Introduction to Digital Circuit Design Using Verilog

BLG 222E - Computer Organization

T. Tolga Sarı, Kadir Özlem, Tuğba Pamay, Mücahit Altıntaş
{sarita, kadir.ozlem, pamay, maltintas}@itu.edu.tr

Istanbul Technical University

February 25, 2022

Outline

Hardware Description Language (HDL)

- Describe hardware using code.
- No need to define what every transistor/gate does when coding.
- Simulate logic before building. Running simulations are quite faster than wiring manually.
- Port previously developed module to another project easily.
- Write parametric units. 8-bit multiplier $\rightarrow$ 32-bit multiplier.
- Synthesize code into gates and layout (with help of a standard cell library)
- Verilog, VHDL, SystemVerilog...

Field Programmable Gate Array

- After creating the design, it can be ported to the ASIC(application specific integrated circuit) which is a whole different level of design process or, the design can be realized using FPGA(field programmable gate array). Which is quite easier process.
- An FPGA is basically programmable digital circuit and it consists of LUT(look up table) which is basically RAM tied to the multiplexer and programmable routing logic.

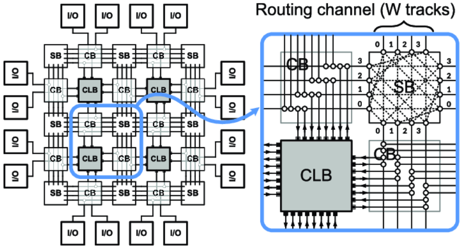


Figure 1: Inside of the FPGA.

Verilog

- One of the mainstream HDL.
- Has syntax similar to C programming language $\rightarrow$ More easier to adapt. Also it needs less typing compared to other languages.
- Allows hardware designer to describe language both at high levels (if-else, for-while, switch-case) and at low levels of abstractions.
- Allows programmers to create macros and functions.

General Module Structure

```
module <name>(<port list>);
    <Declarations>

    <Module Items>
endmodule
```

General Module Structure

- Module Name is basically function name.
- Inside the port list a list of input, inout and output ports which are referenced in other modules.
- Section specifies data objects as registers, memories and wires as well as procedural constructs such as functions and tasks.
- Assignments, logical blocks, initial constructs are given inside the Module Items
- Let's look at an example code for the AND gate.

Simple AND Gate

```
module and_gate(i_1, i_2, o);
    // Inputs
    input   wire        i_1;
    input   wire        i_2;

    // Outputs
    output  wire        o;

    assign o = i_1 & i_2;
endmodule
```

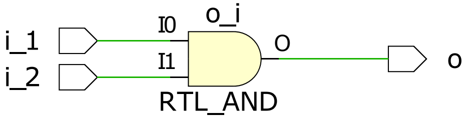


Figure 2: RTL Schematic of AND Gate Module

Simple AND Gate

- At first, module name and its connections are defined (Line 1).
- Then the connection directions and the data type is defined (Lines 3-7). Here A and B are the inputs of the **and_gate** and the output will be C.
- **wire** data type is the most basic data type in Verilog. It can be treated as psychical wires.
- Lastly these wires are connected using **assign** keyword.

Using a Module Inside Another Module

- We don't need to re-code frequently recurring code parts in HDL. We can use them anywhere of a module given that the connections are correct.
- This will help us to divide the digital design into smaller and less complex parts.
- Also good design on the paper $\rightarrow$ less headache when coding the hardware.

Using a Module Inside Another Module

```
module module_name(wire1, wire2);
    ...
    ...
    and_gate G1(.i_1 (some_wire_1),
                .i_2 (some_wire_2),
                .o   (some_wire_3));
    ...
    ...
endmodule
```

Simple Sum of Products

- Let's create a digital circuit for $D = AB + BC$.
- This can be done by either first creating an OR gate and combining them in another module, or realizing this function in another module.
- Both methods will be demonstrated.

Simple Sum of Products

```
module or_gate(i_1, i_2, o);
    // Inputs
    input   wire          i_1;
    input   wire          i_2;

    // Outputs
    output  wire          o;

    assign o = i_1 | i_2;
endmodule
```

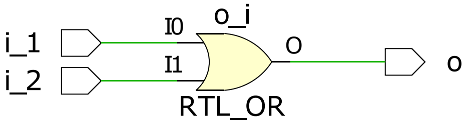


Figure 3: RTL Schematic of OR Gate Module

Simple Sum of Products

```
module SOP(A, B, C, D);
    // Inputs
    input   wire        A;
    input   wire        B;
    input   wire        C;

    // Outputs
    output  wire        D;

    // Intermediate Wires
            wire        AB;
            wire        BC;

    and_gate AND1(.i_1(A ), .i_2(B ), .o(AB));
    and_gate AND2(.i_1(B ), .i_2(C ), .o(BC));
    or_gate   OR1(.i_1(BC), .i_2(AB), .o(D) );
endmodule
```

Simple Sum of Products

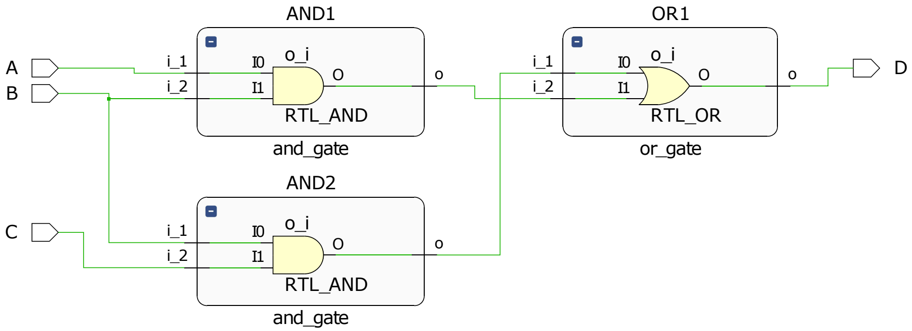


Figure 4: RTL Schematic of SOP Module

Simple Sum of Products

- This can be done by coding the boolean function in one line also:

```
module SOP(A, B, C, D);
    // Inputs
    input   wire        A;
    input   wire        B;
    input   wire        C;

    // Outputs
    output  wire        D;

    assign D = (A & B) | (B & C);
endmodule
```

Simple Sum of Products

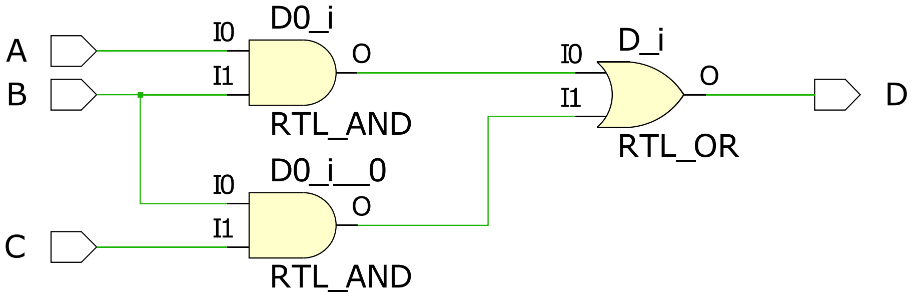


Figure 5: RTL Schematic of Alternative SOP Module

Simple Sum of Products

- So, we created our module. How can we simulate it?
- We need to create a test bench in which we send input signals to our module.
- The test bench code structure is fairly simple and follows the general code structure.
- We basically create a module that has our **SOP** and applies inputs to this module at specific times.
- The inputs of the uut(unit-under-test) will be data type called **reg** and the output will be **wire** data type. **reg** data type will be discussed later.

Simple Sum of Products

```
module SOP_test();
    // Test module has no inputs and outputs.
            reg          a;
            reg          b;
            reg          c;

            wire         d;
    SOP uut(.A(a), .B(b), .C(c), .D(d));

    initial begin
    // Assign test values and wait 250 time units.
        a=0; b=0; c=0; #250;
        a=0; b=1; c=0; #250;
        a=1; b=0; c=1; #250;
        a=1; b=1; c=1; #250;
    end
endmodule
```

Simple Sum of Products Simulation

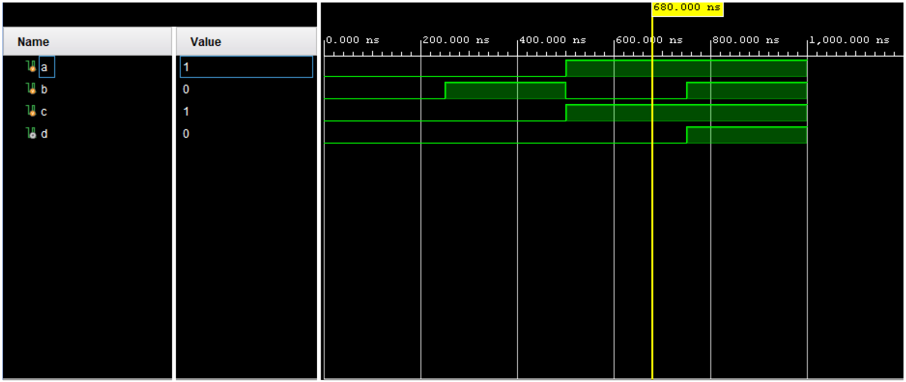


Figure 6: Simulation Timeline for SOP Test Module

Syntax Outline

- Let's discuss more detailed syntax of the Verilog.
- We will discuss more indept operators.
- Multi bit operands.
- **reg** data type and initial block.
- Always block

List of Operators

Verilog Operator	Name
&	Bitwise AND
|	Bitwise OR
~	Bitwise NOT
^	Bitwise XOR
+	Arithmetic Addition
-	Arithmetic Subtraction
*	Arithmetic Multiply
/	Arithmetic Division
<<	Logical Left Shift
>>	Logical Right Shift
{}	Concatenation

Table 1: Commonly used Verilog Operators.

Multi Bit Data

- Until now, we have only dealt with the 1-bit data.
- Using the multi bit data is fairly simple with the Verilog. Only needed thing is specifying the bit length of the data.
- For example let's look at 32-bit AND gate.

