



# Computer Organization

---

## Lab8 Verilog and EDA tools



Verilog and EDA tools



# Topic

- **Verilog**
  - A kind of **Hardware Description Language**
  - **module, block, statement, operator, data**
  - **suggestion**
- **EDA tools**
  - **vivado**
  - Icarus Verilog + GTKwave
- **Practice(optional)**



# 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
- **Operator and data**
  - Variable vs Constant
  - Reg vs Wire, Splicing { , }, Number system



# DUT vs Testbench

*design under test*

- **DUT is a designed module for circuit 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(s) is(are) instanced and connected in the top module
- **Testbench is build for test circuit 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
```

和用户交互, IO不为空

组合逻辑, 和时钟无关

```
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
```

结构化设计  
(用已定义好的模块搭建系统)

Here are 3 pieces of verilog code, Which one is(are) the circuit design, which one is(are) the testbench?

What are the common point(s) and difference(s) between the circuit design and the testbench?

```
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
```

无输入输出



# Module (Circuit Design)

## ➤ Gate Level

- Implementation from the **perspective of gate-level structure** of the circuit, **using gates as components, connecting pins of gates** 用门搭建
- 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); 信号线

## ➤ 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)

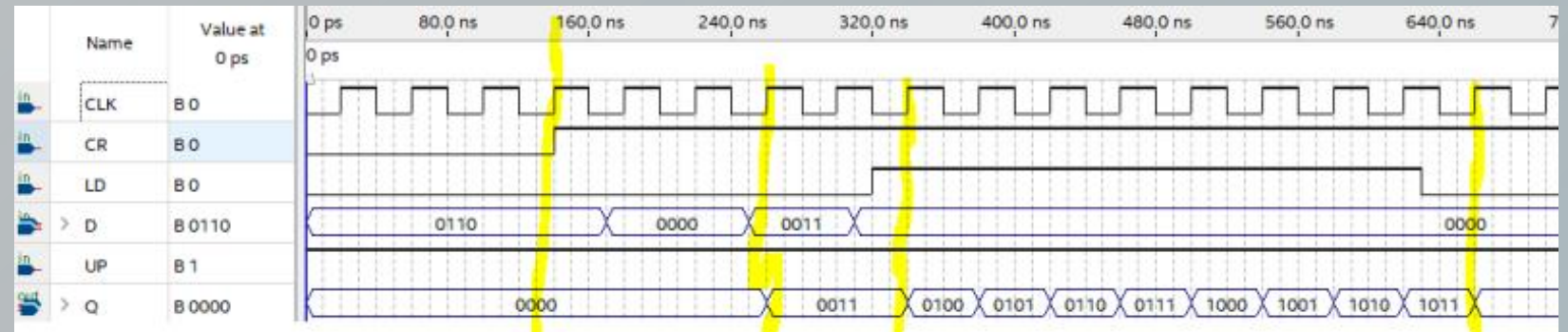
always/initial 一起使用

"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(more clear while reading, less latency).

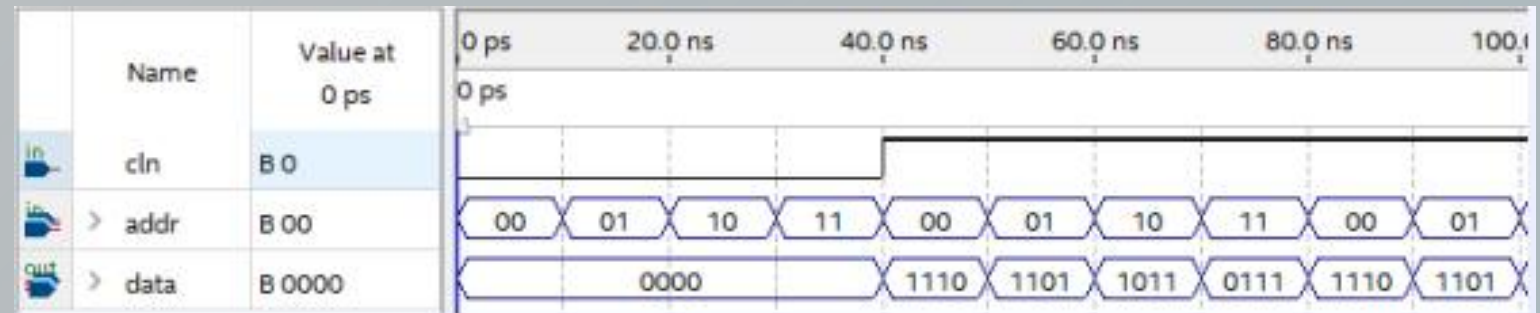


# Behavior Modeling(case)

