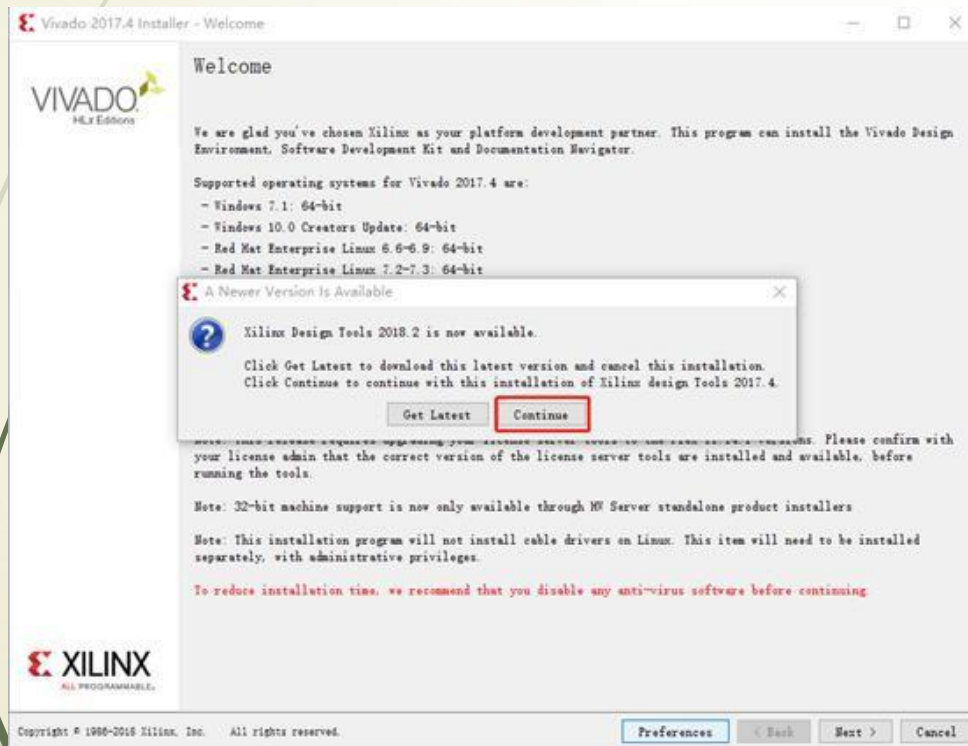
option 1(in campus): <ftp://10.20.118.226/>

account: ftp-d-logic

password: ggsddu

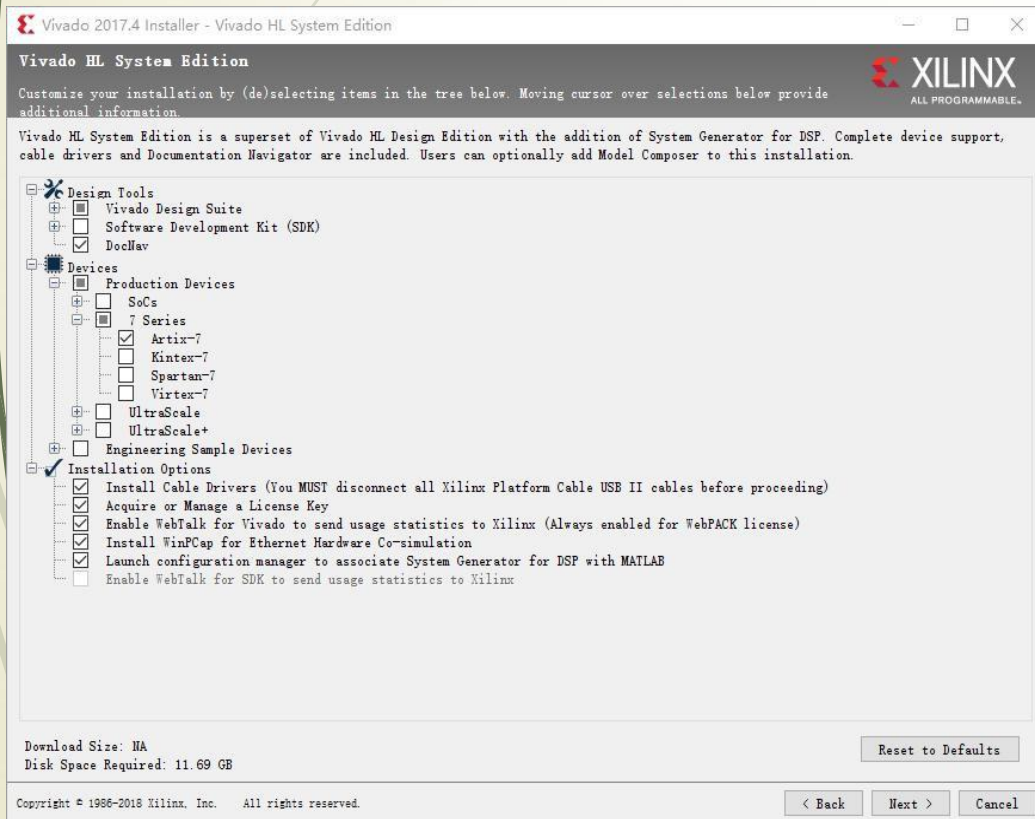
option 2(not in campus): <https://pan.baidu.com/s/1MfeMCK2igcsf1WEQA49ObA>

