Introduction to High Level Design Workshop

2019.05.31

Contents

1. Overview

1.1. Introduction

With the ever-growing number of software engineers in the field, High-level Synthesis (HLS) has become a tool many tech companies have decided to invest in. As we discussed earlier, HLS is an automated design process that interprets an algorithmic description of a desired behavior and creates digital hardware that implements that behavior. It works at a higher level of abstraction than traditional hardware descriptive languages (such as Verilog or VHDL) by starting with an algorithmic description in C/C++, and synthesizes it down to a register-transfer level (RTL) design. The goal of HLS is to allow hardware AND software designers to efficiently build and verify hardware in an algorithmic domain.

In this exercise, we will first observe the flow for designing a simple traffic light controller in Verilog using Intel® Quartus® Prime and simulating it using ModelSim® simulator. We will then observe the HLS flow by implementing the same traffic light algorithm in C++ and using the HLS compiler to synthesize the code down to RTL. The last section of this lab will compare the compilation reports generated by Quartus and HLS.

1.2. Setup

The following steps describe how to set up and connect to the virtual machine needed for this lab which uses ModelSim* simulator, the Intel Quartus Prime software, and HLS.
If you are connecting via Oracle VirtualBox, refer to section 1.2.1
If you are connecting via NoMachine (Intel users only), refer to section 1.2.2

1.2.1. Oracle VirtualBox Setup

1. Launch Oracle VM VirtualBox to open to VM VirtualBox Manager

   The latest version of VirtualBox can be installed at https://www.virtualbox.org/

2. Insert the USB Stick handed out into your PC
3. In the VM VirtualBox Manger, click **Machine -> Add New Device.** Map to the USB Drive plugged into your PC and open the file *FPGA_Train_CentOS6_18_0.vbox*
4. In the VM VirtualBox Manager, **right click** on the recently added *FPGA_Train_CentOS6_18_0* machine then click **Settings**
   a. Ensure *Ubuntu (64-bit)* version is selected under the 'General' tab
   b. In the 'System Tab', under 'Motherboard' ensure the Base Memory you are allocating is in the green area of the spectrum. See figure below for details.