```
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

The piece of verilog code(on the right hand) is a **2-stage** code(using **2 always block** describe the **combinational** logic and **sequential** logic separately).

Does it describe the **Moore-type** FSM or **Mealy-type** FSM?

**NOTES:** While implement a **Mealy-type** FSM, 1-stage(using 1 always block describe the combinational logic and sequential logic) is **NOT** suggested!!

符号化常量

**parameter** S0=2'b00, S1=2'b01, S2=2'b10, S3=2'b11;

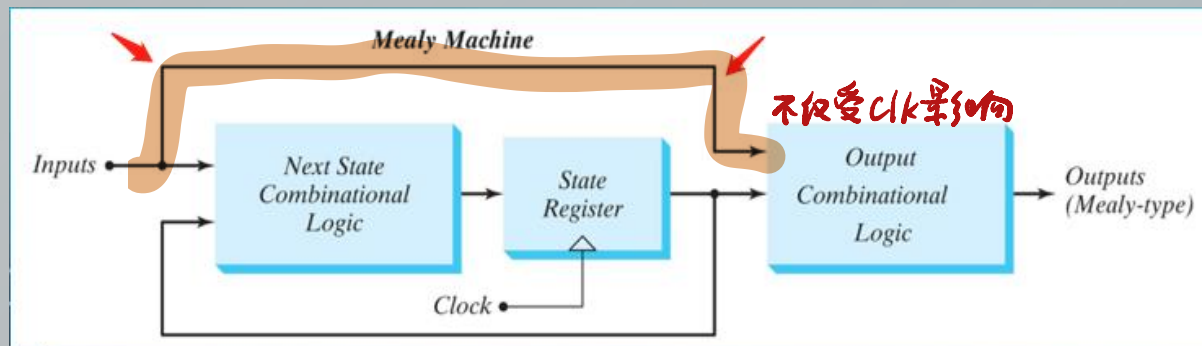
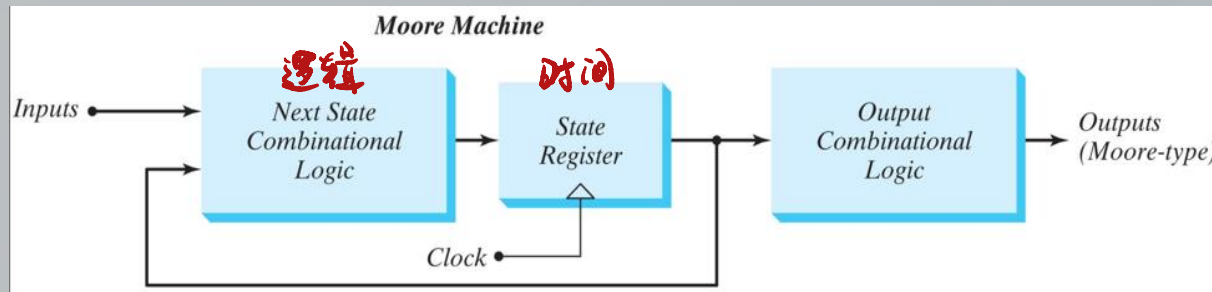
时序