key: fhr4

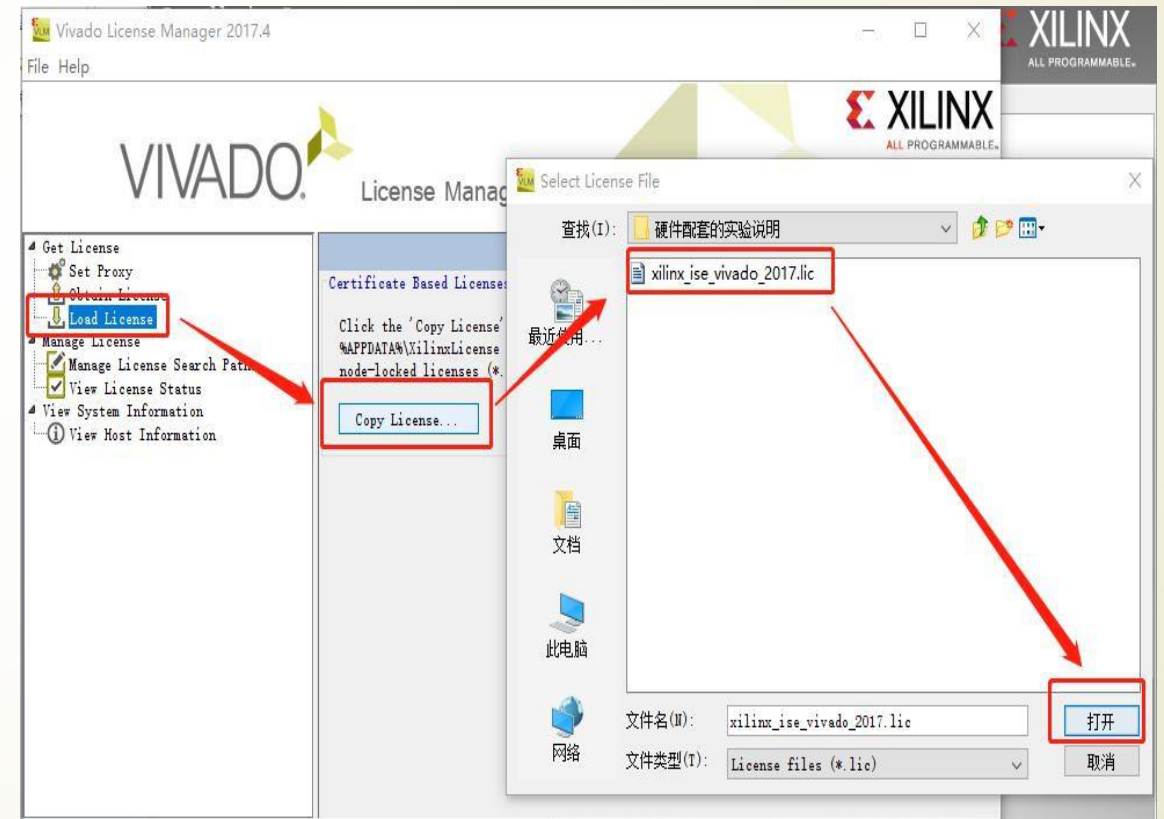


VIVADO Installation

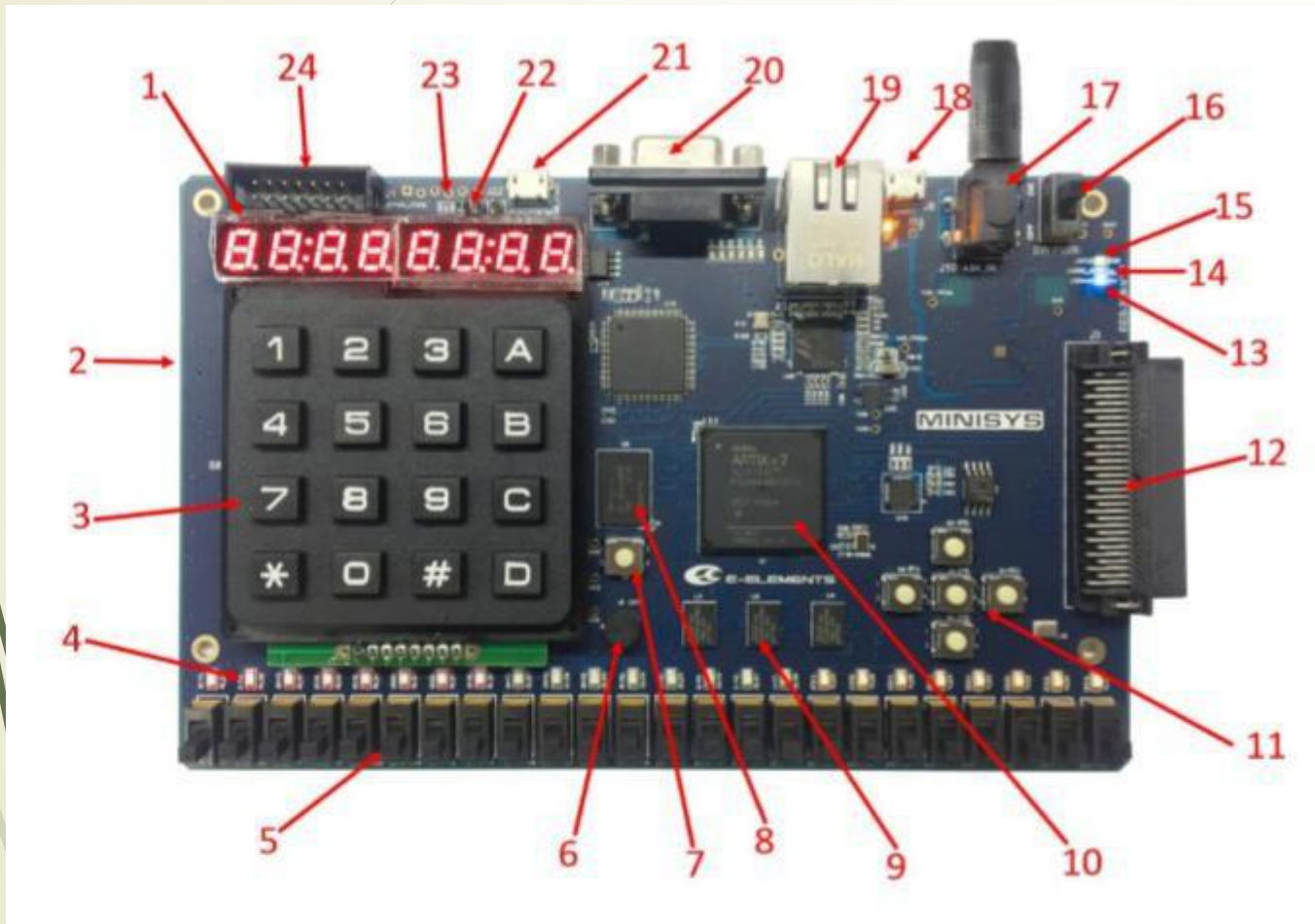
Select only what is needed



At the end of installing, load license



MINISYS Board For Developing



- Artix 7 **fpga** chip (10)
- Minisys soc system inside
- Power interface (17)
- USB-jtag interface (21)
- Dial switch *24 (5)
- LED *24 (4)
- SRAM (9)
- Mini keyboard (3)
- seven-segment digital tube (1)

