Computer Organization

Lab8 Verilog and EDA tool

Verilog and EDA tool

Topic

- **Verilog**
  - A kind of **H**ardware **D**escription **L**anguage
  - **module**, **block**, **statement**, **operator**, **data**
  - **suggestion**

- **EDA tool**
  - **vivado(1)**
    - **project, design(IP cores), testbench**
    - **simulation**
    - **IP core**

- **Practices**
  - Instruction & Analysis

Verilog

- **Design-Under-Test vs Test-Bench**
  - Structured Design(top module, instance module)
- **Block**
  - Combinational, Sequential
- **Statement**
  - Continuous assignment
  - Procedure assignment: Non blocking assignment vs Blocking assignment
  - if - else, case, loop
- **Operator and data**
  - Variable vs Constant
  - Reg vs Wire, Splicing { , }, Number system
- **Verilog suggestion**

DUT vs Testbench

- **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
```

```
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
```

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
```

Module (Circuit 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:

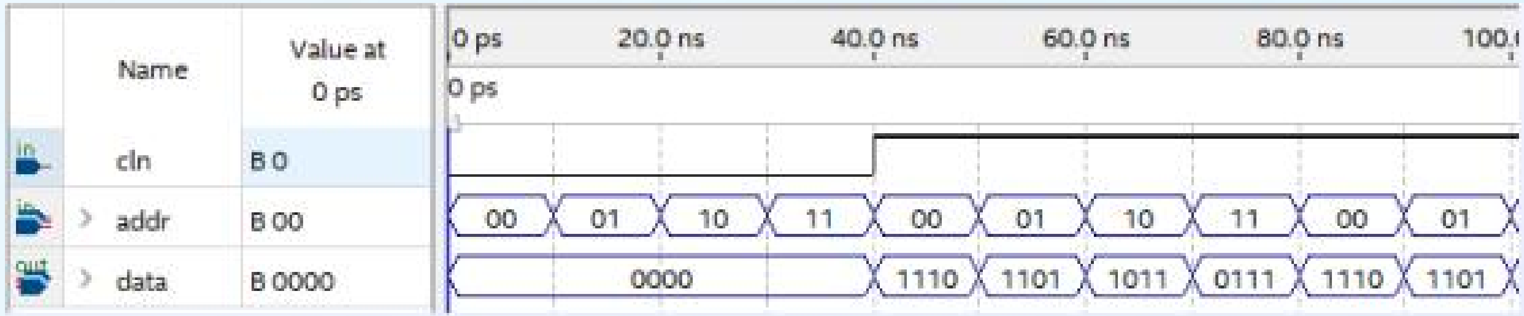
1) If there is no 'else' branch in the statement, latches will be generated while doing the synthesis.

2) Nested 'if-else' is NOT suggested, 'case' is suggested as an alternative(more clear while reading, less latency).

Behavior Modeling(case)

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

```
module decorder(cln,data,addr);
input cln;
input [1:0] addr;
output reg [3:0] data;

always @(cln or addr )

begin

if(0==cln)
   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
```



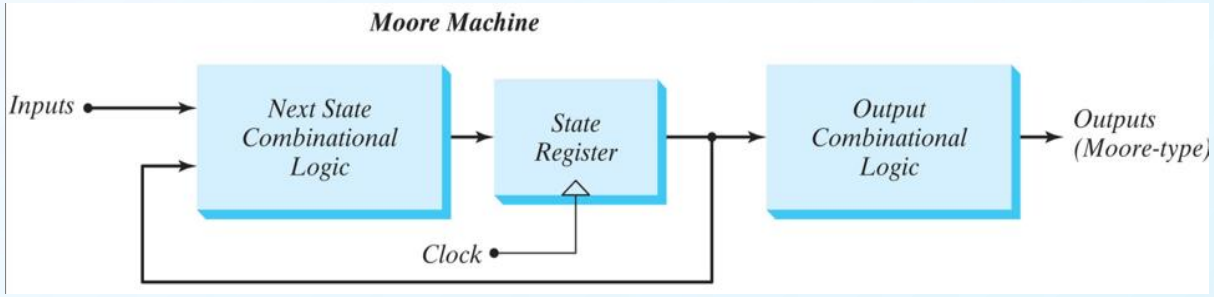
NOTIC:

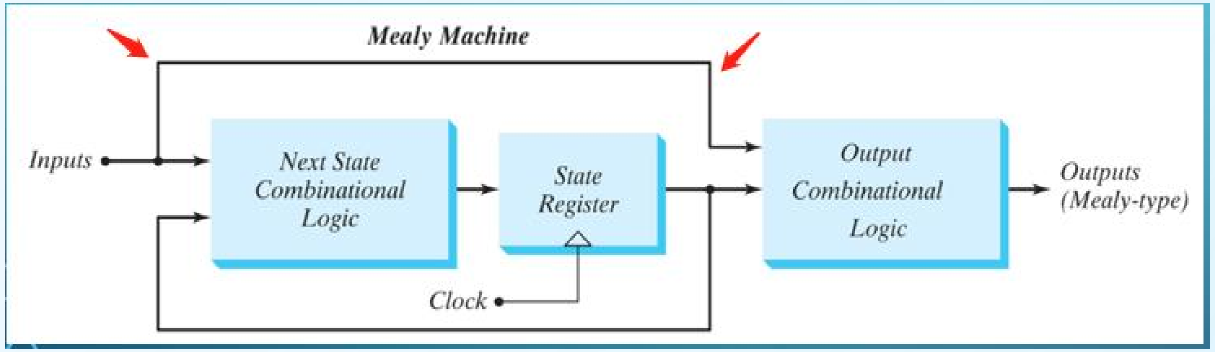
Without defining 'default' branches and declearing all situations under the “case”, latches will be generated while doing the synthesis.