Multi Bit Data

```verilog
module and_gate_32(i_1, i_2, o);
    // Inputs
    input   wire [31:0] i_1;
    input   wire [31:0] i_2;

    // Outputs
    output  wire [31:0] o;

    assign o = i_1 & i_2;
endmodule
```

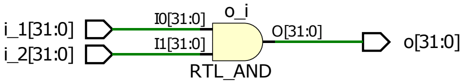


Figure 7: RTL Schematic of 32-bit AND Gate Module

Multi Bit Data

- Numbers in Verilog can be specified in many ways. All of them needs bit count and base that is used. Let's look at some examples.

```
    ...
    // foo and bar are 8-bit numbers.
    // All of the things below mean the same thing.
    // Which is foo = bar + 153
    assign foo = bar + 8'b10011001;
    assign foo = bar + 8'b1001_1001;
    assign foo = bar + 8'h99;
    assign foo = bar + 8'd153;
    ...
```

- Let's finish this section by looking at the testbench of the 32-bit and gate.

Testbench for 32-bit AND Gate

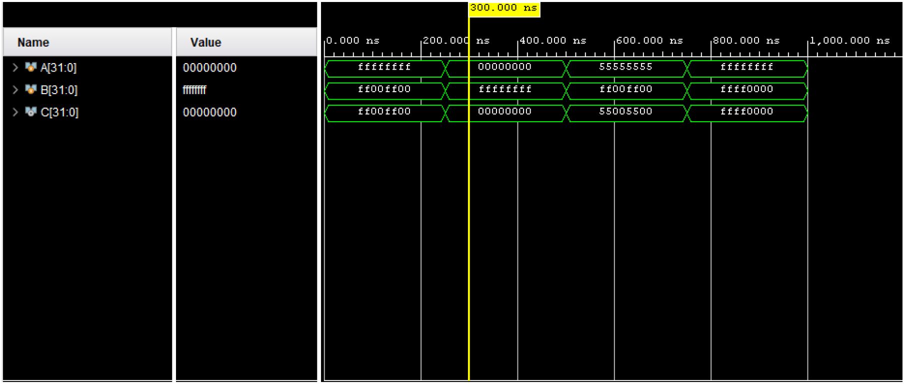
```
module and_gate_32_test();
    // Test inputs
            reg [31:0] A;
            reg [31:0] B;
    // Test outputs
            wire[31:0] C;

    and_gate_32 uut(.i_1(A), .i_2(B), .o(C));

    initial begin
        A = 32'hFFFFFFFF; B = 32'hFF00FF00; #250;
        A = 32'h00000000; B = 32'hFFFFFFFF; #250;
        A = 32'h55555555; B = 32'hFF00FF00; #250;
        A = 32'hFFFFFFFF; B = 32'hFFFF0000; #250;
    end

endmodule
```

32-bit AND Gate Simulation



Name	Value	0–250 ns	250–500 ns	500–750 ns	750–1,000 ns
A[31:0]	00000000	ffffffff	00000000	55555555	ffffffff
B[31:0]	ffffffff	ff00ff00	ffffffff	ff00ff00	ffff0000
C[31:0]	00000000	ff00ff00	00000000	55005500	ffff0000

Figure 8: Simulation Timeline for 32-bit AND Gate Module

Bit Values

- Aside from logical high and low, the bits in a variable can take more values in the Verilog.
- The bit can be in high impendence mode, which is donated as 32'dZ for 32-bit data.
- The bit value can be undetermined which is donated as 32'dX for 32-bit data.
- That is why even if we define behaviour for all bit combinations we still have to create a default case(we will discuss switch case in the following slides).

reg Data Type

- **reg** data type is more capable data type that can be used in more complex code blocks in Verilog.
- It can be used to define flip-flops and other memory elements such as RAM or Cache.
- Unlike its name, using reg data type **will not generate flip-flops always**.
- Its power lies in the **always** code blocks.

always Block

- **always** block consists of 2 parts.
- It has a sensitivity window that determines when the updates in the always block will be applied. For example always block can be sensitive to the falling edge of the clock.
- After defining the sensitivity window, functionality of the always block will be coded.
- In always code block, more high level abstractions(if-else, switch-case) can be used.
- **ONLY THE REG DATA TYPE CAN BE UPDATED IN THE ALWAYS BLOCK.**

Example Counter

```
...
        reg [31:0] A;
...
// Simple counter that counts up by using
// ADDER In each positive edge of the clock.
always @(posedge clock)
begin
    A <= A + 32'd1;
end
...
```

Example Counter

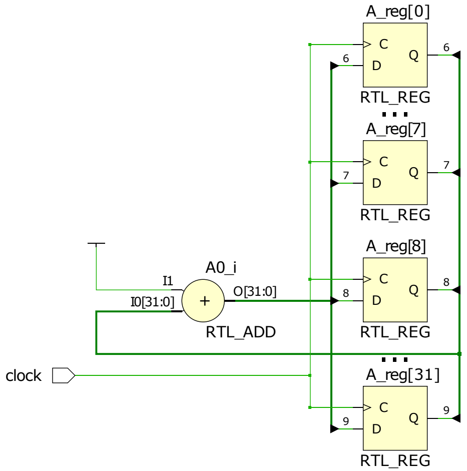


Figure 9: RTL Schematic of Example Counter

Assignment in Always block

- Two types of assignments can be done in the always block.
- In the last example we saw parallel(<=) assignment. Despite it's name, this assignment **overrides** all of other previous assignments to that variable in that always block.
- The other type of assignment is sequential(=) assignment. With this one we can cascade multiple operations in one always block.
- **ONLY ONE TYPE OF ASSIGNMENT CAN BE USED IN THE SAME ALWAYS BLOCK.**

Parallel vs Sequential Assignment

```
...
        reg [31:0] A;
        reg [31:0] B;
...
always @(posedge clock)
begin
    A <= A + 32'd1; // This line will be
    A <= A << 2;    // Overriden by this line.
end                 // A = 4A
...
...
always @(posedge clock)
begin
    B = B + 32'd1;
    B = B << 2;
end                 // B = 4(B + 1)
...
```

Parallel vs Sequential Assignment

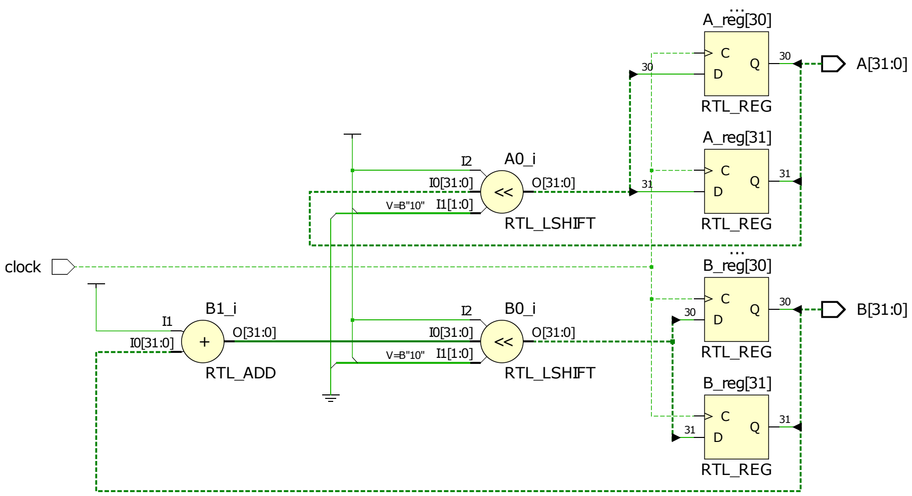


Figure 10: RTL Schematic of Parallel and Sequential Assignment

Counter With Reset Using If-Else Block

```
...
        reg [31:0] A;
...
always @(posedge clock or negedge reset)
begin
    if(!reset)          // Reset is active low.
    begin
        A <= 32'd0;
    end
    else
    begin
        A <= A + 32'd1;
    end
end
...
```

Counter With Reset Using If-Else Block

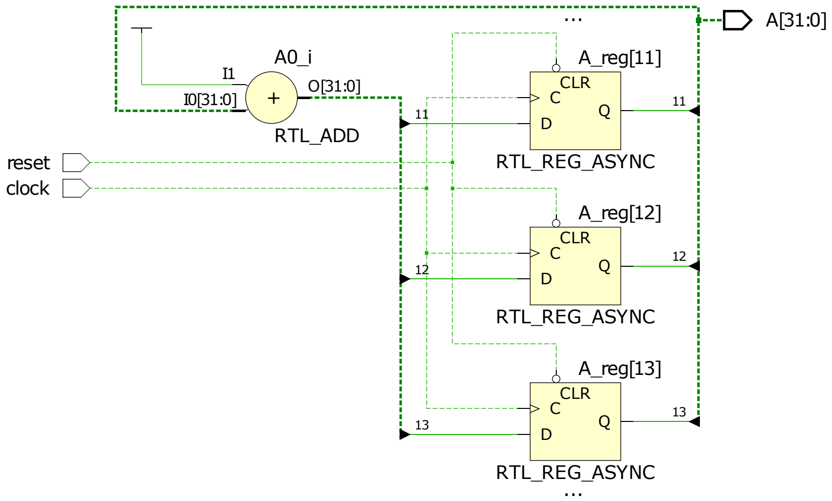


Figure 11: RTL Schematic of Counter With Reset Using If-Else Block

Sample FSM using Switch-Case

```
    case(state)
        2'b00:     begin
                       A <= 32'd1234;
                       state <=  2'b11;
                   end
        2'b01 :    begin
                       A <=  32'd2222;
                       state <=  2'b00;
                   end
        2'b11 :    begin
                       A <=  32'd1773;
                       state <=  2'b01;
                   end
        default  : begin
                       A <=  32'd0000;
                       state <=  2'b00;
                   end         // State can be
    endcase                    // 2'bXX or 2'bZZ
```