Vivado Project

1. Do the design with verilog (Vivado)

2. Do the simulation to verify the function of the design(Vivado)

3. Do the synthesis , Do the implementation, Generate bit stream file(Vivado)

4. Connect with FPGA chip, Programe the chip with bitstream file (Vivado + FPGA chip)

5. Do the test on the programmed FPGA chip (FPGA chip)

A vivado project

1. Manage all the files

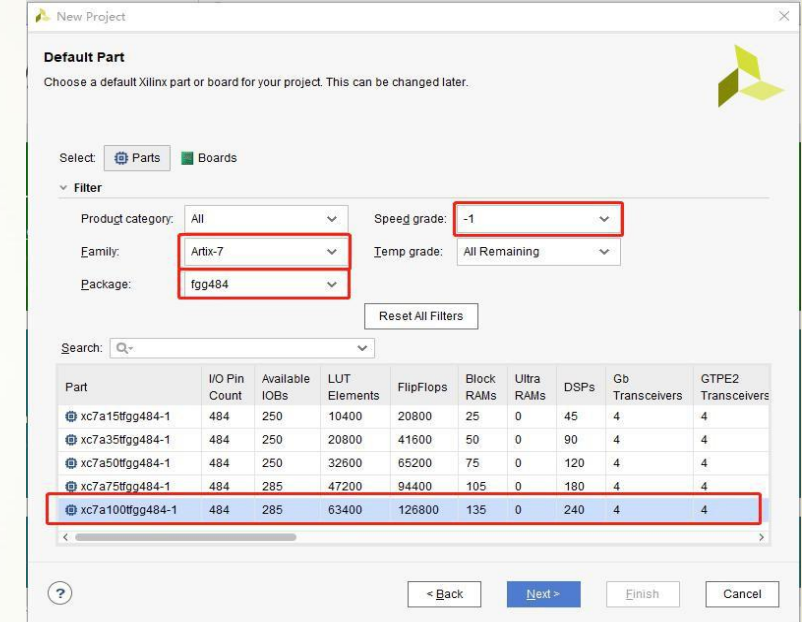
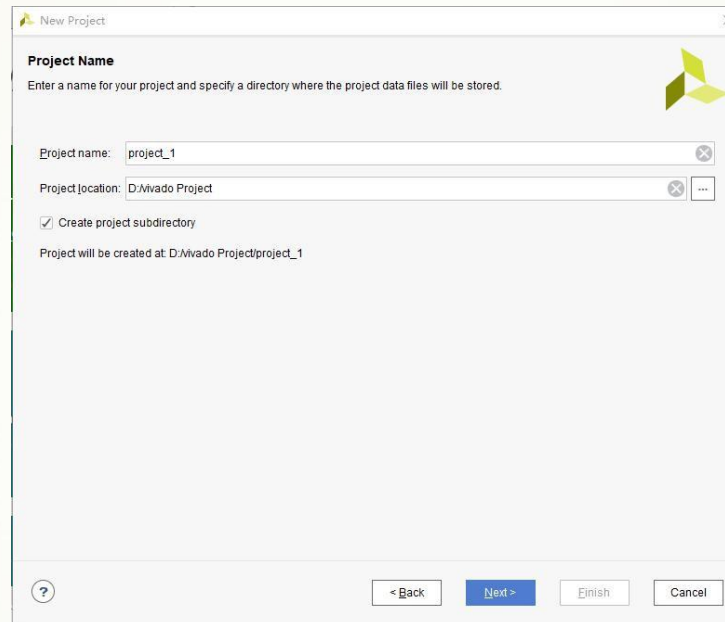
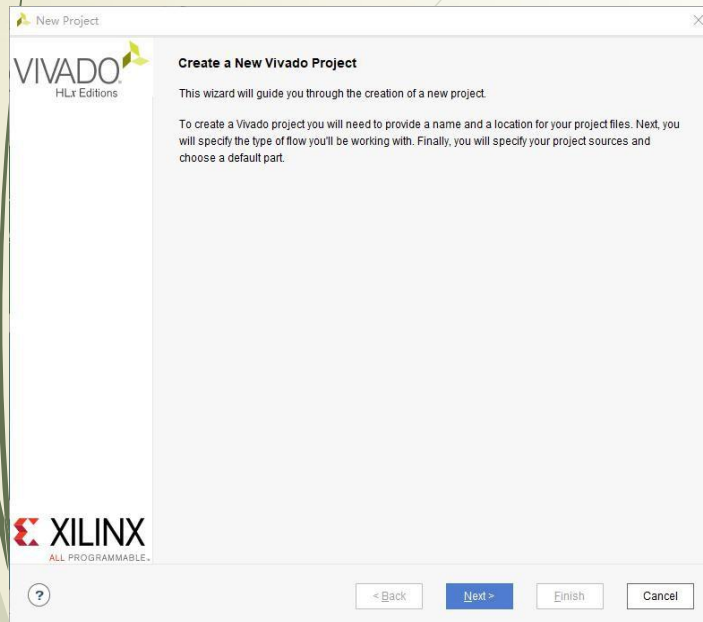
(including design file, simulation file, constraint file and other resource file)