Sequential Circuit: FSM

The piece of verilog code(on the right hand) is a **2-stage** code(*using **2 always block** decribe the **combinational** logic and **sequential** logic separately*).
Does it describe the **Moor-type** FSM or **Mealy-type** FSM?

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






```
`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**
  - 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
```

```
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
```

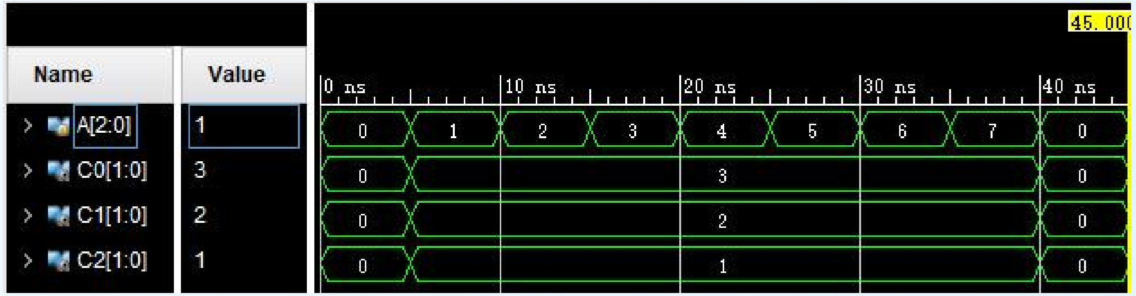
```
Error: procedural assignment to a non-register in2 is not permitted, left-hand side should be reg/integer/time/genvar
```

```
sim_1 (2 errors)
    [VRFC 10-529] concurrent assignment to a non-net o1 is not permitted [test.v:29]
```

```
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
```

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
endmodule
```

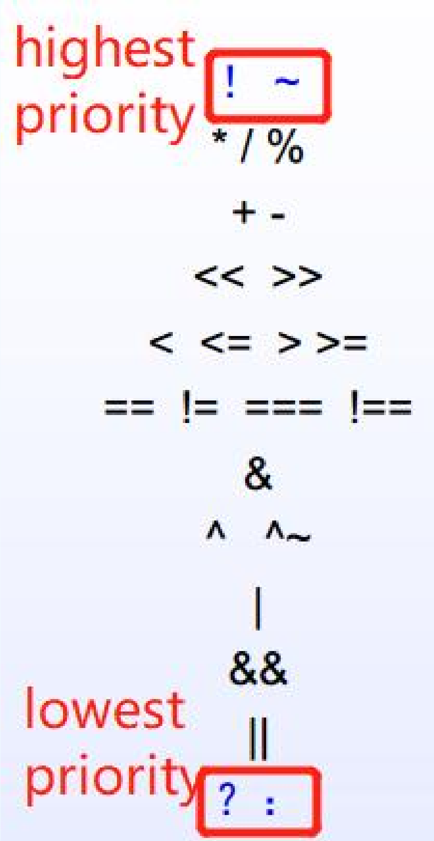


Q1: Does the waveform belongs to the two test?
If not, which one does it belong to?

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
endmodule
```

Operator



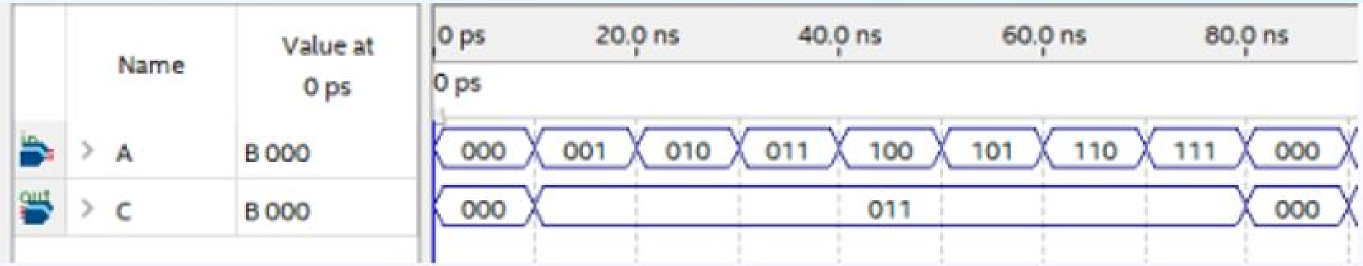
Bit splicing operator { }

- 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}

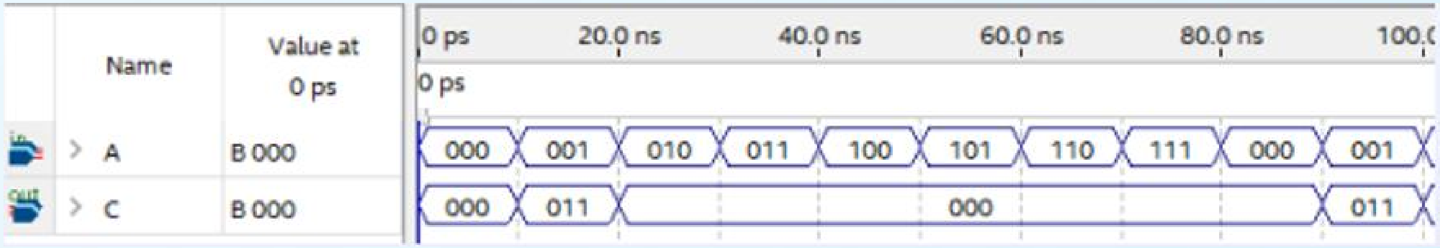
Operator continued

```verilog
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?