Sample FSM using Switch-Case

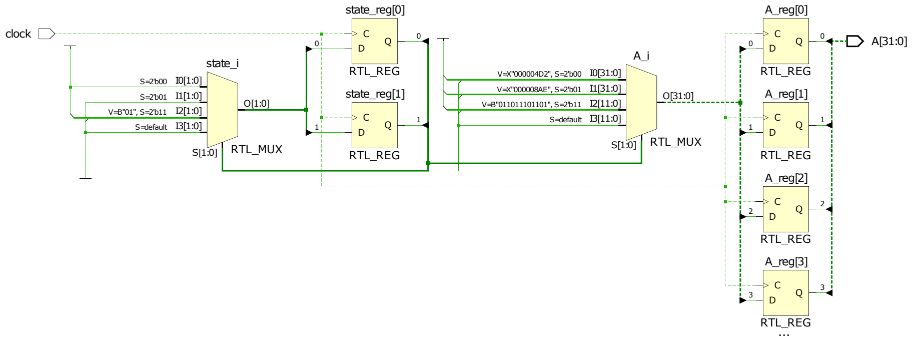


Figure 12: RTL Schematic of Sample FSM using Switch-Case

Combinational Always Blocks

```
    ...
            reg [3:0] A;
            reg [1:0] B;
    // Simple 2-to-4 Decoder.
    // * means when any variable in always block
    // Changes its value apply always block
    always @(*)
    begin
        case(B)
            2'b00:    A <= 4'b0001;
            2'b01:    A <= 4'b0010;
            2'b10:    A <= 4'b0100;
            2'b11:    A <= 4'b1000;

            default: A <= 4'b0000;
        endcase
    end
    ...
```

Combinational Always Blocks

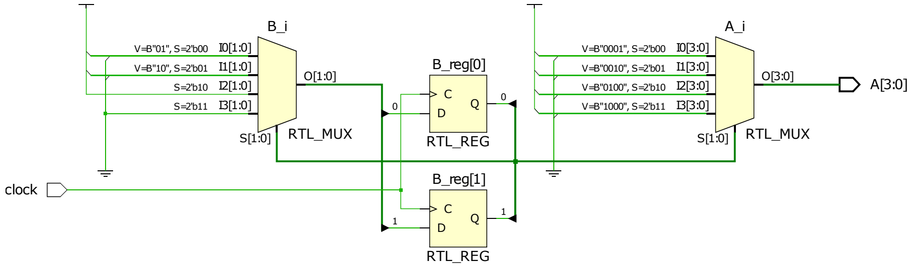


Figure 13: RTL Schematic of Combinational Always Blocks

General Rules of Thumb

- Never assign to same variable in different always blocks. Race condition between them almost always create problems.
- Try to seperate your FSM designs into 2 parts. In one of only update the state values using **always@(posedge clock)** and in the other calculate rest of the things. Doing this seperation correctly will cause less problems when coding.
- You are coding hardware, never forget that. So do not try to do many things in one clock cycle.
- Never use initial blocks in your design files. It is impossible to physically set a bit when module powers up. Set your registers using reset functionality instead.

Example Properly Coded FSM

```
module coded_fsm(clock, reset, o);
    input   wire              clock;
    input   wire              reset;
            reg      [1:0]    state_reg_q;
            reg      [1:0]    state_reg_d;
    output  reg      [7:0]    o;
    always @(posedge clock or negedge reset) begin
        if(!reset)
        begin
            state_reg_q <= 2'd0;
            o <= 8'd0;
        end
        else
        begin
            state_reg_q <= state_reg_d;
            o <= o + state_reg_q;
        end
    end
```

Example Properly Coded FSM

```
    always @(*)
    begin
        case(state_reg_q)
            2'b00:   state_reg_d <= 2'b10;
            2'b01:   state_reg_d <= 2'b00;
            2'b10:   state_reg_d <= 2'b01;

            default: state_reg_d <= 2'b00;
        endcase
    end
endmodule
```

Example Properly Coded FSM

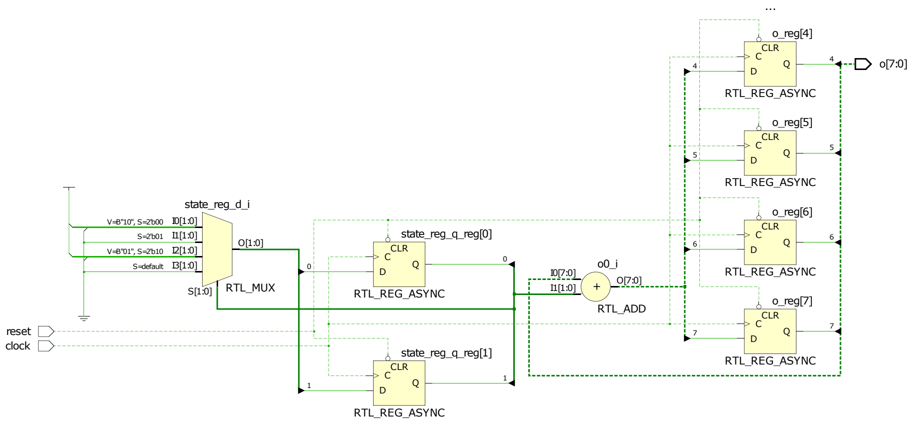


Figure 14: RTL Schematic of Example Properly Coded FSM

Creating Clock in the Testbench

```
module coded_fsm_tb();
        reg             clock;
        reg             reset;
        wire [7:0]      out;

    coded_fsm module_test(clock, reset, out);
    initial begin
        clock=0; reset=1; #100; //Initiate signals
        reset=0; #30;
        reset=1; #200;
        reset=0; #20;
        reset=1; #150;
        $finish; //Teminate simulation
    end
    always begin
        clock = ~clock; #25; //Toggle clock signal
    end
endmodule
```

Simulation for Coded FSM

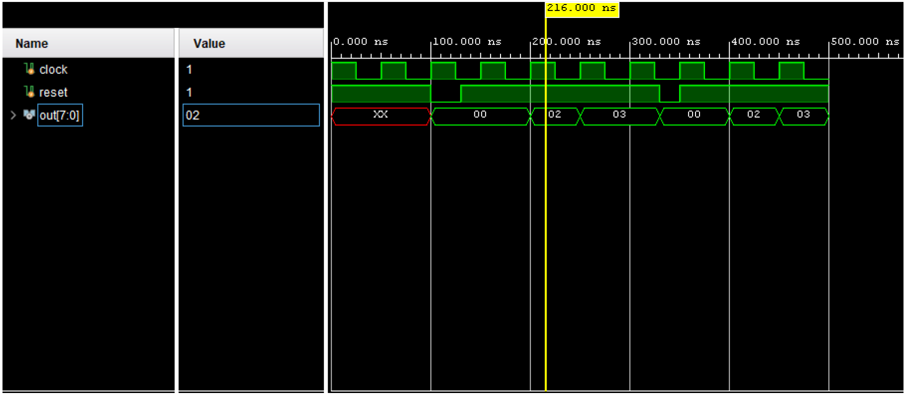


Figure 15: Simulation Timeline for Coded FSM

Example ALU Using Combinational Always Block

```
always @(*)
begin // Example 8-bit ALU
    case(operation)
        // Addition
        2'b00:   o <= A + B;
        // Subtraction
        2'b01:   o <= A - B;
        // Circular left shift
        2'b10:   o <= {A[6:0],A[7]};
        // Circular right shift
        2'b11:   o <= {A[0],A[7:1]};

        default: o <= 8'd0;
    endcase
end
```

Example ALU Using Combinational Always Block

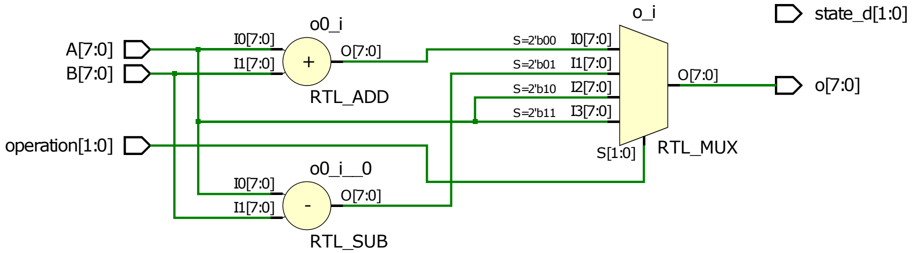


Figure 16: RTL Schematic of ALU Module

Parametric Modules

- We will generate n-bit ripple carry adder/subtractor.
- First, we need to code a full adder.

```
module FA (a, b, cin, s, cout);
    input   wire        a;
    input   wire        b;
    input   wire        cin;
    output  wire        s;
    output  wire        cout;
            wire        a_xor_b;

    assign a_xor_b = a ^ b;
    assign s = a_xor_b ^ cin;
    assign cout = (a_xor_b & cin) | (a & b);
endmodule
```

FSM Schematic

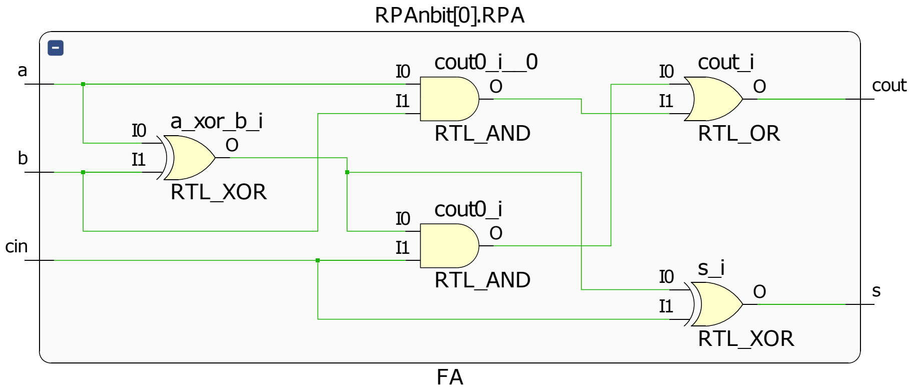


Figure 17: FSM schematic that is generated.