2. Manage the operation flow

3. Connect with FPGA chip

4. Program the FPGA chip

Using VIVADO



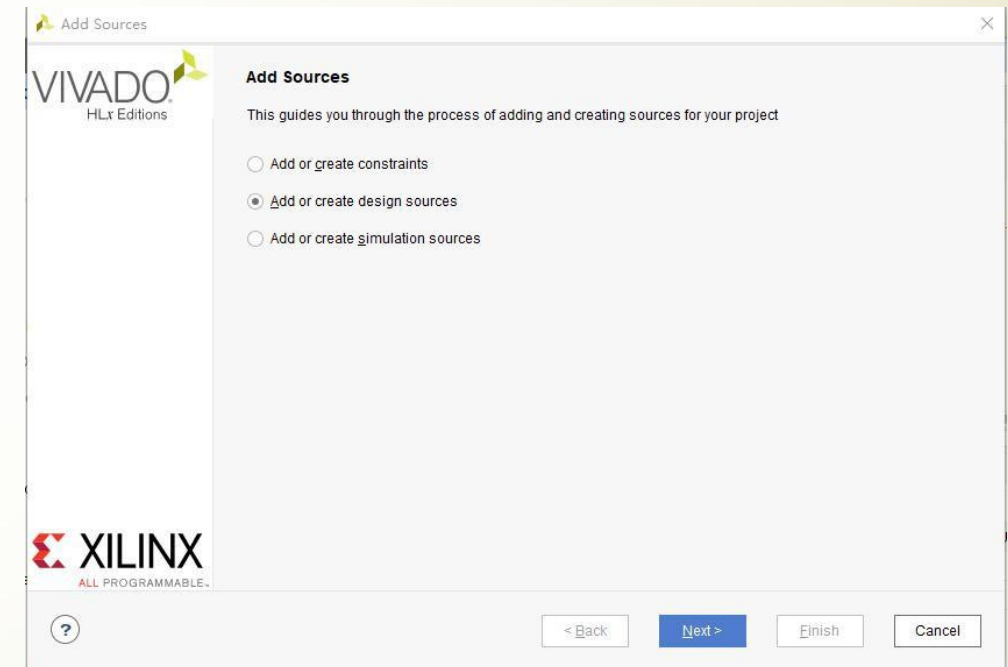
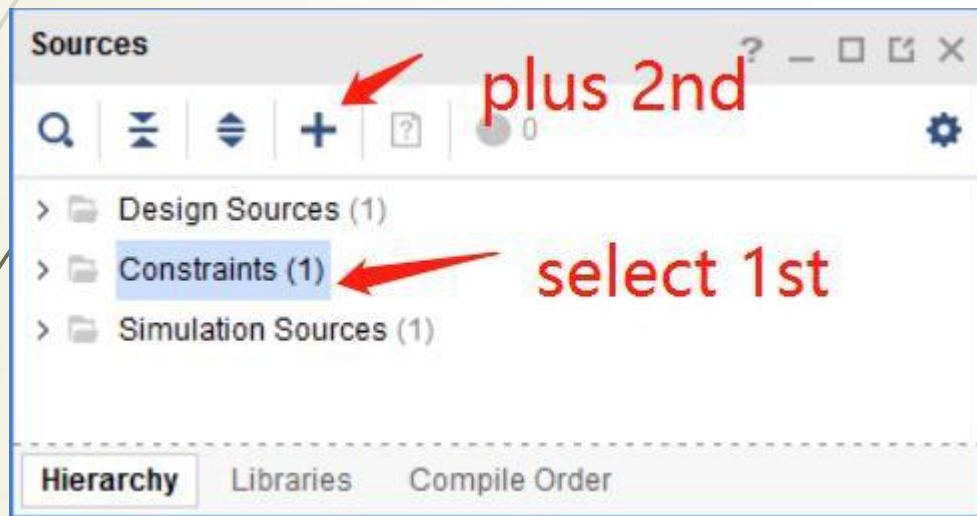
1. Create project