```verilog
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

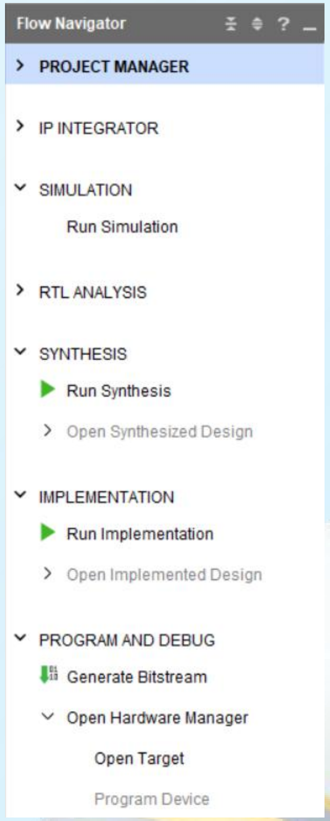
- **Non-Synthesizable Verilog** which **is NOT suggested** to use in your design
  - initial;   Task, function;   System task： $display, $monitor, $strobe, $finish
  - fork... join; UDP
  - using "#number" as delay
- **Suggested**
  - Using an asynchronous reset to make your system go to initial state
  - 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

- **Installation**
  - It' s suggest to recall to pages 5 to 9 of document "Digital design lab1.pdf"

    Tips: Digital design lab1.pdf" file could be found in the directory "**labs/lab8**" of course blackboard site

- **Vivado project**
  - Vivado files
    - vivado project file, design files, testbench files, constraints files…
  - Vivado process flow
    - option1: design, simultaion
    - option2: design, simultaion, synthesise, implemention, generate bitstream file, open target, program device



- **Vivado IP cores**
  - Block Memory(ROM)

Vivado process flow

1. Do the design with verilog (Vivado)

2. Do the simulatin 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)

At the very beginning, a vivado project is needed!

Vivado project

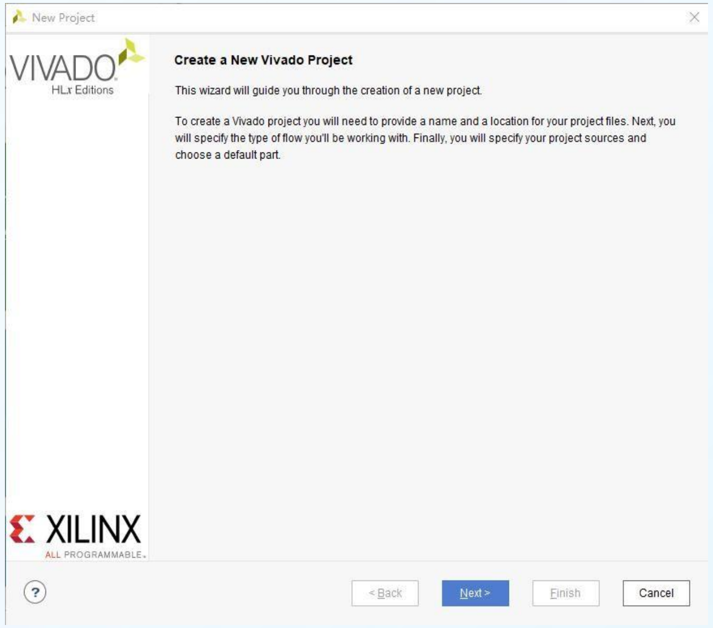
1. Manage all the files
(including design file, simulation file, constraint file and other resource file)