Parametric Modules

- After coding full adder. We will use these full adders as basic building blocks for the ripple carry adder.
- n-bit ripple carry adder needs n full adders. To parametize this using Verilog, we will use **generate** code block.
- Basically inside generate block, we will create a iterator (called **genvar**) and use this iterator in a for loop to create our parametized ripple carry adder.

n-bit Ripple Carry Adder

```
module RPA_nbit #(parameter n = 4)
                 (a, b, select, s, cout);
    input   wire    [n-1:0]     a;
    input   wire    [n-1:0]     b;
    // Addition or Substraction ?
    input   wire                select;

    output  wire    [n-1:0]     s;
    output  wire                cout;

    // Will connect carry ports of FAs
            wire    [n  :0]     miniCout;
            wire    [n-1:0]     c;
    // If select=1 in then subtraction happens.
```

n-bit Ripple Carry Adder

```verilog
    // If select=1 in then subtraction happens.
    assign c = (select) ? ~b : b;
    assign miniCout[0] = select;
    assign cout = miniCout[n];

    generate
        genvar x;
        for(x = 0; x < n; x = x + 1) begin: RPAnbit
        FA RPA( .a(a[x]),
                .b(c[x]),
                .s(s[x]),
                .cin(miniCout[x]),
                .cout(miniCout[x+1]));
        end
    endgenerate
endmodule
```

n-bit Ripple Carry Adder Schematic

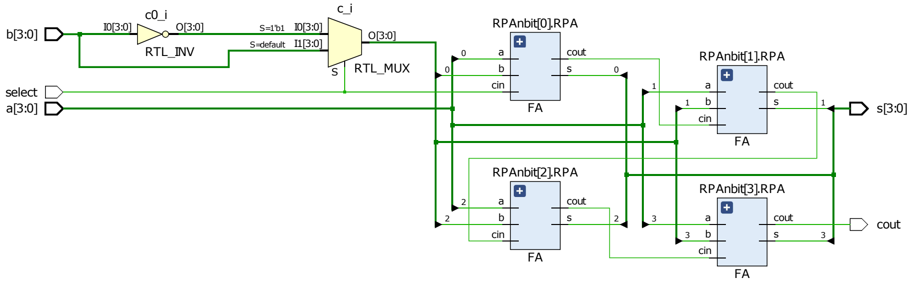


Figure 18: 4-bit ripple carry adder.

n-bit Ripple Carry Adder Testbench

```verilog
module Test_RPA_nbit;
    parameter N = 8; // Since we parametized RPA
                     // Its width can be changed.
    // Inputs
    reg [N-1:0] a;
    reg [N-1:0] b;
    reg select;
    // Outputs
    wire [N-1:0] s;
    wire cout;
    // #(.n(N)) overrides the n parameter of
    // module to 8 making it 8 bit adder.
    RPA_nbit  #(.n(N)) uut(.a(a),
                           .b(b),
                           .select(select),
                           .s(s),
                           .cout(cout));
```

n-bit Ripple Carry Adder Testbench

```
    initial begin
        // Initialize Inputs
        a = 8'h11; b = 8'h11; select = 0; #100;
        a = 8'h31; b = 8'hff; select = 0; #100;
        a = 8'h53; b = 8'hac; select = 0; #100;
        a = 8'h45; b = 8'hbc; select = 0; #100;
        a = 8'h84; b = 8'h22; select = 0; #100;

        a = 8'h01; b = 8'h0a; select = 1; #100;
        a = 8'h0a; b = 8'h0b; select = 1; #100;
        a = 8'h0c; b = 8'h0f; select = 1; #100;
        a = 8'hff; b = 8'h0f; select = 1; #100;
        a = 8'h0f; b = 8'h0a; select = 1; #100;
    end
endmodule
```

Simulation Results of RPA

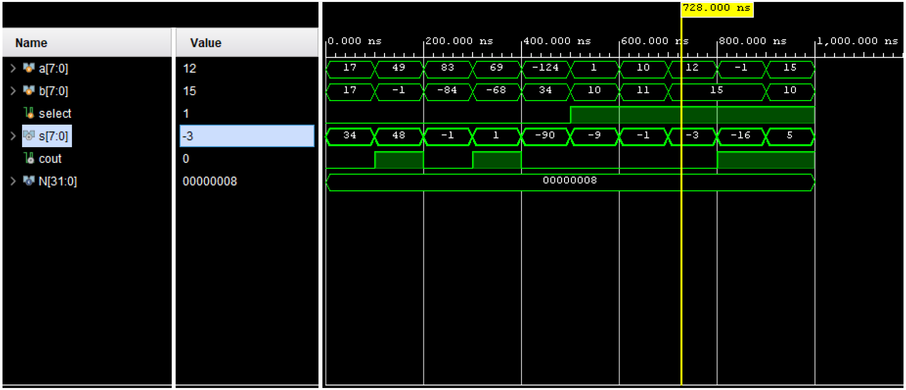
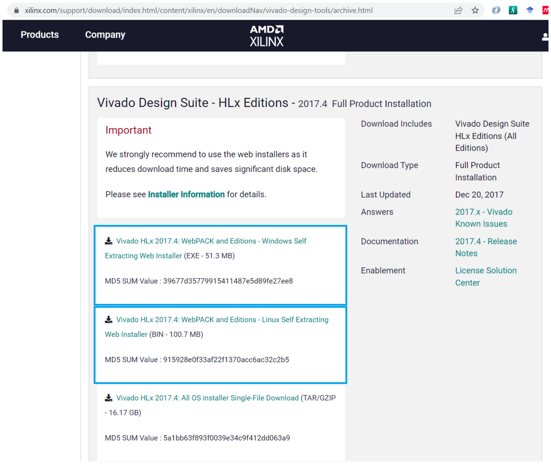

Name	Value
a[7:0]	12
b[7:0]	15
select	1
s[7:0]	-3
cout	0
N[31:0]	00000008

Figure 19: The module is working as intended.

Vivado Design Suite

- Xilinx is the mainstream FPGA company. Its products are widely used in the industry.
- To program their FPGAs Xilinx first developed ISE Design Platform.
- After version 14.7, Xilinx have created another design platform called Vivado. Which is more versatile and more inline with the industry standards.
- Both of these design platforms have built-in simulators.
- After synthesizing(mapping them to the cells) the design's RC values can be extracted and simulations with propagation delays can be done.

Install Xilinx Vivado (WebPack version)


xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/archive.html

Vivado Design Suite - HLx Editions - 2017.4 Full Product Installation

Important

We strongly recommend to use the web installers as it reduces download time and saves significant disk space.

Please see **Installer Information** for details.

Vivado HLx 2017.4: WebPACK and Editions - Windows Self Extracting Web Installer (EXE - 51.3 MB)

MD5 SUM Value : 39677d35779915411487e5d89fe27ee8

Vivado HLx 2017.4: WebPACK and Editions - Linux Self Extracting Web Installer (BIN - 100.7 MB)

MD5 SUM Value : 915928e0f33af22f1370acc6ac32c2b5

Vivado HLx 2017.4: All OS installer Single-File Download (TAR/GZIP - 16.17 GB)

MD5 SUM Value : 5a1bb63f893f0039e34c9f412dd063a9

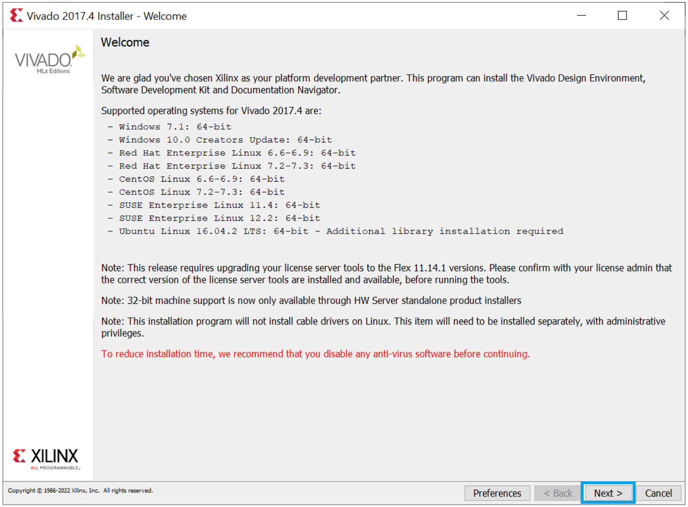
Download Includes	Vivado Design Suite HLx Editions (All Editions)
Download Type	Full Product Installation
Last Updated	Dec 20, 2017
Answers	2017.x - Vivado Known Issues
Documentation	2017.4 - Release Notes
Enablement	License Solution Center

Figure 20: Download proper *Xilinx Unified Installer 2017.4* executable file with your operating system from `https://www.xilinx.com/support/download.html`

Installation of Xilinx Vivado on Linux

- To install Xilinx Vivado on linux, you can watch the video on url: `https://www.youtube.com/watch?v=cQHe651AEaM`.

Install Xilinx Vivado (WebPack version)



Welcome

We are glad you've chosen Xilinx as your platform development partner. This program can install the Vivado Design Environment, Software Development Kit and Documentation Navigator.

Supported operating systems for Vivado 2017.4 are:

```
 - Windows 7.1: 64-bit
 - Windows 10.0 Creators Update: 64-bit
 - Red Hat Enterprise Linux 6.6-6.9: 64-bit
 - Red Hat Enterprise Linux 7.2-7.3: 64-bit
 - CentOS Linux 6.6-6.9: 64-bit
 - CentOS Linux 7.2-7.3: 64-bit
 - SUSE Enterprise Linux 11.4: 64-bit
 - SUSE Enterprise Linux 12.2: 64-bit
 - Ubuntu Linux 16.04.2 LTS: 64-bit - Additional library installation required
```

Note: This release requires upgrading your license server tools to the Flex 11.14.1 versions. Please confirm with your license admin that the correct version of the license server tools are installed and available, before running the tools.

Note: 32-bit machine support is now only available through HW Server standalone product installers

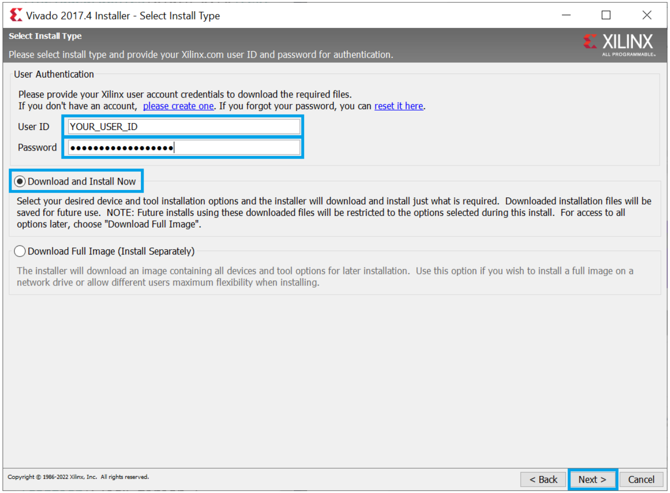
Note: This installation program will not install cable drivers on Linux. This item will need to be installed separately, with administrative privileges.