最多2个

非阻塞, 时序中使用

组合

阻塞, 组合逻辑中使用



```
`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
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
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( data type)

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

## ➤ 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

## ➤ Reg *initial / always 中被赋值的*

- **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 ( data type) continued

Please find the syntax error about the data type in the following pieces of code.

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

端口名放在()内  
输入必须wire

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

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

wire  
输出不可给reg

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

assign in1 = 1'b1; 连续赋值

```
initial begin  
  in2 = 1'b1;  
end
```

过程赋值  
左侧需为reg

endmodule

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

对input赋值 X multiple driver

sim\_1 (2 errors)

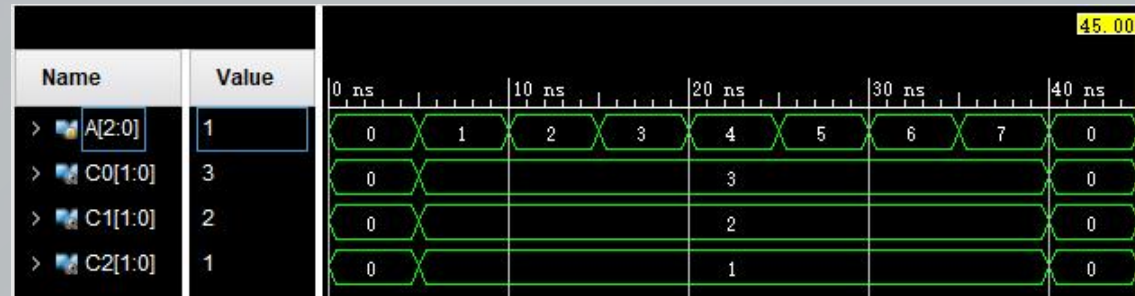
[VRFC 10-529] concurrent assignment to a non-net o1 is not permitted [test.v:29]





# 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
```



Q1: Does the waveform belongs to the two test?  
If not, which one does it belong to?

Q2: While do the following assignment: “{B[0],B[1],B[2]} = 6'b011011;”, what’s the value of B[0] ? Is it same as the comments list on the right picture?

```
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	<div>↑ ! ~ ↘ 位取反</div>
	* / %
	+ -
	<< >>
	< <= > >=
	== != === !==
	&
	^ ^~
	&&
lowest priority	<div>   ? : ↗</div>

## Bit splicing operator { }

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

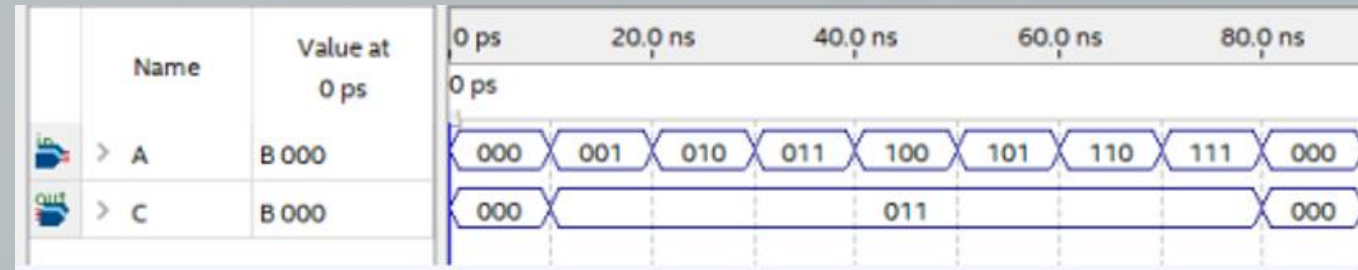


# Operator continued

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

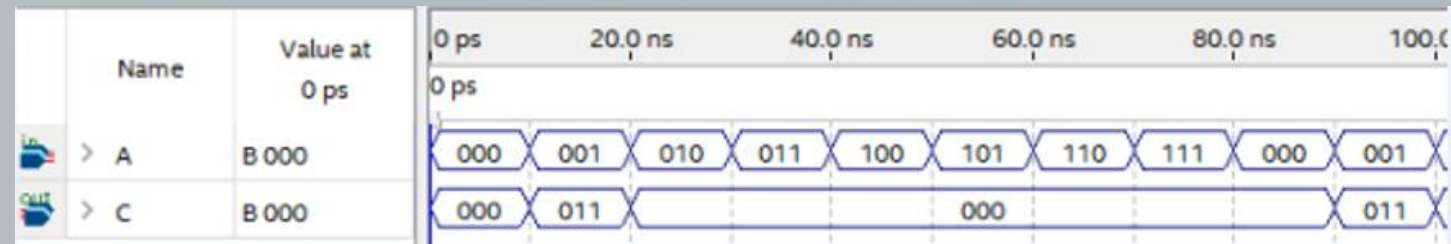
always @(A )
begin
if (A)
C=2'B11;
else
C=2'B00;
end
endmodule
```