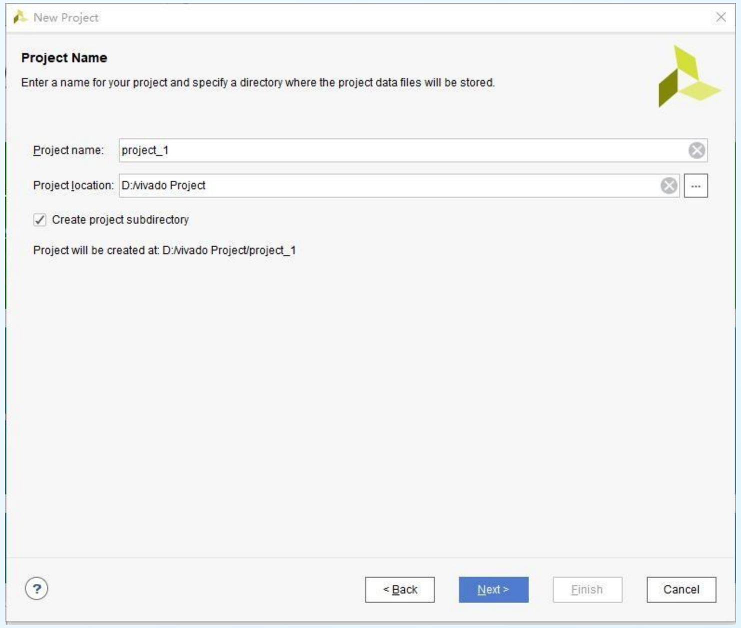
2. Manage the operation flow

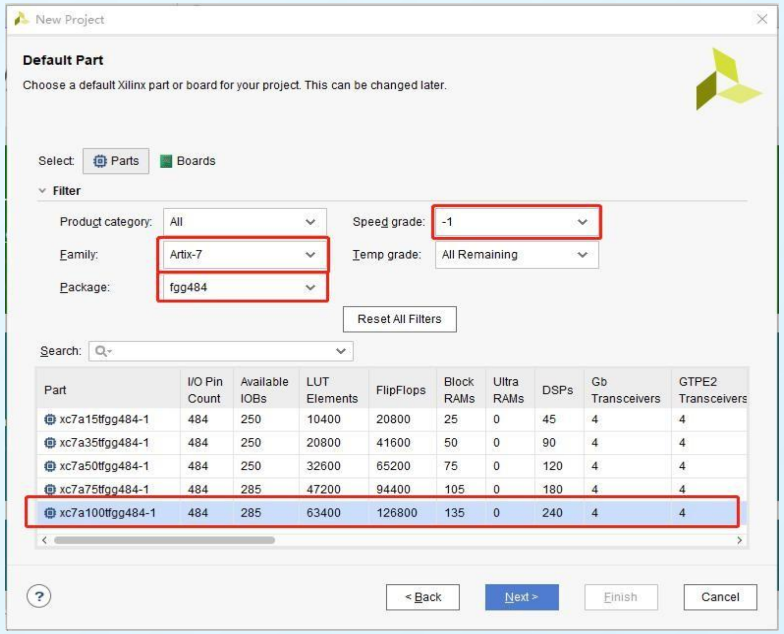
3. Connect with FPGA chip

4. Program the FPGA chip

Creat and set a vivado project







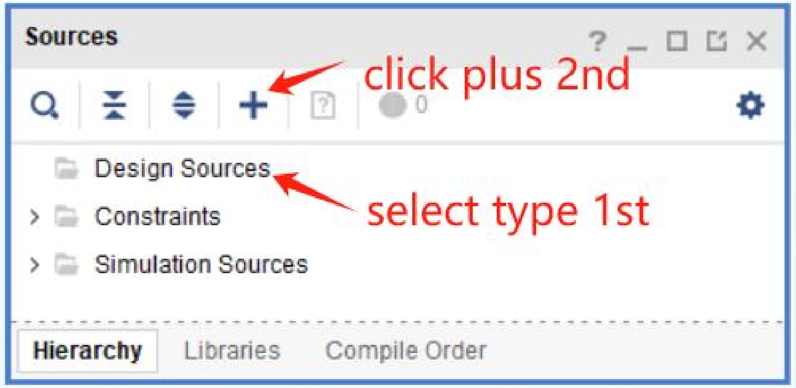
1. Create project

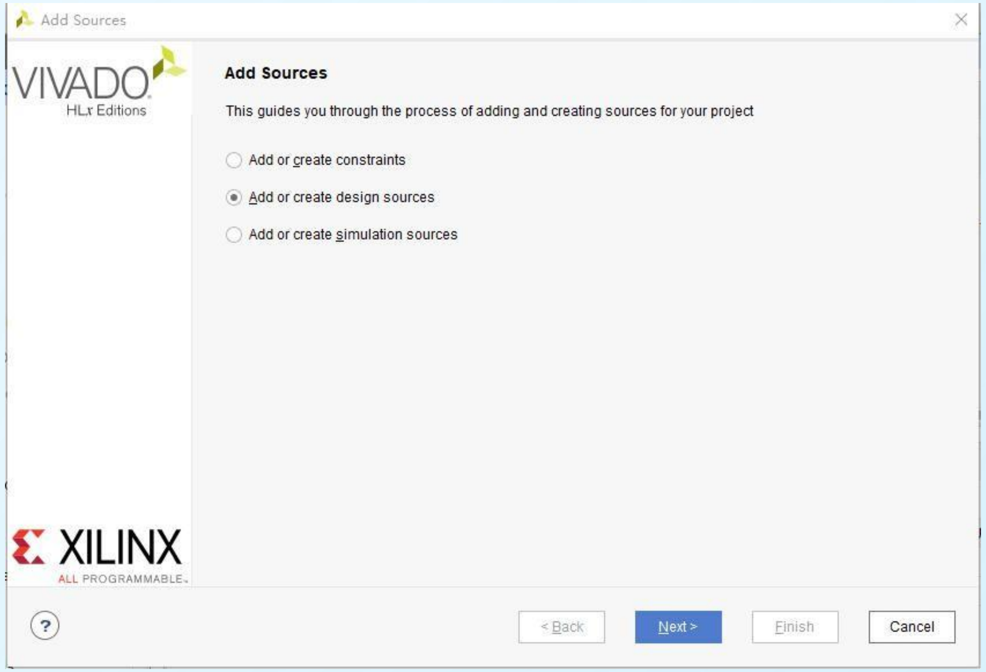
1) Select “**rtl type**” as project type

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

Using VIVADO continued

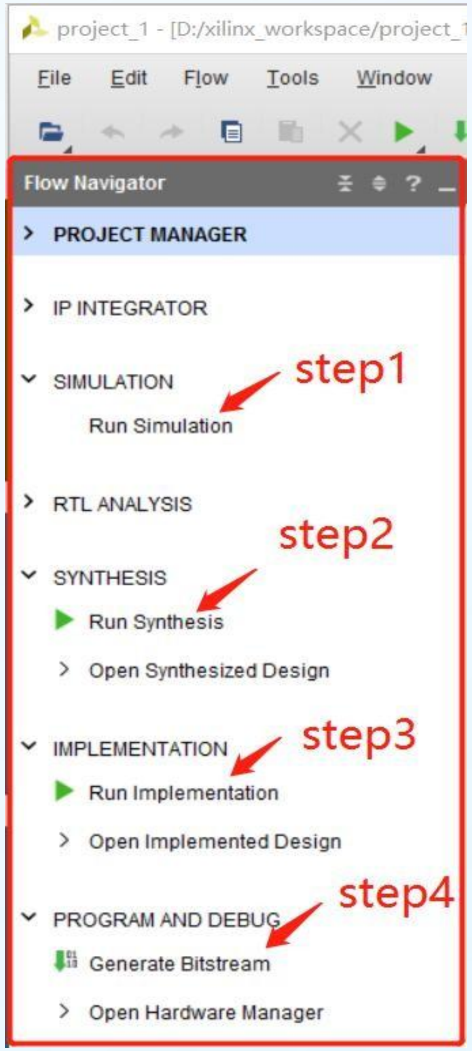
2. Add files to vivado project
design file(s), simulation file(s) and constraint file(s)





Tips: In this experiment, since there is no need to use the development board for testing, only design and simulation files need to be added to the vivado project, no constraint files need to be added.