1) Select "rtl type" as project type

2) Select xc7atfgg484-100 as the corresponding chip name

Using VIVADO continued

2. Add source file, simulation file and constraints file to vivado project



Using VIVADO continued



3. Do the following steps to generate bitstream file.

1) Do the **simulation** to verify the Circuit function[**step1**]

2) After simulation, a waveform file is generated which records the states of input and output signals.

3) If the function of circuit is ok, **Run synthesis**[**step2**] , then **Run implements**[**step3**]

4) After implementation is finished, **Generate Bitstream**[**step4**], the generated “.bit” file could be used to program device later.

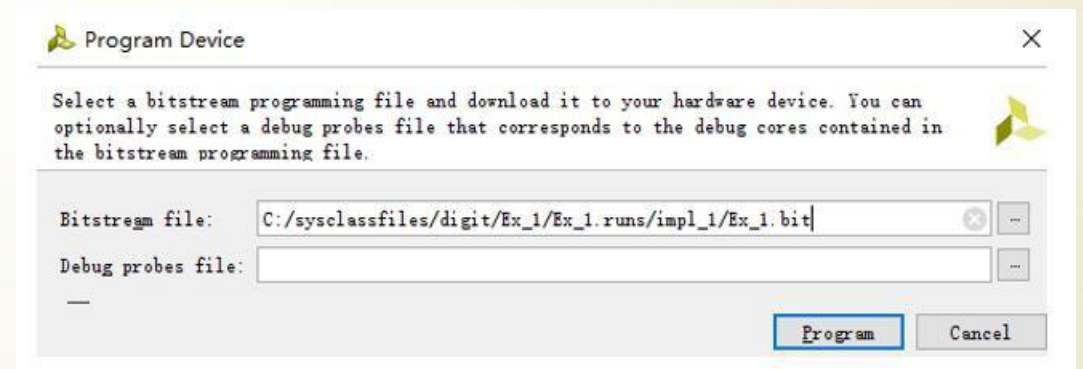
Using VIVADO + MINISYS

4. **Connect** MINISYS board with PC (USB JTAG interface) which run the vivado project
5. **Turn on** the power of minisys board
6. Click on “open target ” to **connect** the vivado project with MINISYS board



Using VIVADO + MINISYS continued

7. Right click “**program device**”, then choose the device name.
8. Select the **bitstream file**, click “**program**” button
9. While the led of “Done” on minisys is on, it means the bit file has been written into the device.
10. Do the **testing** on the MINISYS board.