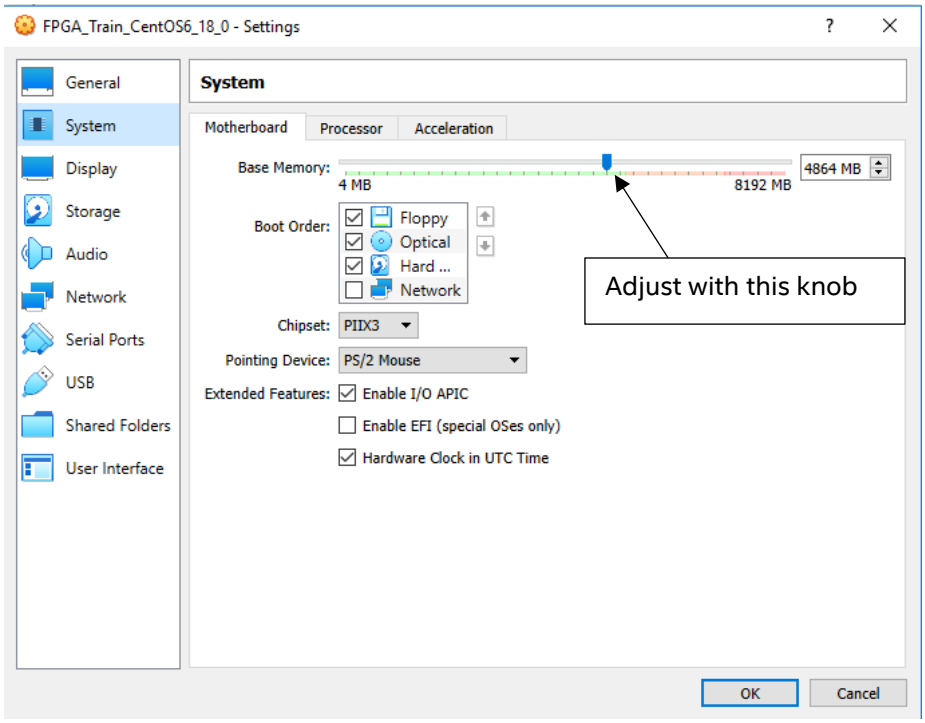


Figure 1: Adjust Base Memory in System Settings

c. In the Acceleration tab of 'System' ensure your settings match those below

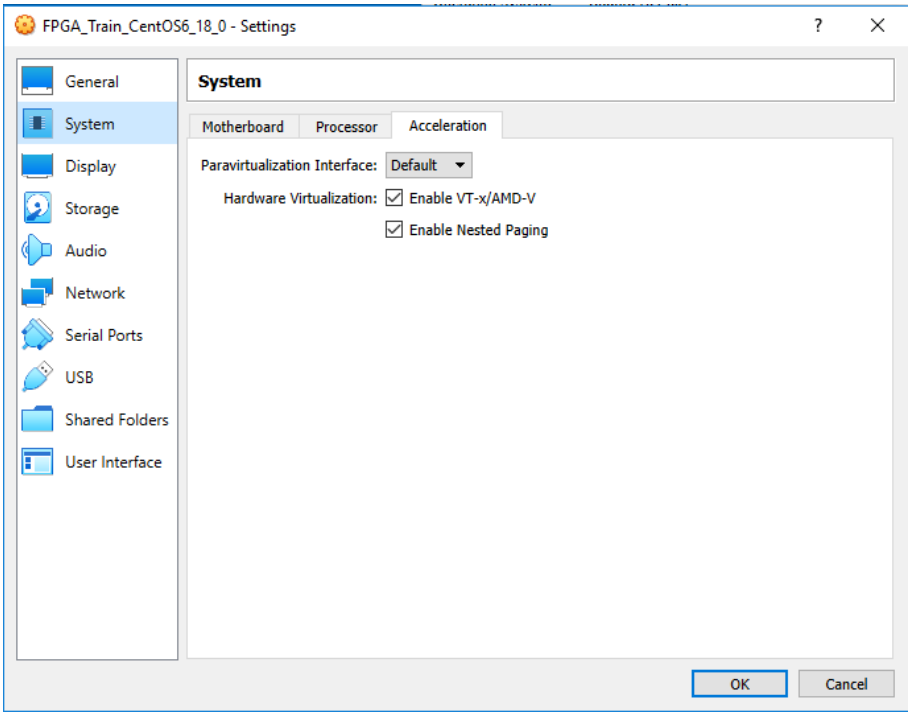


Figure 2: Acceleration Enablement in System Settings

5. Once all the appropriate settings are set, click **Start**
    a. When prompted enter the following password : *QPrime.1*

Once the Ubuntu virtual machine is successfully running, you will need to run a script to retrieve the Quartus Pro, ModelSim-Intel, and HLS resources. Perform the following steps:

1. Open a Terminal session by right clicking -> Open in Terminal
2. Type **cd**
3. Type **source fpga_trn/High_Level_Design_Workshop/init.sh**

You now have all the necessary resources and are ready to begin the exercises!

1.2.2. NoMachine Setup

1. Launch the NoMachine Enterprise Client and connect to a virtual desktop
2. Open up a terminal
    a. Right click → Open in Terminal
3. Copy the Workshop tar file to your current directory & unzip
    a. Type the following two commands in your Terminal:

    ```
    cp /home/mberglun/High_Level_Design_Workshop.tar.gz .
    tar –xvzf High_Level_Design_Workshop.tar.gz
    ```

4. Open up a terminal and connect to a 16G machine
    a. Type

    ```
    arc submit -I mem=16000 -- konsole
    ```

5. Retrieve the ARC resources needed to launch Quartus Prime Pro Edition, ModelSim-Altera, and HLS compiler
    a. Type

    ```
    arc shell acds/19.1, modelsim_se-lic/hdl, acltest/19.1,
    qedition/pro, modelsim_se/10.6d, aclboardpkg/a10_sdk,
    gcc/7.2.0/1, hls, hlsgcc, modelsim_se
    ```

You now have all the necessary resources and are ready to begin the exercises!

2. RTL Implementation of Simple Traffic Light State Machine

This section involves observing the Verilog code and ModelSim simulation for a simple 4-state traffic light controller. It will provide a showcase of how easy it is to design and simulate a state machine directly in RTL as well as a setup for Section 3, where we will compare the ease of designing/simulating a state machine in RTL with that of HLS.