Here are two circuits described in verilog and the corresponding waveforms. What' s the difference between two pieces of code?



```
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
```



Tips: When data are used for conditional judgment, **non-zero represent logical truth** and **zero represent logical false**.



# verilog suggestion

- **Non-Synthesizable Verilog** which is **NOT suggested** to use in your design
  - initial; Task, function; System task : \$display, \$monitor, \$strobe, \$finish
  - fork... join; UDP
- **Suggested**
  - Using an asynchronous reset to make your system go to initial state
  - Using case instead of embedded 'if-else' to avoid unwanted priority and longer delay
- **NOT suggested**
  - Embedded 'if-else'
  - Two different edge trigger for one always block
  - **(!!!) a signal/port is assigned in more than one always block** (it won't report error while synthesized but its behavior maybe wrong after synthesize)
  - Mix-use blocking assignment and non-blocking assignment in one always block



# EDA TOOLS: VIVADO

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, Program the chip with bitstream file (Vivado + FPGA chip)

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

**At the very beginning,  
a vivado project is needed!**

## **Vivado project**

1. Manage all the files

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

2. Manage the operation flow

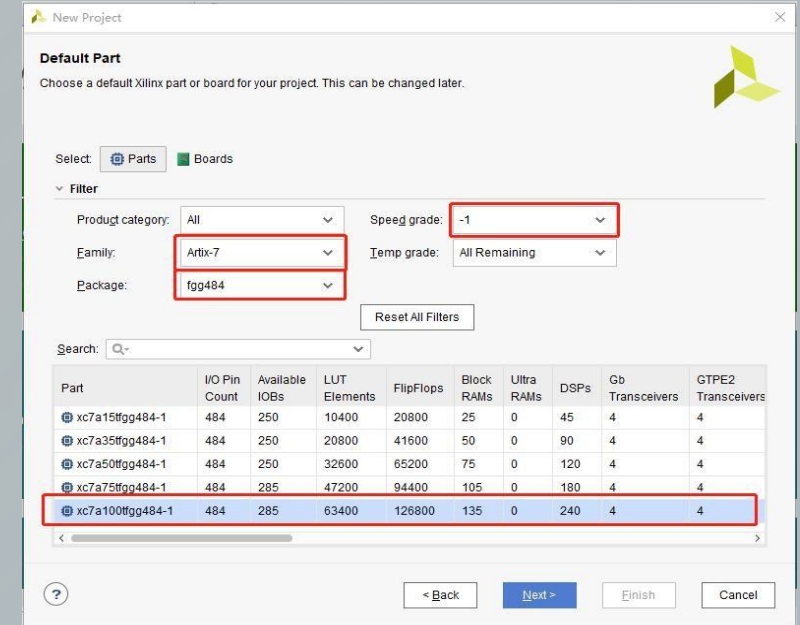
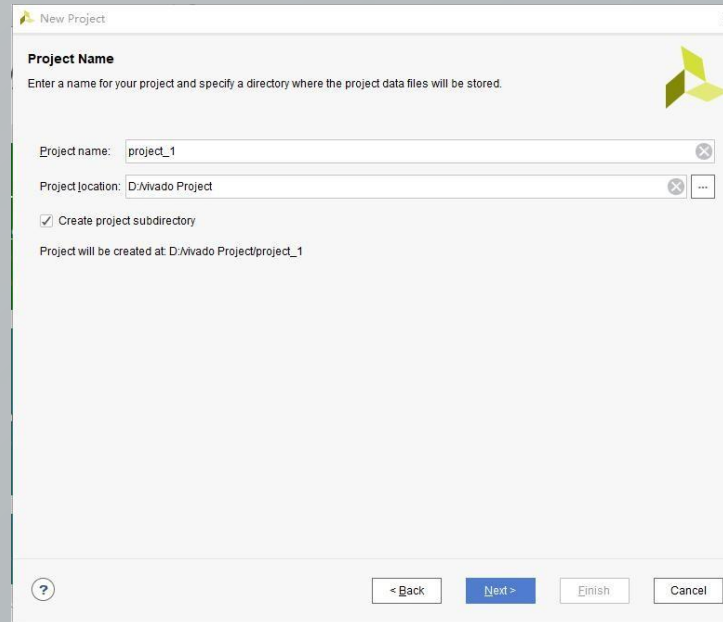
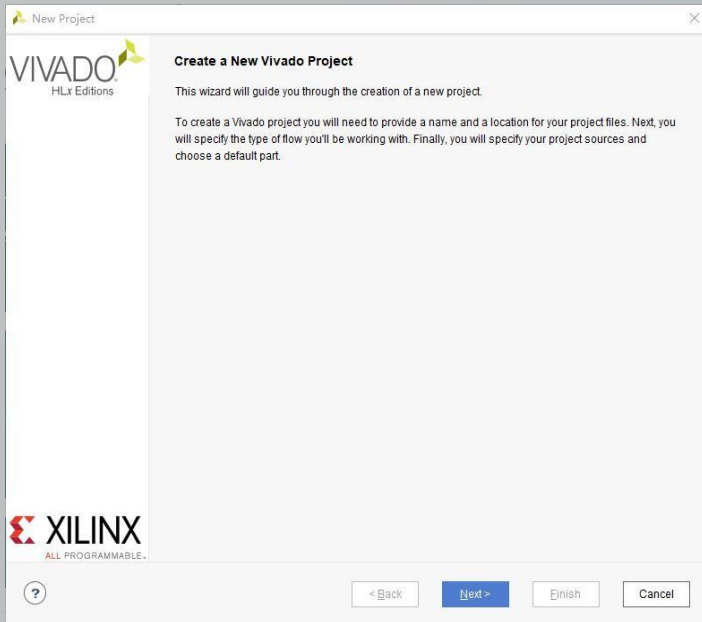
3. Connect with FPGA chip