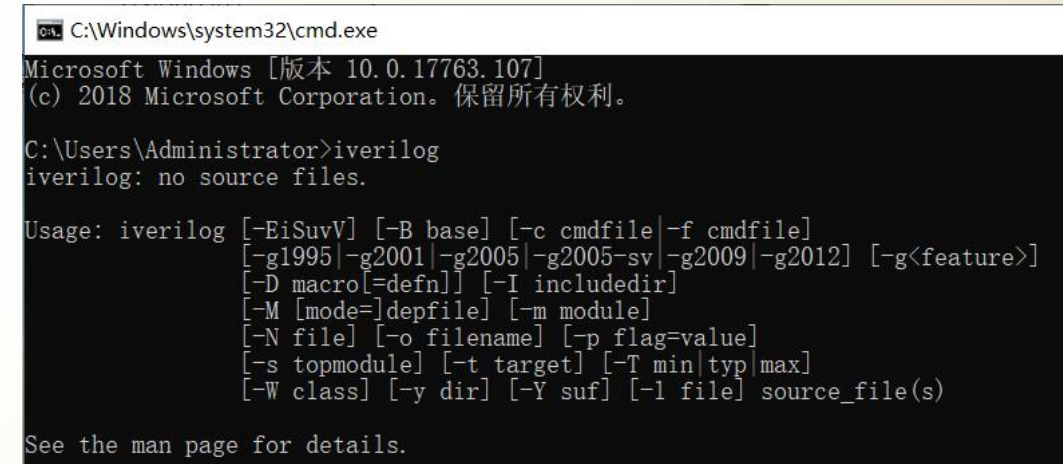


EDA 2. Icarus Verilog and GTKwave

If it's really hard for you to install vivado now, try a free simulator on verilog for temporary use.

Following steps tells how to install and invoke it on Windows:

- Download from
 - <http://bleyer.org/icarus/>
- **Double click** to install
- Add following value to **PATH** (one of the system environment variable)
 - `path-to-install-folder\bin`
 - `path-to-install-folder\gtkwave\bin`(“`path-to-install-folder`” is the actual folder of your iverilog installation)
- While iverilog is installed and configured, invoke a cmd window, input **iverilog**
 - if it works, the following message will show on the cmd window as the picture on right hand.



```
C:\Windows\system32\cmd.exe
Microsoft Windows [版本 10.0.17763.107]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>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.
```

EDA 2. Icarus Verilog and GTKwave continued

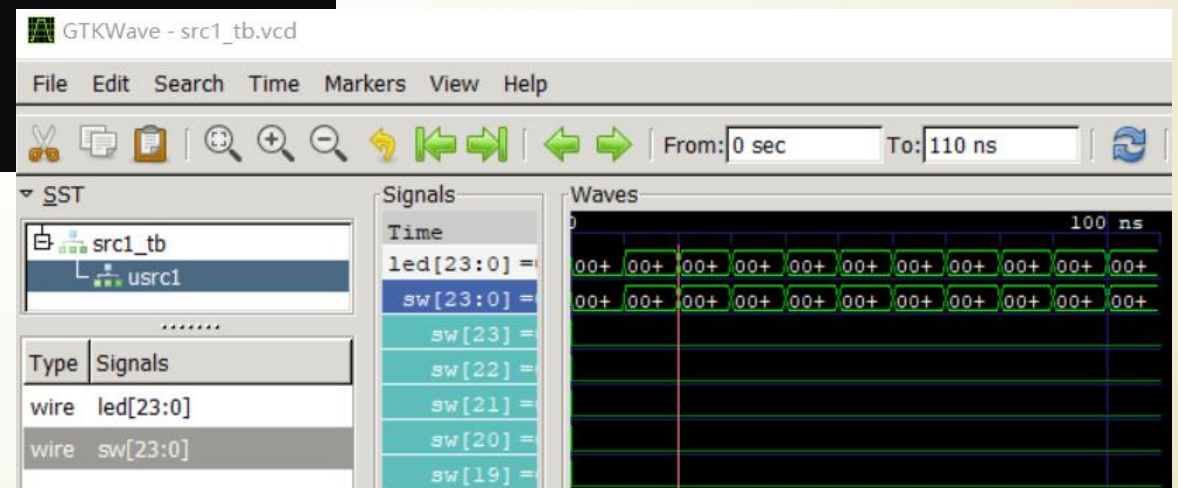
While the design file and testbench file are all ready, follow the following steps to do the simulation and check the result.