To reduce installation time, we recommend that you disable any anti-virus software before continuing.

Copyright © 1986-2022 Xilinx, Inc. All rights reserved.

Figure 21: Welcome page.After executable file is double clicked, this page appears.

Install Xilinx Vivado (WebPack version)


Vivado 2017.4 Installer - Select Install Type

Select Install Type

Please select install type and provide your Xilinx.com user ID and password for authentication.

User Authentication

Please provide your Xilinx user account credentials to download the required files.
If you don't have an account, please create one. If you forgot your password, you can reset it here.

User ID YOUR_USER_ID

Password ••••••••••••••••••

[✓ Download and Install Now]

Select your desired device and tool installation options and the installer will download and install just what is required. Downloaded installation files will be saved for future use. NOTE: Future installs using these downloaded files will be restricted to the options selected during this install. For access to all options later, choose "Download Full Image".

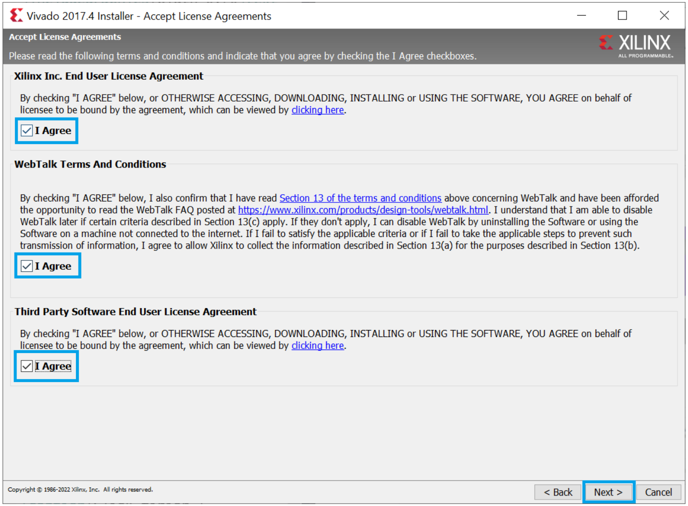
Download Full Image (Install Separately)

The installer will download an image containing all devices and tool options for later installation. Use this option if you wish to install a full image on a network drive or allow different users maximum flexibility when installing.

Copyright © 1986-2022 Xilinx, Inc. All rights reserved.

Figure 22: You should enter your Xilinx account information here. If you don't have any, you must sign up firstly.

Install Xilinx Vivado (WebPack version)



Vivado 2017.4 Installer - Accept License Agreements

Accept License Agreements

Please read the following terms and conditions and indicate that you agree by checking the I Agree checkboxes.

Xilinx Inc. End User License Agreement

By checking "I AGREE" below, or OTHERWISE ACCESSING, DOWNLOADING, INSTALLING or USING THE SOFTWARE, YOU AGREE on behalf of licensee to be bound by the agreement, which can be viewed by clicking here.

[✓ I Agree]

WebTalk Terms And Conditions

By checking "I AGREE" below, I also confirm that I have read Section 13 of the terms and conditions above concerning WebTalk and have been afforded the opportunity to read the WebTalk FAQ posted at https://www.xilinx.com/products/design-tools/webtalk.html. I understand that I am able to disable WebTalk later if certain criteria described in Section 13(c) apply. If they don't apply, I can disable WebTalk by uninstalling the Software or using the Software on a machine not connected to the internet. If I fail to satisfy the applicable criteria or if I fail to take the applicable steps to prevent such transmission of information, I agree to allow Xilinx to collect the information described in Section 13(a) for the purposes described in Section 13(b).

[✓ I Agree]

Third Party Software End User License Agreement

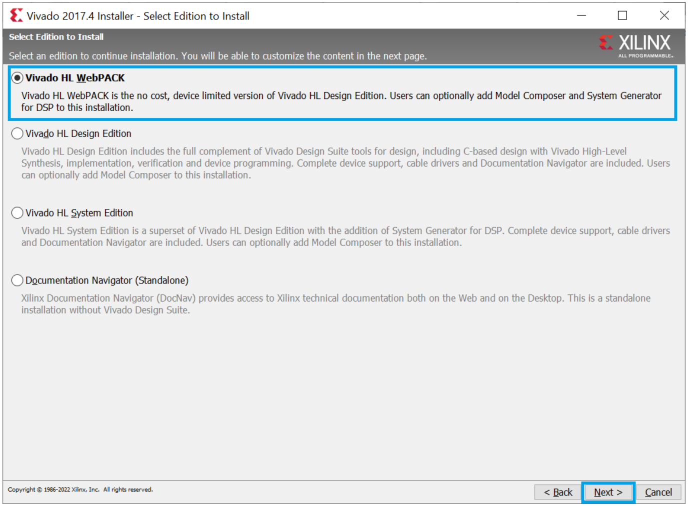
By checking "I AGREE" below, or OTHERWISE ACCESSING, DOWNLOADING, INSTALLING or USING THE SOFTWARE, YOU AGREE on behalf of licensee to be bound by the agreement, which can be viewed by clicking here.

[✓ I Agree]

Copyright © 1986-2022 Xilinx, Inc. All rights reserved.

Figure 23: After all agreement check boxes are clicked, please click "Next".

Install Xilinx Vivado (WebPack version)


Vivado 2017.4 Installer - Select Edition to Install

Select Edition to Install

Select an edition to continue installation. You will be able to customize the content in the next page.

[✓ **Vivado HL WebPACK**]

Vivado HL WebPACK is the no cost, device limited version of Vivado HL Design Edition. Users can optionally add Model Composer and System Generator for DSP to this installation.

Vivado HL Design Edition

Vivado HL Design Edition includes the full complement of Vivado Design Suite tools for design, including C-based design with Vivado High-Level Synthesis, implementation, verification and device programming. Complete device support, cable drivers and Documentation Navigator are included. Users can optionally add Model Composer to this installation.

Vivado HL System Edition

Vivado HL System Edition is a superset of Vivado HL Design Edition with the addition of System Generator for DSP. Complete device support, cable drivers and Documentation Navigator are included. Users can optionally add Model Composer to this installation.

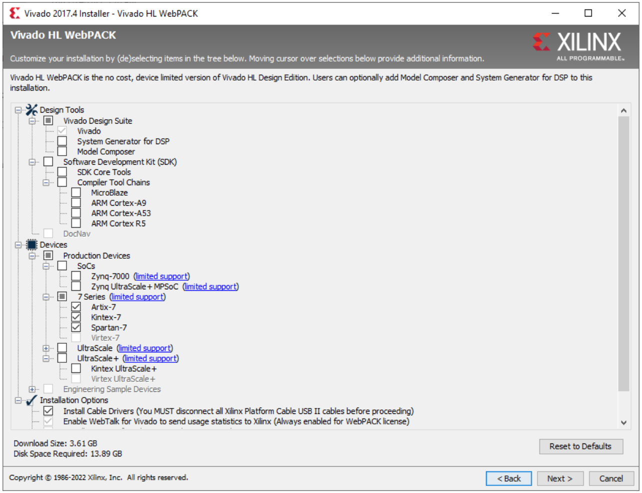
Documentation Navigator (Standalone)

Xilinx Documentation Navigator (DocNav) provides access to Xilinx technical documentation both on the Web and on the Desktop. This is a standalone installation without Vivado Design Suite.

Copyright © 1986-2022 Xilinx, Inc. All rights reserved.

Figure 24: At this step, you must select "Vivado HL WebPACK" option.

Install Xilinx Vivado (WebPack version)



Vivado 2017.4 Installer - Vivado HL WebPACK

Vivado HL WebPACK

Customize your installation by (de)selecting items in the tree below. Moving cursor over selections below provide additional information.

Vivado HL WebPACK is the no cost, device limited version of Vivado HL Design Edition. Users can optionally add Model Composer and System Generator for DSP to this installation.

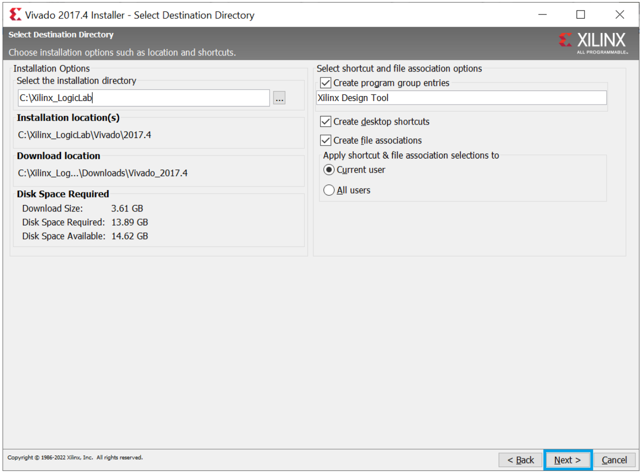
- Design Tools
  - ■ Vivado Design Suite
    - ✓ Vivado
    - ☐ System Generator for DSP
    - ☐ Model Composer
  - ☐ Software Development Kit (SDK)
    - ☐ SDK Core Tools
    - ☐ Compiler Tool Chains
      - ☐ MicroBlaze
      - ☐ ARM Cortex-A9
      - ☐ ARM Cortex-A53
      - ☐ ARM Cortex R5
  - ☐ DocNav
- Devices
  - ■ Production Devices
    - ☐ SoCs
      - ☐ Zynq-7000 (limited support)
      - ☐ Zynq UltraScale+ MPSoC (limited support)
    - ■ 7 Series (limited support)
      - ✓ Artix-7
      - ✓ Kintex-7
      - ✓ Spartan-7
      - ☐ Virtex-7
    - ☐ UltraScale (limited support)
    - ☐ UltraScale+ (limited support)
      - ☐ Kintex UltraScale+
      - ☐ Virtex UltraScale+
  - ☐ Engineering Sample Devices
- Installation Options
  - ✓ Install Cable Drivers (You MUST disconnect all Xilinx Platform Cable USB II cables before proceeding)
  - ✓ Enable WebTalk for Vivado to send usage statistics to Xilinx (Always enabled for WebPACK license)

Download Size: 3.61 GB
Disk Space Required: 13.89 GB

Copyright © 1986-2022 Xilinx, Inc. All rights reserved.