4. Program the FPGA chip





# Creat and set a vivado project



## 1. Create project

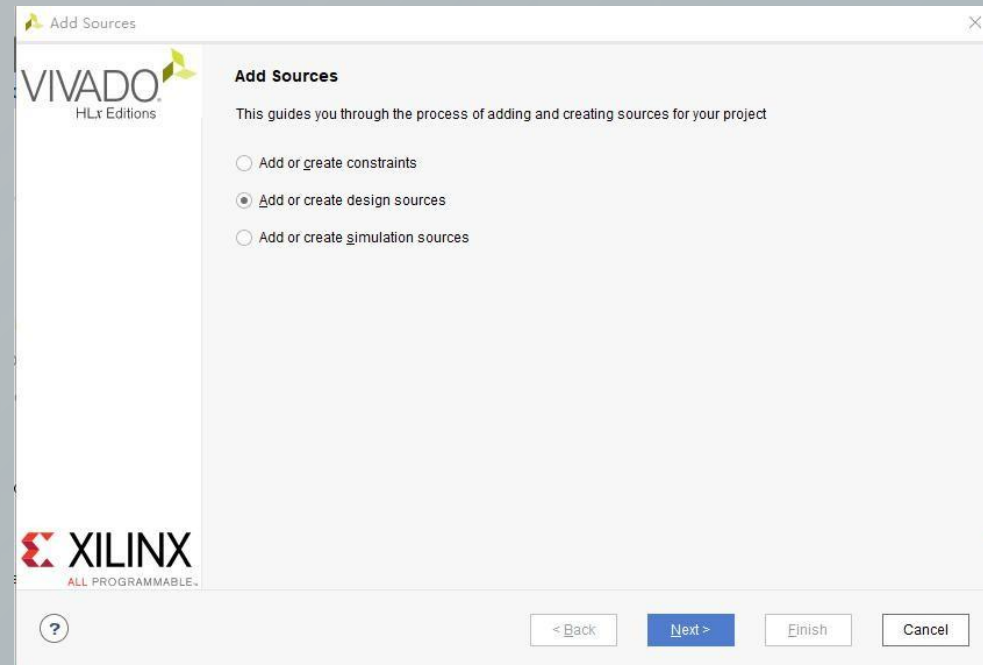
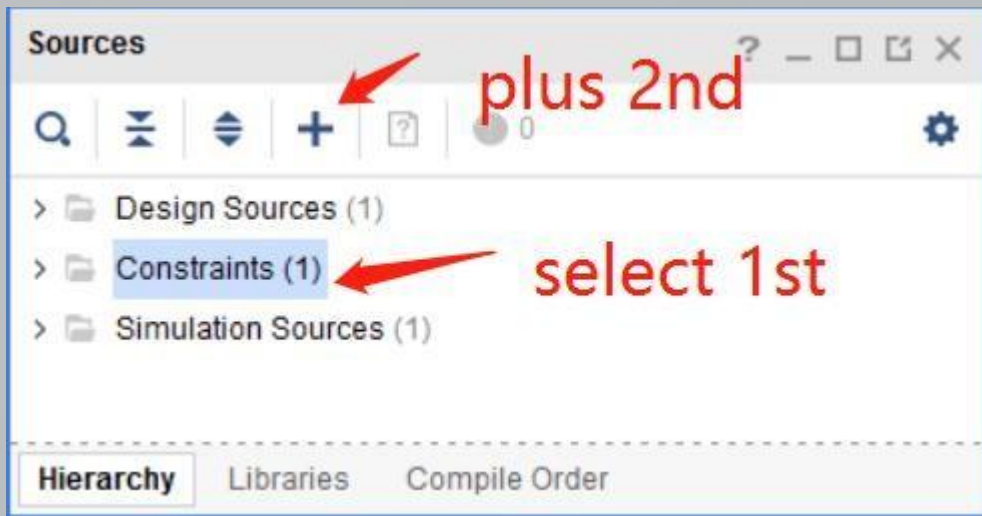
1) Select "rtl type" as project type

2) Select the xilinx board(based on the FPGA chip type embedded in the board), the settings about the xilinx board could be reset after the project is built.