2.1. Review the traffic controller state machine in Verilog

1. From the terminal, navigate to the directory where the Quartus project is stored and open it
   a. Type
      `cd High_Level_Design_Workshop/Lab1/traffic_control_rtl`
   b. Type
      `quartus traffic_control.qpf &`
      to open the Quartus project.
2. In the Project Navigator window located on the left, click the **Files** tab and double click *traffic_control_tb.v* and *traffic_control.v* open the Verilog files

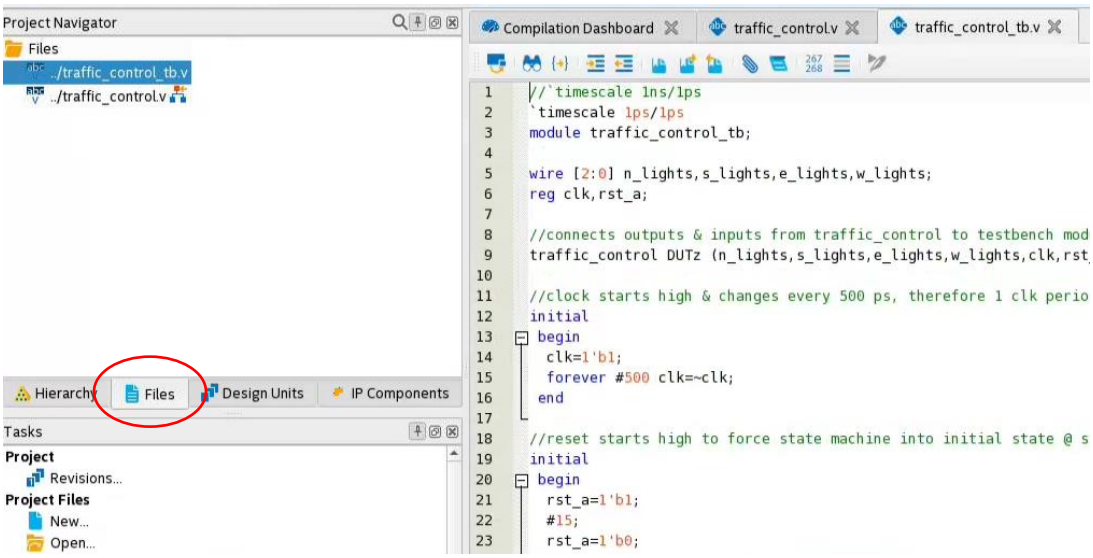


Figure 3: Project Navigator File Tab

3. Inspect the two files for details on the state machine algorithm & test bench (for simulation). Comments throughout the code will explain what is going on & is summarized below. When ready, minimize Quartus, as it will be revisited in Section 4 when comparing the Compilation Reports generated from RTL & from HLS.

The traffic controller state machine contains four states as shown in **Figure 4**. Each state contains four 3-bit wide outputs used to describe the status of all the lights at the intersection:

a. *north_south* is the initial state of the traffic controller. In this state, the north & south traffic light outputs are green. Consequently, the east & west traffic light outputs are red. The state machine remains in this state for 60 clock cycles, as more cars tend to travel north/south compared to east/west at this intersection.

b. *north_south_y* is the next state the controller transitions into, where the north & south traffic light outputs are yellow and the east & west traffic light outputs are red. The state machine remains in this state for 3 clock cycles.

c. *east_west* is the next state the controller transitions into, where the north & south traffic light outputs are red and the east & west traffic light outputs are green. The state machine remains in this state for 40 clock cycles, as less cars tend to travel east/west compared to north/south at this intersection.

d. *east_west_y* is the final state the controller transitions into, where the north & south traffic light outputs are red and the east & west traffic light outputs are yellow. The state machine remains in this state for 3 clock cycles. After this state, the controller transitions back into the initial state *north_south.*

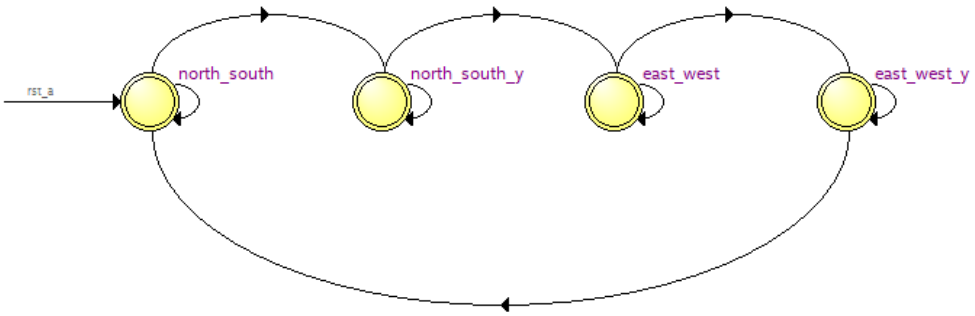


Figure 4: Traffic Controller State Machine (this view is from Quartus Lite only)