Figure 25: By selecting minimum configuration, you can decrease download and disc space requirements. Then, you can directly go to the next page.

Install Xilinx Vivado (WebPack version)


Vivado 2017.4 Installer - Select Destination Directory

Select Destination Directory

Choose installation options such as location and shortcuts.

Installation Options

Select the installation directory: C:\Xilinx_LogicLab

Installation location(s)

C:\Xilinx_LogicLab\Vivado\2017.4

Download location

C:\Xilinx_Log...\Downloads\Vivado_2017.4

Disk Space Required

Download Size:	3.61 GB
Disk Space Required:	13.89 GB
Disk Space Available:	14.62 GB

Select shortcut and file association options

[✓] Create program group entries: Xilinx Design Tool

[✓] Create desktop shortcuts

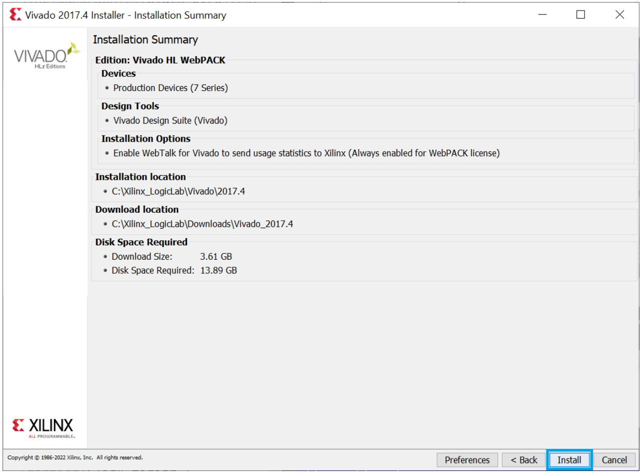
[✓] Create file associations

Apply shortcut & file association selections to: [✓ Current user] All users

Copyright © 1986-2022 Xilinx, Inc. All rights reserved.

Figure 26: Be careful about "Disk Space Required" information. According to these information, you can easily change your installation directory in "Installation Options" section.

Install Xilinx Vivado (WebPack version)



Vivado 2017.4 Installer - Installation Summary

Installation Summary

Edition: Vivado HL WebPACK

Devices
- Production Devices (7 Series)

Design Tools
- Vivado Design Suite (Vivado)

Installation Options
- Enable WebTalk for Vivado to send usage statistics to Xilinx (Always enabled for WebPACK license)

Installation location
- C:\Xilinx_LogicLab\Vivado\2017.4

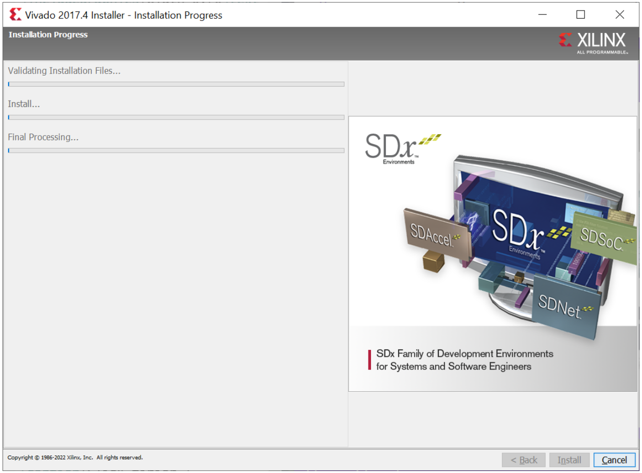
Download location
- C:\Xilinx_LogicLab\Downloads\Vivado_2017.4

Disk Space Required
- Download Size: 3.61 GB
- Disk Space Required: 13.89 GB

Copyright © 1986-2022 Xilinx, Inc. All rights reserved.

Figure 27: Your final configurations are listed here. Then, click "Install" button to start installation.

Install Xilinx Vivado (WebPack version)


Vivado 2017.4 Installer - Installation Progress

Installation Progress

Validating Installation Files...

Install...

Final Processing...

SDx Family of Development Environments
for Systems and Software Engineers

Copyright © 1986-2022 Xilinx, Inc. All rights reserved.

Figure 28: Installation just started.

Using Vivado

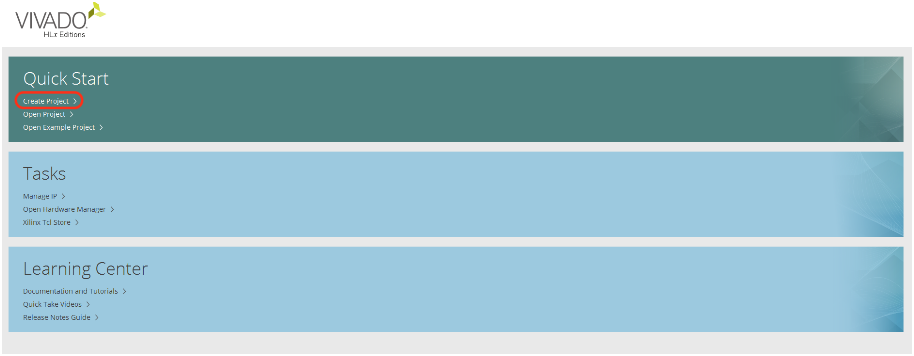
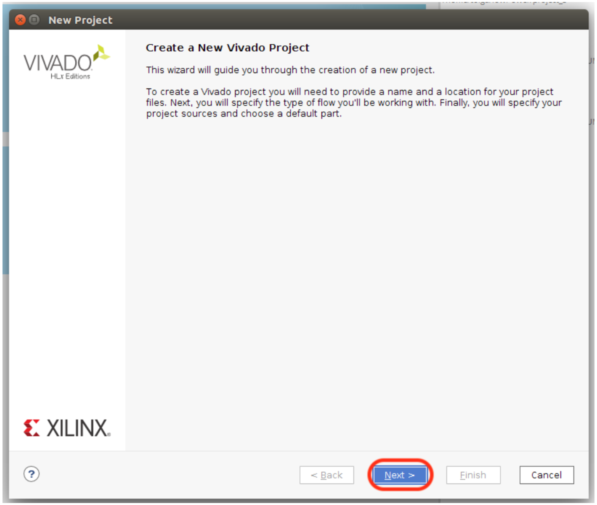


Figure 29: Open Vivado, and create a new project.

Using Vivado


New Project

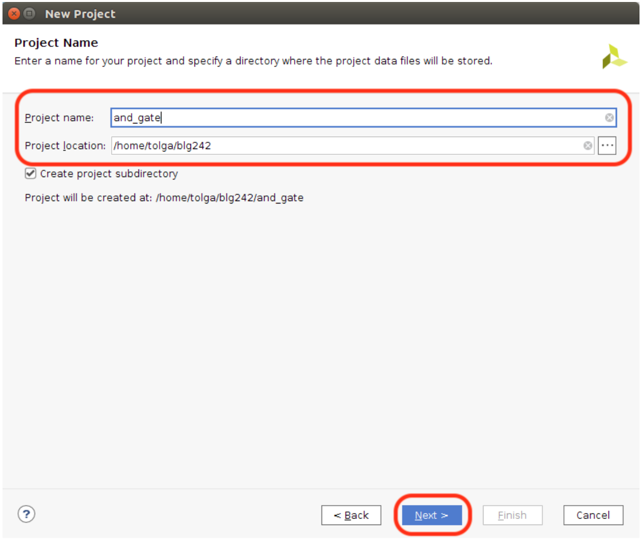
Create a New Vivado Project

This wizard will guide you through the creation of a new project.

To create a Vivado project you will need to provide a name and a location for your project files. Next, you will specify the type of flow you'll be working with. Finally, you will specify your project sources and choose a default part.

Figure 30: Click next.

Using Vivado



New Project

Project Name

Enter a name for your project and specify a directory where the project data files will be stored.

Project name: and_gate

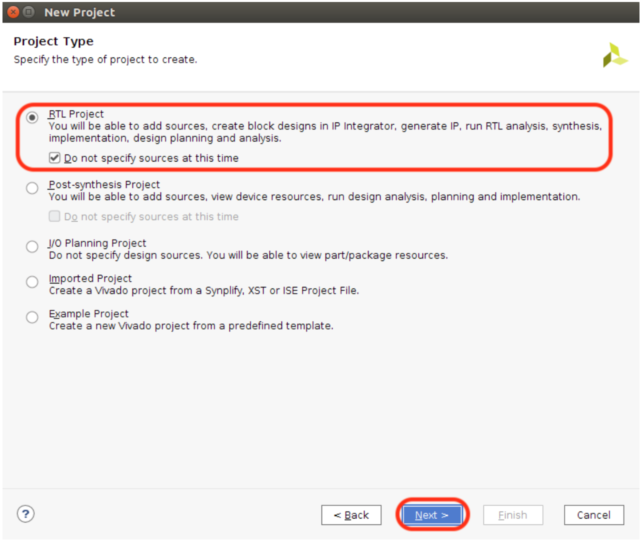
Project location: /home/tolga/blg242

[✓] Create project subdirectory

Project will be created at: /home/tolga/blg242/and_gate

Figure 31: Enter project name.

Using Vivado



Project Type
Specify the type of project to create.