# Using VIVADO continued

## 2. Add files to vivado project design file(s), simulation file(s) and constraint file(s)



# Using VIVADO continued



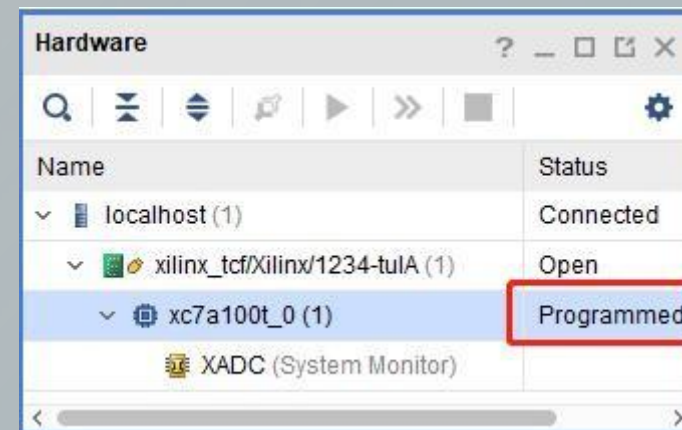
## 3. Following the steps to generate bitstream file.

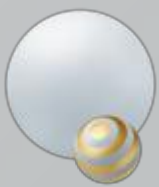
- 1) Do the **simulation** to verify the Circuit function[**step1**]
- 2) After simulation, a waveform file is generated which records the states of input and output signals.
- 3) If the function of circuit is ok, **Run synthesis[step2]** , then **Run implements[step3]**
- 4) After implementation is finished, **Generate Bitstream[step4]**, the generated ".bit" file could be used to program device later.



# Using VIVADO + MINISYS

4. **Connect the developing board**(with FPGA chip embeded) with PC (USB JTAG interface) which run the vivado project
5. **Turn on** the power of minisys **board**
6. Click on “**Open Target** ” of **vivado Project** to **connect** the vivado project with the 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 the board is on, it means the **bitstream** file has been **written** into the device(the FPGA chip in the board has been **programmed**).
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(<http://iverilog.icarus.com/>) for temporary use. Or use the iverilog tools on line: <https://hdlbits.01xz.net/wiki/Iverilog>  
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** (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.
```



# EDA2.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

- ```
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
```

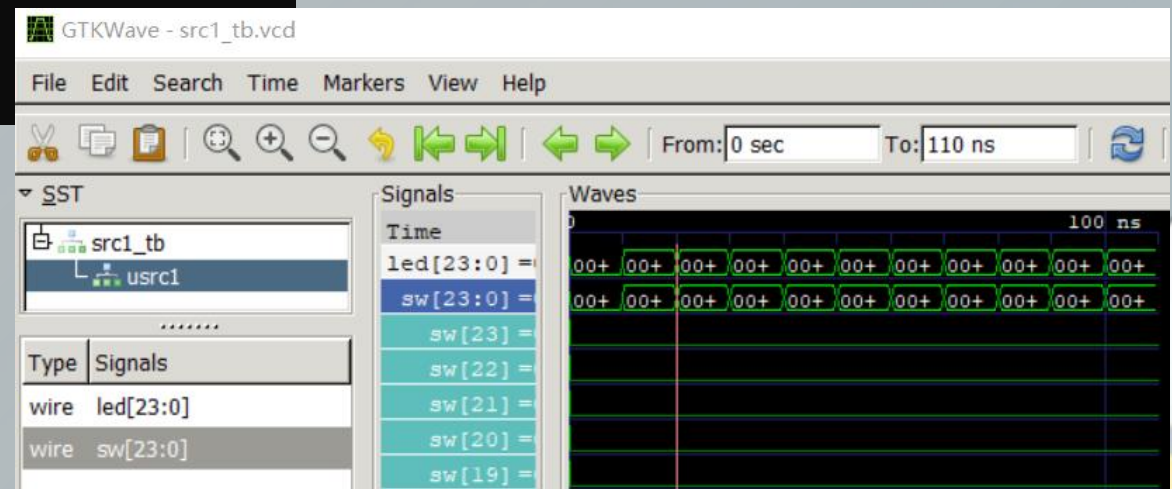
```
VCD info: dumpfile src1_sim.vcd opened for output.
```

```
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);
```





- Q2 : Test the circuit(adder/substraction with overflow detection) on the board.
  - The circuit of practice on lab 6 could be reused here.