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**]

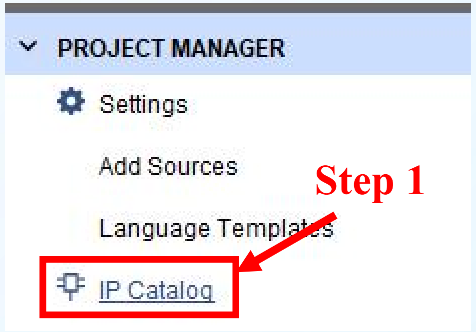
4) After implementation is finished, **Generate Bitstream**[**step4**], the generated ".bit" file could be used to program device later.

Tips:If you need to test circuits on the development board, you must set the chip type in the project, add design files and constraint files, then do the synthesise, implemention, and finally generate a bitstream file which is used for programming the FPGA chip embedded on the development board.

In this experiment, as there is no need for testing on the development board, design and simulation files were added to the project for simulation to verify the functionality of the circuit.

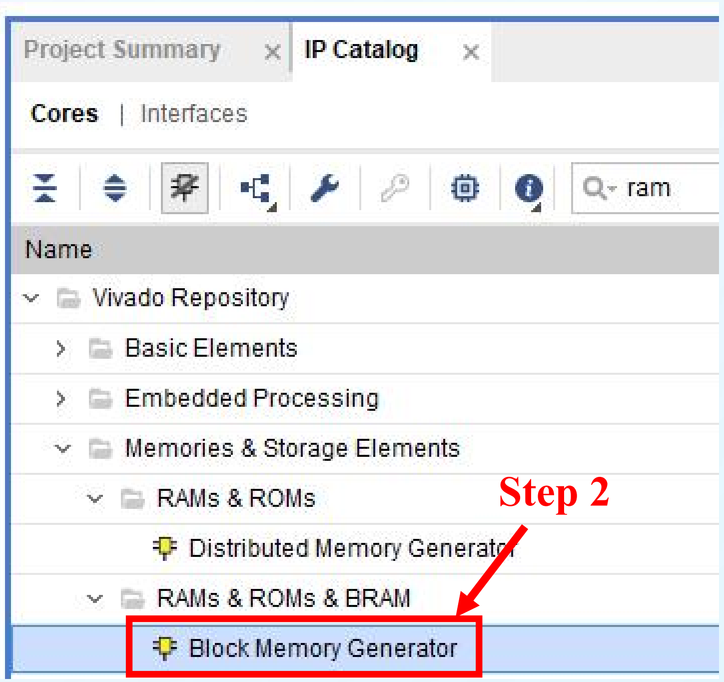
Using IP core in Vivado: Block Memory

Using the **IP core** '**Block Memory**' of Xilinx to implement the Data-memory.