(●) RTL Project
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.
[✓] Do not specify sources at this time

( ) Post-synthesis Project
You will be able to add sources, view device resources, run design analysis, planning and implementation.
[ ] Do not specify sources at this time

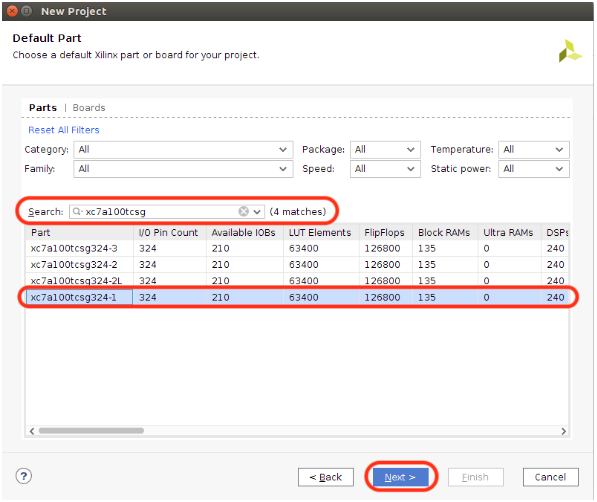
( ) I/O Planning Project
Do not specify design sources. You will be able to view part/package resources.

( ) Imported Project
Create a Vivado project from a Synplify, XST or ISE Project File.

( ) Example Project
Create a new Vivado project from a predefined template.

Figure 32: The project will be an RTL project.

Using Vivado


New Project

Default Part

Choose a default Xilinx part or board for your project.

Parts | Boards

Reset All Filters

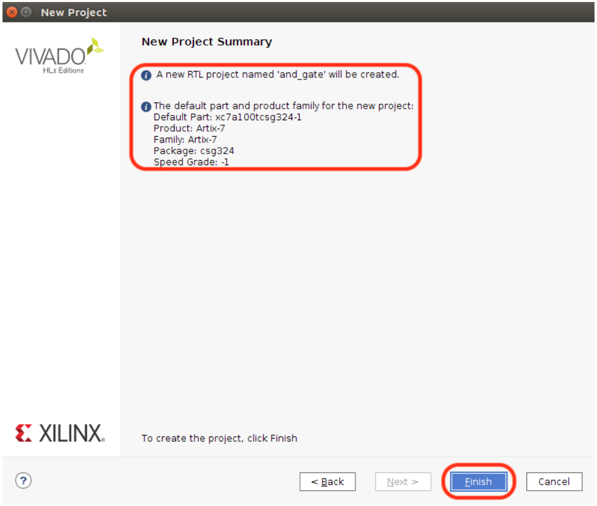
Category:	All	Package:	All	Temperature:	All
Family:	All	Speed:	All	Static power:	All

Search: xc7a100tcsg (4 matches)

Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	Ultra RAMs	DSPs
xc7a100tcsg324-3	324	210	63400	126800	135	0	240
xc7a100tcsg324-2	324	210	63400	126800	135	0	240
xc7a100tcsg324-2L	324	210	63400	126800	135	0	240
xc7a100tcsg324-1	324	210	63400	126800	135	0	240

Figure 33: Select part as xc7a100tcsg324-1

Using Vivado



New Project Summary

A new RTL project named 'and_gate' will be created.

The default part and product family for the new project:
Default Part: xc7a100tcsg324-1
Product: Artix-7
Family: Artix-7
Package: csg324
Speed Grade: -1

To create the project, click Finish

Figure 34: Click finish.

Using Vivado

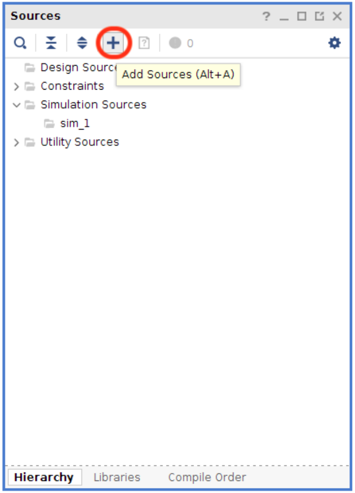
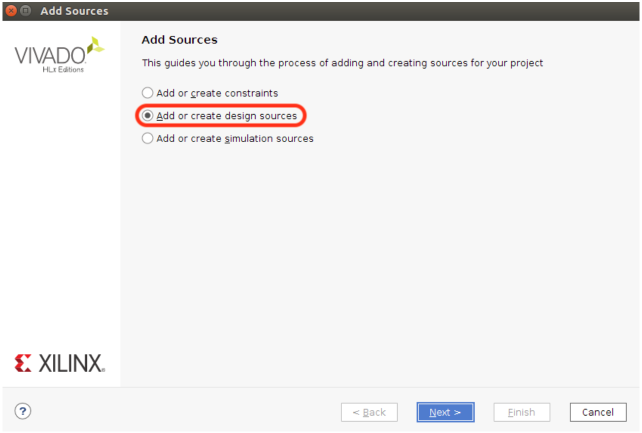


Figure 35: Add new sources.

Using Vivado



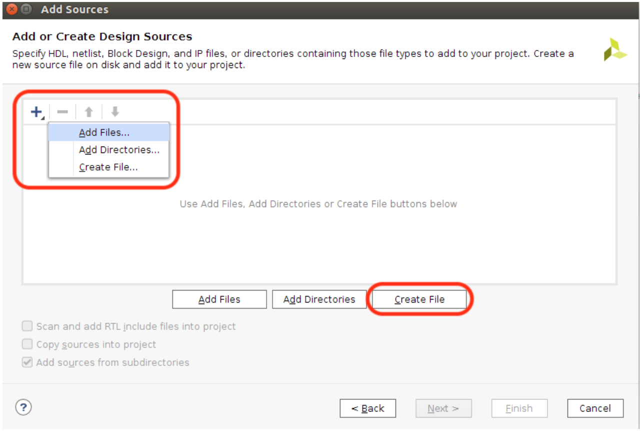
Add Sources

This guides you through the process of adding and creating sources for your project

Add or create constraints
[✓ Add or create design sources]
Add or create simulation sources

Figure 36: At first we will add design sources.

Using Vivado


Add Sources

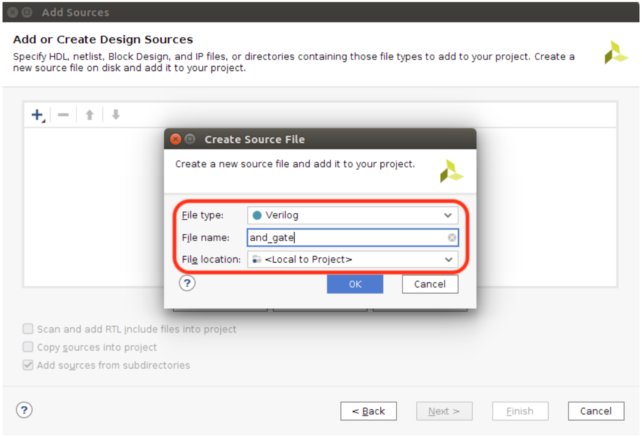
Add or Create Design Sources

Specify HDL, netlist, Block Design, and IP files, or directories containing those file types to add to your project. Create a new source file on disk and add it to your project.

Scan and add RTL include files into project
Copy sources into project
Add sources from subdirectories

Figure 37: You can add existing files or create a brand new file.

Using Vivado



Add Sources

Add or Create Design Sources

Specify HDL, netlist, Block Design, and IP files, or directories containing those file types to add to your project. Create a new source file on disk and add it to your project.

Create Source File

Create a new source file and add it to your project.

File type: Verilog

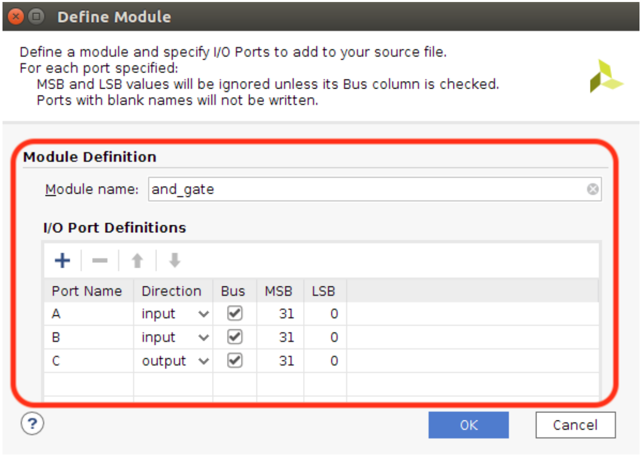
File name: and_gate

File location: <Local to Project>

OK
Cancel
Scan and add RTL include files into project
Copy sources into project
Add sources from subdirectories
< Back
Next >
Finish
Cancel

Figure 38: File type will be Verilog.

Using Vivado



Define a module and specify I/O Ports to add to your source file.
For each port specified:
MSB and LSB values will be ignored unless its Bus column is checked.
Ports with blank names will not be written.

Module Definition

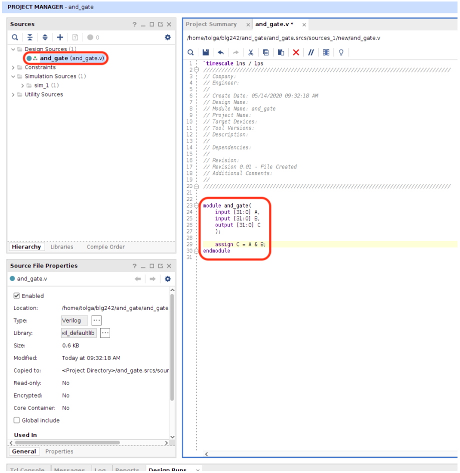
Module name: and_gate

I/O Port Definitions

Port Name	Direction	Bus	MSB	LSB
A	input	✓	31	0
B	input	✓	31	0
C	output	✓	31	0

Figure 39: When creating a file Vivado gives us option to specify inputs and outputs before creation.