2.2. Observe the traffic controller simulation

1. From the terminal, type the following command to open the ModelSim project for the traffic controller simulation

    a. **`vsim simulation/modelsim/traffic_control_sim.mpf &`**

2. In the *Transcript* window located at the bottom of the ModelSim session, type in the following command to run the simulation script:

    a. **`do wave.do`**

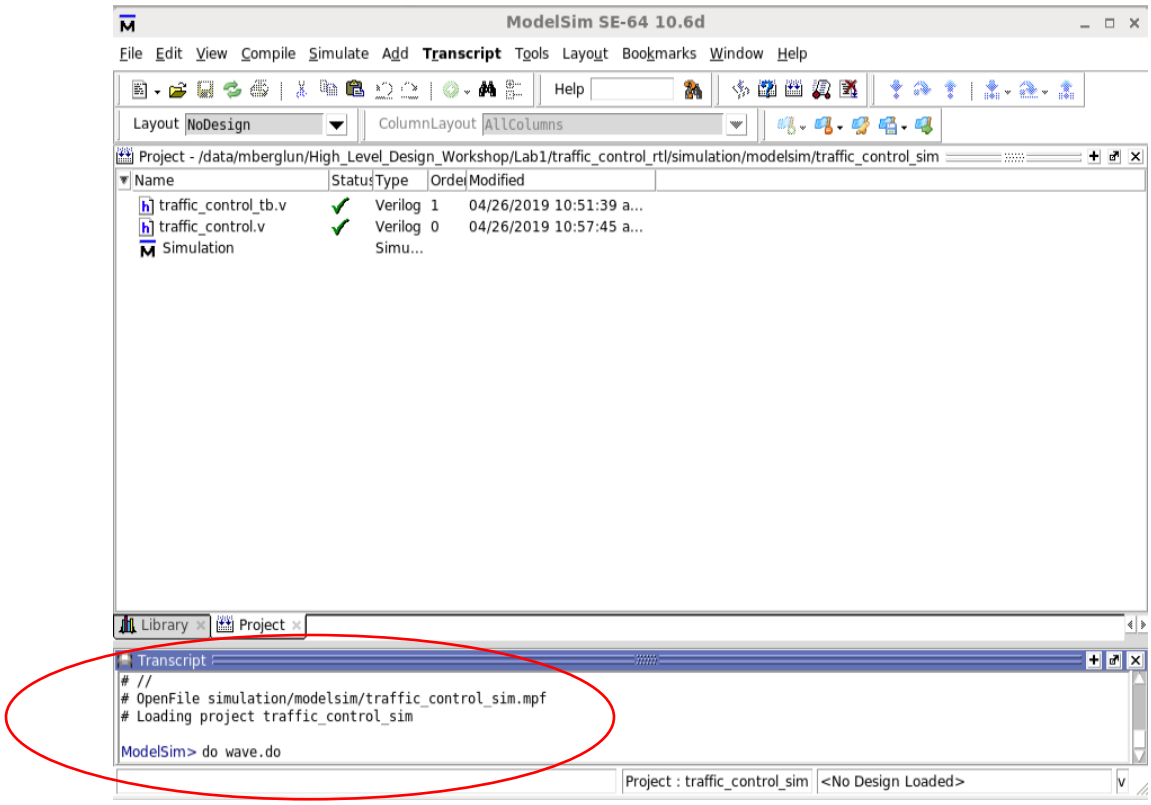


Figure 5: Run ModelSim simulation

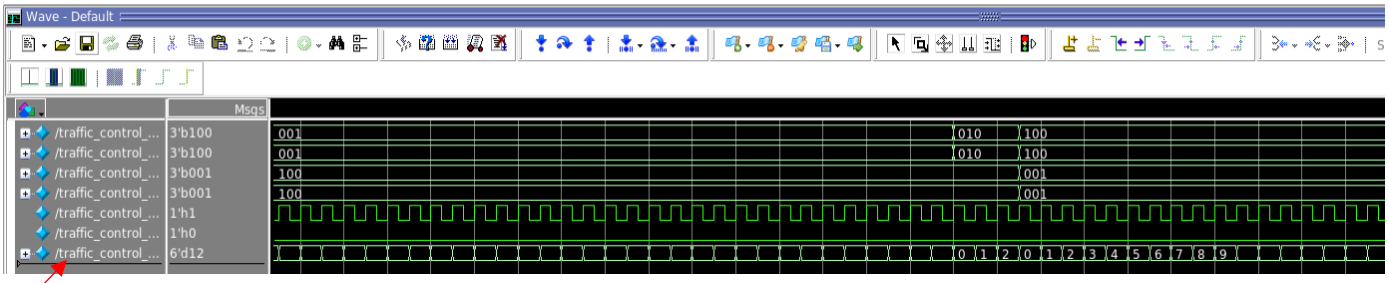
3. Maximize the *Wave* window which opened from the previous command.
4. Press F to get a full view of the simulation from start to finish.
5. Zoom in to obtain the view shown in Figure 6. This will give a better view of the output values at a given point in the simulation.

   i : keyboard shortcut to zoom in
   o : keyboard shortcut to zoom out

Notice the *n_lights* and *s_lights* output signals transitioning from 001 to 010 to 100 before returning to 001. Consequently, the *e_lights* and *w_lights* output signals transition from 100 to 001 to 010 before returning to 100.

- **001 represents green**
- **010 represents yellow**
- **100 represents red**

Upon closer inspection, you can see that the simulation stays in each state for the corresponding amount of clock cycles mentioned in Section 2.1 step 3.
Note, you may need to zoom in further to properly observe the state durations (bottom signal of simulation)



Counter signal determining state durations

Figure 6: Traffic Control simulation (RTL)

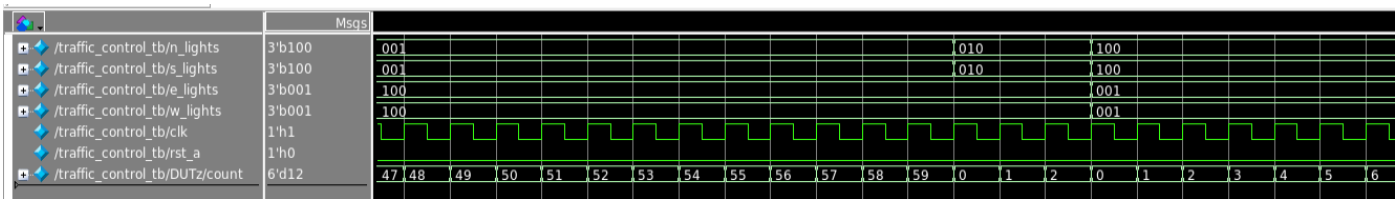


Figure 7: Zoomed in further to show counter values

3. HLS Implementation of Simple Traffic Light State Machine