Import the **IP core** in vivado project

1) in **"PROJECT MANAGER"** window
click **"IP Catalog"**



2) in "**IP Catalog**" window

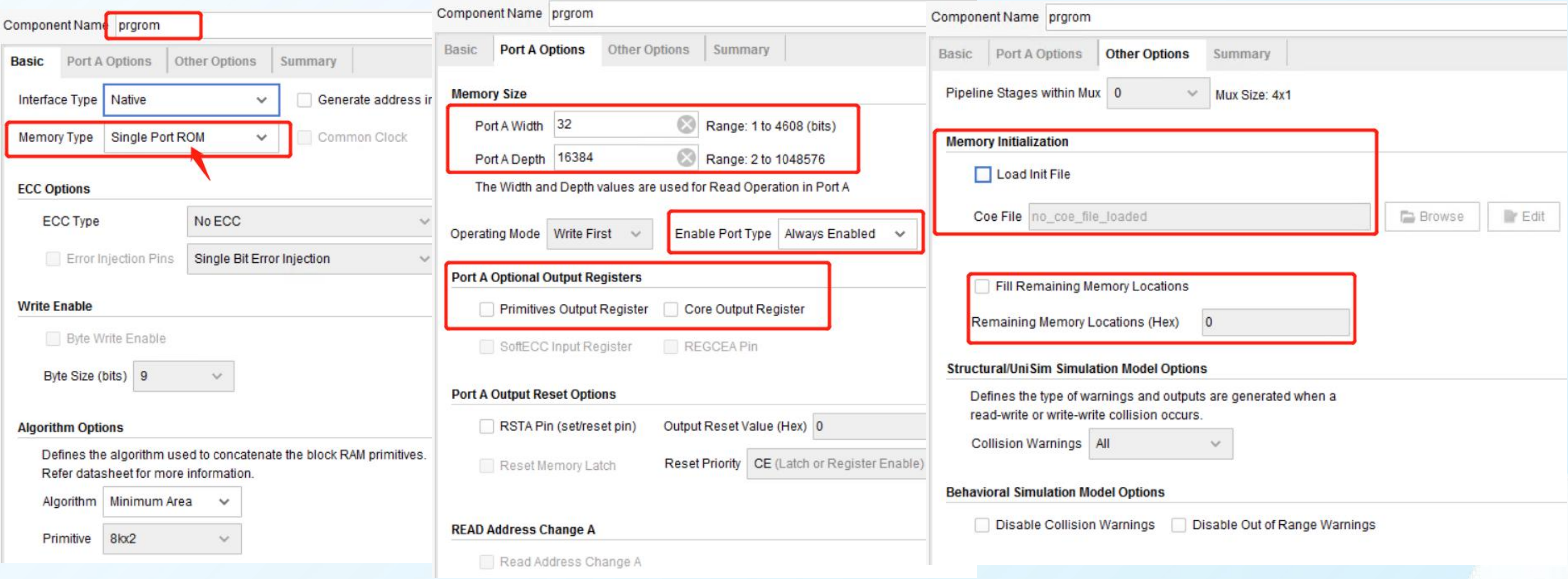
> Vivado Repository

> Memories & Storage Elements

> RAMs & ROMs & BRAM

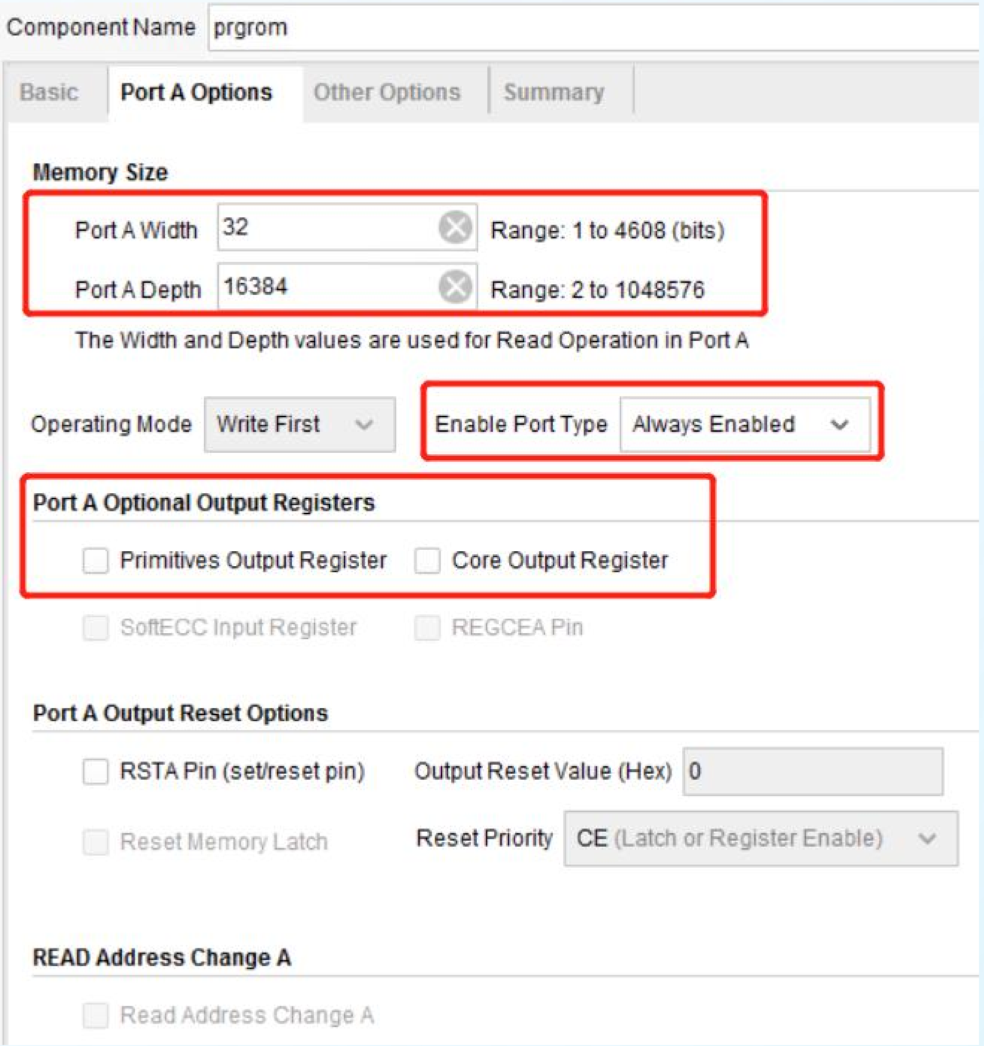
> **Block Memory Generator**

Customize Memory IP core



NOTE: set the init file of prgrom after this IP core has been added into vivado project.
Same steps as the RAM IP core used in Data-memory.

Customize Memory IP core continued



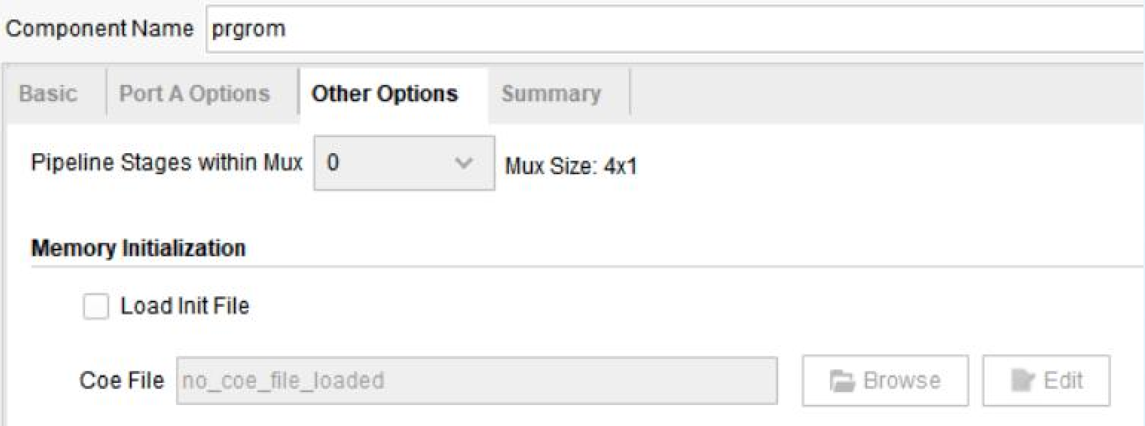
3) **PortA Options** settings:

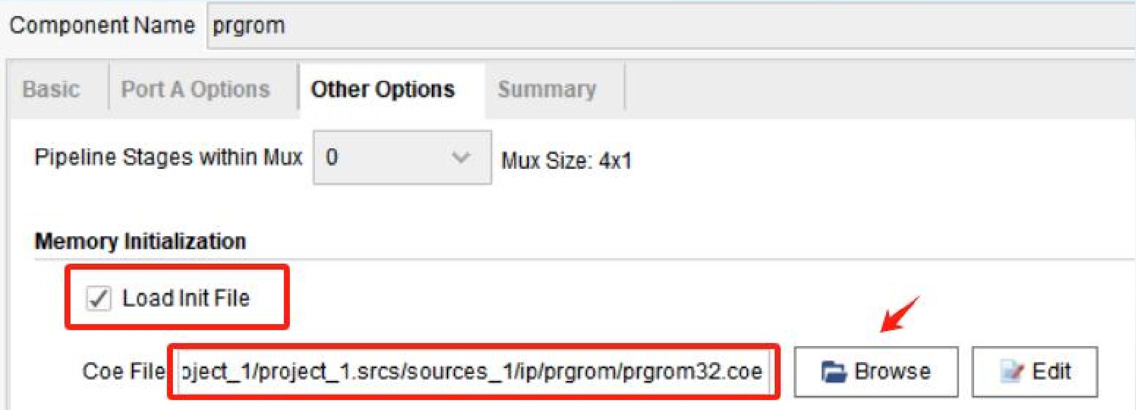
- Data read and write **bit width**: **32 bits (4Byte)**
- Read **Depth**: **16384, size: 2^{14} * 4Byte = 64KB**
- Operating Mode: **Write First**
- Enable Port Type: **Always Enabled**
- PortA Optional Output Registers: **NOT SET**

Customize Memory IP core continued

4) **Other Options** settings:

- **1.** When **specifying the initialization file** for customize the ROM on the 1st time, the IP core ROM just customized WITHOUT initial file and **corresponding path**, so set it to no initial file when creating ROM.
- **2. After** the ROM IP core created
  - **2-1. COPY** the initialization file prgrom32.coe **to** projectName.srcs/sources_1/ip/ComponentName. (“projectName.srcs” is under the project folder, “componentName” here is ‘prgrom’)
  - **2-2.** Double-click the newly created ROM IP core, **RESET** it with the **initialization file**, select the prgrom32.coe file that has been in the directory of projectName.srcs/sources_1/ip/prgrom.

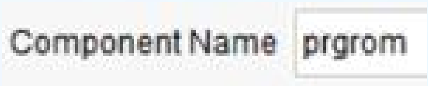




Tips: “**prgrom32.coe**” file could be found in the directory “**labs/lab8**” of course blackboard site

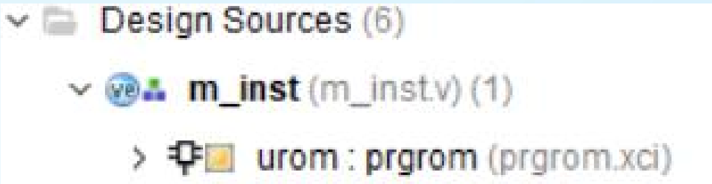
Instance the Memory IP core

Step1. Find the name and the ports of the IP core:



Step2. Build a module in verilog to instance the IP core and bind its ports:

```
module m_inst(
clk, addr,dout
    );
    input clk;
    input [13:0] addr;
    output [31:0] dout;
    prgrom urom(.clka(clk),.addra(addr),.douta(dout));
endmodule
```



Test the IP core

Step1.Build the testbecn to verify the function of the IP core.

```
module tb_inst_mem( );
reg clk;
reg [13:0] addr;
wire [31:0] dout;
m_inst urom(.clk(clk),.addr(addr),.dout(dout));

initial begin
   clk = 1'b0;
   forever  #5 clk = ~clk;
end

initial begin
   addr=14'h0;
   repeat(20) #17 addr = addr + 1;
   #20 $finish;
end

endmodule
```

Step2. do the simulation based on the testbench.

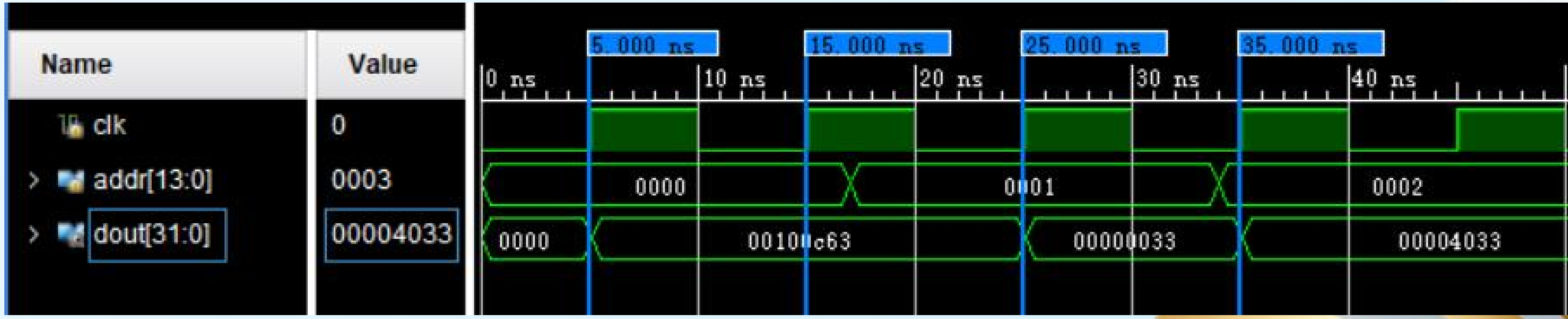
Step3. Check the waveform generated by the simulation and the coe file which used to initialize the IP core to check if the prgrom IP core work as a ROM(Read Only Memory):