- Using cmd "**iverilog -o a.out a.v a_tb.v**" to compile the design (a.v) and testbench (a_tb.v) to generate a result file a.out.
- Using cmd "**vvp a.out**" to generate a waveform file whose name(such as a_tb.vcd) has been set in the testben file
- Using cmd "**gtkwave a_tb.vcd**" to inspect the waveform in a_tb.vcd .

```
d:\iverilog\user_space\lab1_src>iverilog -o src1_sim.out src1.v src1_sim.v
d:\iverilog\user_space\lab1_src>vvp src1_sim.out
VCD info: dumpfile src1_sim.vcd opened for output.
d:\iverilog\user_space\lab1_src>gtkwave src1_sim.vcd
```

NOTIC: to generate the wave form, following instructions need to be add in testbench file :

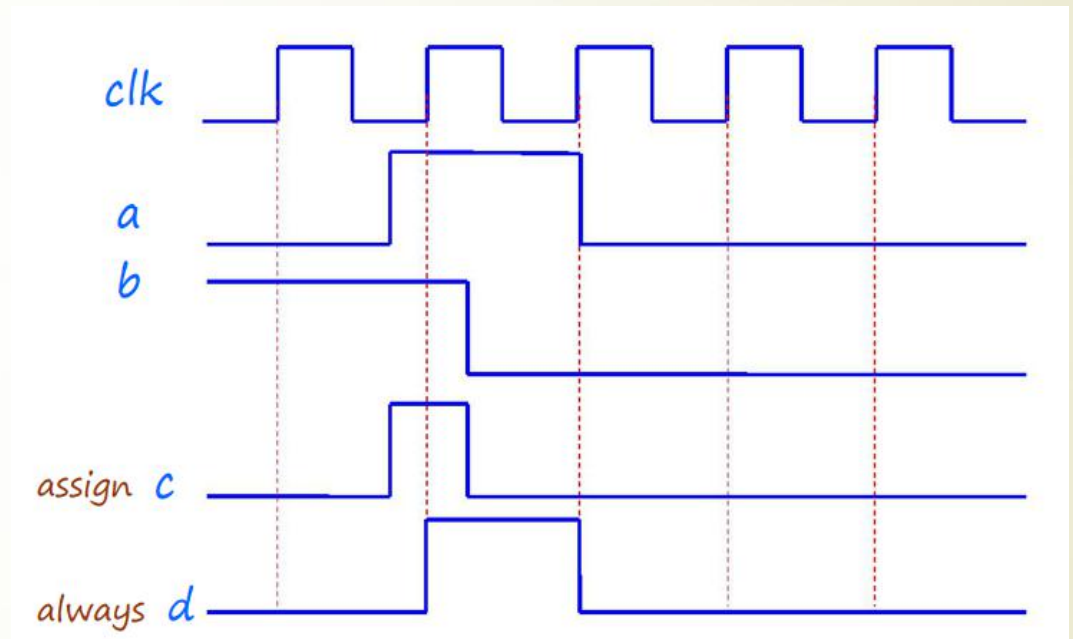
```
$dumpfile("src1_tb.vcd");
$dumpvars(0,src1_tb);
```



Practice

Question 1

- ▶ **Build a circuit** with 3 inputs a, b and clk, 2 outputs c and d.
 - ▶ a, b, clk, c and d are all 1-bit width.
 - ▶ clk is a clock signal with a duty cycle of 50%.
 - ▶ While both a and b are 1'b1, c is 1'b1, otherwise c is 1'b0;
 - ▶ On every posedge of clk, the value of a and b are checked, if both a and b are 1'b1, d is 1'b1, otherwise d is 1'b0. on other time, d keeps its state.
- ▶ **Build the testbench** as the snap on the right, **testing the function** of the design by simulation on the vivado/iverilog.



Practice continued

➤ Question 2

- Build a circuit as a signed adder, there are 2 inputs with 4bits width('a' and 'b'), and 2 outputs('sum' is 4bits width, 'of' is 1bit width).
 - 'of' is the overflow flag on signed data calculation. While the XOR result on the Maximum significant bit and the bit which is on the right hand of Maximum significant bit of addition result is 1, it means there is overflow on signed calculation, otherwise reverse.
 - for example: $4'b1011 + 4'b1011 = 4'b0110$, 'of' = 1
- Build the design and testbench in verilog, and do the simulation to verify its function with vivado/iverilog.