Using Vivado



Enabled

Location: /home/tolga/blg242/and_gate/and_gate

Type: Verilog

Library: xil_defaultlib

Size: 0.6 KB

Modified: Today at 09:32:18 AM

Copied to: <Project Directory>/and_gate.srcs/sour

Read-only: No

Encrypted: No

Core Container: No

Global include

Used In

/home/tolga/blg242/and_gate/and_gate.srcs/sources_1/new/and_gate.v

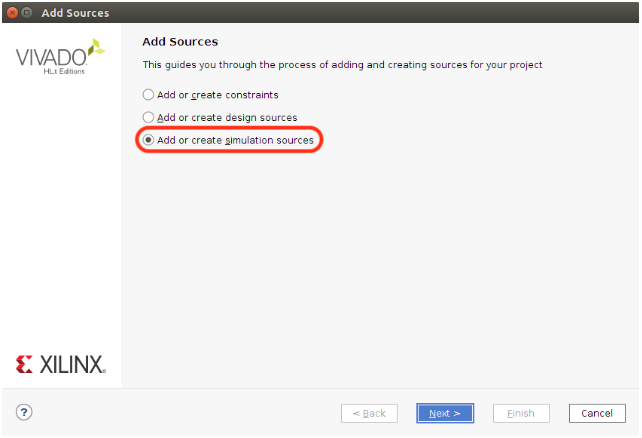
```verilog
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 05/14/2020 09:32:18 AM
// Design Name:
// Module Name: and_gate
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module and_gate(
    input [31:0] A,
    input [31:0] B,
    output [31:0] C
    );

    assign C = A & B;
endmodule
```

Figure 40: Code your module.

Using Vivado

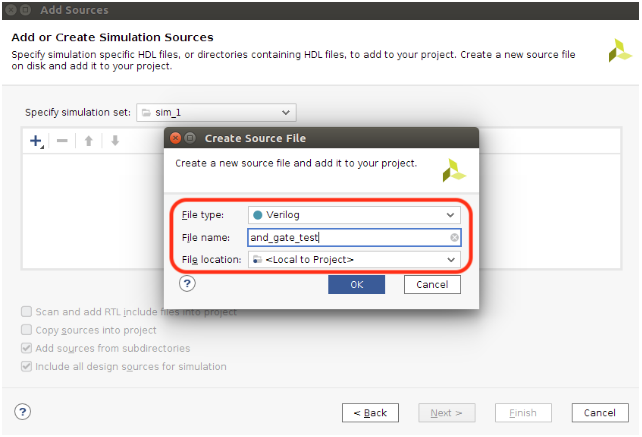

Add Sources

This guides you through the process of adding and creating sources for your project

- Add or create constraints
- Add or create design sources
- [✓ Add or create simulation sources]

Figure 41: Now, we are ready to simulate. We need to create simulation source which will be the testbench.

Using Vivado


Add Sources

Add or Create Simulation Sources

Specify simulation specific HDL files, or directories containing HDL files, to add to your project. Create a new source file on disk and add it to your project.

Specify simulation set: sim_1

Create Source File

Create a new source file and add it to your project.

File type: Verilog

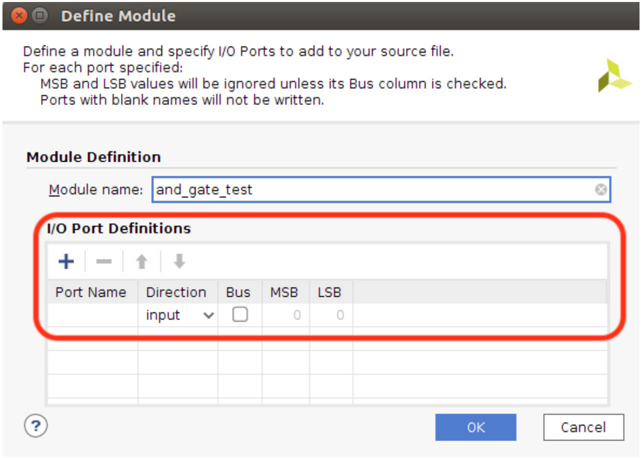
File name: and_gate_test

File location: <Local to Project>

OK
Cancel
Scan and add RTL include files into project
Copy sources into project
Add sources from subdirectories
Include all design sources for simulation
< Back
Next >
Finish
Cancel

Figure 42: Simulation file will be Verilog file.

Using Vivado



Define a module and specify I/O Ports to add to your source file.
For each port specified:
MSB and LSB values will be ignored unless its Bus column is checked.
Ports with blank names will not be written.

Module Definition

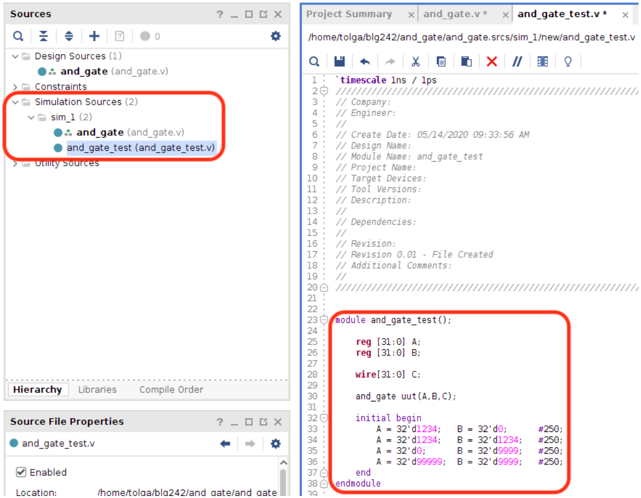
Module name: and_gate_test

I/O Port Definitions

Port Name	Direction	Bus	MSB	LSB
	input	☐	0	0

Figure 43: Remember testbench has no I/O.

Using Vivado



/home/tolga/blg242/and_gate/and_gate.srcs/sim_1/new/and_gate_test.v

```verilog
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 05/14/2020 09:33:56 AM
// Design Name:
// Module Name: and_gate_test
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module and_gate_test();

    reg [31:0] A;
    reg [31:0] B;

    wire[31:0] C;

    and_gate uut(A,B,C);

    initial begin
        A = 32'd1234;   B = 32'd0;      #250;
        A = 32'd1234;   B = 32'd1234;   #250;
        A = 32'd0;      B = 32'd9999;   #250;
        A = 32'd99999;  B = 32'd9999;   #250;
    end
endmodule
```

Figure 44: Code your testbench.

Using Vivado

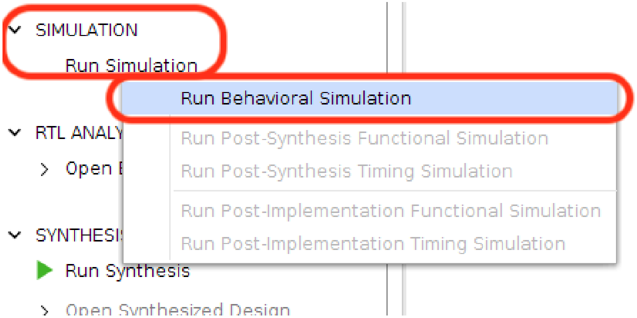
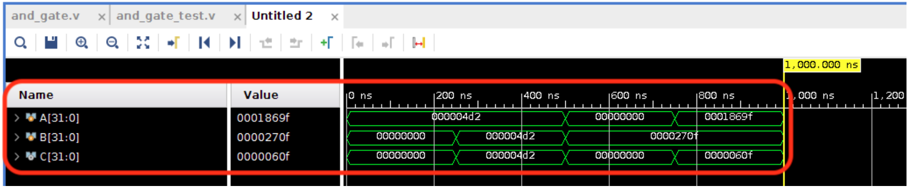

SIMULATION
Run Simulation
Run Behavioral Simulation
Run Post-Synthesis Functional Simulation
Run Post-Synthesis Timing Simulation
Run Post-Implementation Functional Simulation
Run Post-Implementation Timing Simulation
RTL ANALY
Open E
SYNTHESIS
Run Synthesis
Open Synthesized Design

Figure 45: We are now ready to simulate. Click Run simulation and select Behavioral Simulation.

Using Vivado



```
A[31:0]  0001869f   000004d2 | 00000000 | 0001869f
B[31:0]  0000270f   00000000 | 000004d2 | 0000270f
C[31:0]  0000060f   00000000 | 000004d2 | 00000000 | 0000060f
```

Figure 46: A waveform will be opened. You can see your applied inputs.

Using Vivado

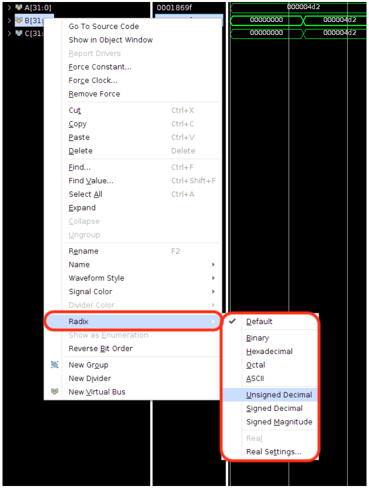
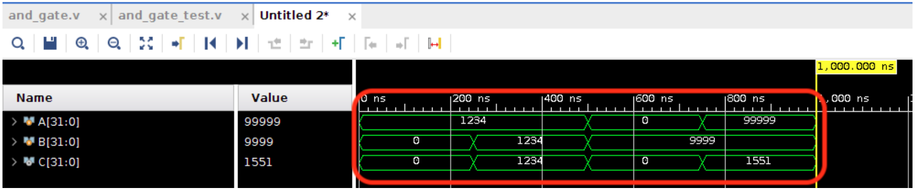


Figure 47: To make following waveform easier you can change the radix(base) of the numbers displayed in the waveform.

Using Vivado



```
A[31:0]  99999
B[31:0]  9999
C[31:0]  1551
```

Figure 48: Waveform numbers are now decimal.

Using Vivado

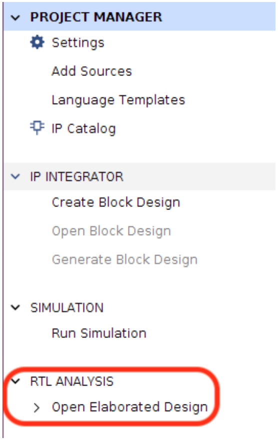


Figure 49: Open eleborated design to see system schematic.

Using Vivado

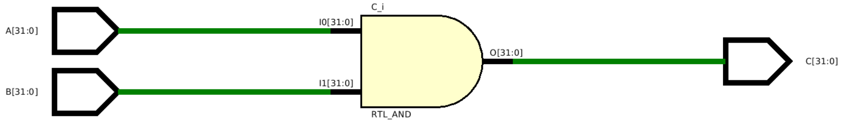


Figure 50: The system schematic.