Determine whether the module can accurately read the data stored in the corresponding storage unit in the ROM based on the address on the rising edge of the clock.

The prgrom IP core is **initialized with file prgrom.coe**

```
memory_initialization_radix = 16;
memory_initialization_vector =
00100c63,
00000033,
00004033,
00002297,
ff428293,
00028083,
fff07013,
ffe00013,
00129023,
fddff06f,
00000000,
```

prgrom32.coe



Practice 1

- Q1: Please refer to the instruction format of RISC-V32I and using Verilog to achieve the following functions:
  - Build a circuit design module in verilog to analysis the RISC-V32I **instruction**(as inputs), and determine the value of **writeR**, **writeM**, **imm** and **imm32**(as outputs), **imm is optional**.
    - The bitwidth of **writeR** is 1, when its value is 1, it indicates a write operation to the Register, and when its value is 0, it indicates no write operation to the register
    - The bitwidth of **writeM** is 1, when its value is 1, it indicates a write operation to the Memory, and when its value is 0, it indicates no write operation to the Memory
    - The bitwidth of **imm** is 20bit(optional)
      - while the type of RISC-V32I instruction is I, S, SB, the 12 bit immediate value extracted from the instruction is stored at the lower 12 bits of **imm**. How about the remaining high bits?
      - while the type of RISC-V32I instruction is U, UJ, the 20 bit immediate value extracted from the instruction is stored at 20bits of imm.
    - The bitwidth of **imm32** is 32bit
      - imm32 is the extended 32bit immediate based on the original immediate extracted from the instruction. NOTE: for SB or UJ instructions, shift the original immediate (which is extracted from the instruction) to left by one bit and extend.
  - Build a testbench and verify through simulation on it to check whether the functionality of the circuit design module is correct.

Tips: The type of the circuit is a combinational design. Using “RISC-V-Reference-Data.pdf” as a reference for the design.

Practice 1

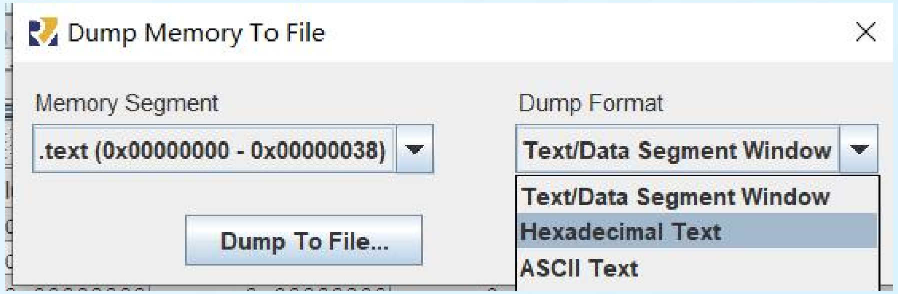
Tips(1):

1. To build the testcases, it is suggested to :

- ✓ 1-1. build a RISC-V assembly soure file in which there is 6 types(R,I,S,SB,U and UJ) of the RISC-V32I instructions.
- ✓ 1-2. Using Rars to asemble the source file, and generate the machine code.
- ✓ 1-3. Dump the machine code as Hexadecimal Text.
- ✓ 1-4. Open the file generated in step 1-3, copy a line and past it to the testbench as the input of the desgin circuit.

test_instr.asm

```
.data
idata: .word 0x12345678
.text
main:
beq x0,x1,here
add x0,x0,x0
xor x0,x0,x0
la t0,idata
lb x1,(t0)
```



Code	Basic	Source
0x00100c63	beq x0,x1,0x00000018	5: beq x0,x1,here
0x00000033	add x0,x0,x0	6: add x0,x0,x0
0x00004033	xor x0,x0,x0	7: xor x0,x0,x0
0x00002297	auipc x5,2	8: la t0,idata
0xff428293	addi x5,x5,0xfffffff4	
0x00028083	lb x1,0(x5)	9: lb x1,(t0)

test_instr_text0 - 记事本

文件(F) 编辑(E) 格式(O) 查

```
00100c63
00000033
00004033
00002297
ff428293
00028083
```

```
initial begin
inst = 32'h00100c63;
#10
inst = 32'h00000033;
#10
inst = 32'h00004033;
#10
inst = 32'h00002297;
```

Practice 2

- Q2: Set the IP core of the Block memory type, observe and compare the circuit behavior of the IP cores.

  - Step1.

  Refer to pages 24 to 27 of the courseware to create a block ROM IP core, except for modifying the option settings marked by the blue arrow in the right image, all other settings remain unchanged.

  - Step2.

  Instantiate and simulate the IP core under this configuration to observe the differences in their behavior under different settings

- Tip: For a ROM IP core, it' s suggested to pay attention to the data storage, data reading, and related timing.