The HLS procedure contains three main steps to integrate a C/C++ algorithm down into RTL. The first step is called *emulation*, which is nothing more than a functional verification of the C/C++ source code that was written. This compiler is essentially the exact same as the standard g++ compiler used in the C/C++ programming language; this will not generate any RTL. The next step is called *cosimulation*, which generates both RTL & compiler reports for a targeted FPGA device, and also allows you to verify your design in simulation. The final step involves running an Intel Quartus Prime compile on all the generated RTL files to obtain a more accurate report on resource utilization and the clock's fmax. This is essentially the same as step 2 with one additional part.

This section will walk you through this HLS flow by first performing emulation, then a cosimulation, and finally a Quartus compile.

3.1. Observe the HLS emulation of the state machine

1. From the terminal, navigate to the traffic_control_hls directory
   a. Type
      ```
      cd ../traffic_control_hls
      ```
2. Inspect the C++ implementation of the traffic light controller
   a. Type
      ```
      gedit traffic_control.cpp &
      ```
   As in the RTL implementation, the algorithm starts in an initial state, where the north & south lights are green and east & west lights are red. After a certain duration, it moves to the next state, and so on. Comments throughout the code explain the C++ implementation for the traffic controller state machine, and any discrepancies between the Verilog version (also discussed at the end of this section).

   Close the gedit window.
3. Run an emulation compile on traffic_control_fin.cpp
   a. Type
      ```
      i++ -march=x86-64 traffic_control.cpp -o emulation
      ```
   This command performs the same compilation as the general g++ compiler used in C++ but using the i++ Intel® HLS Compiler instead.
   *Note that in **cosimulation**, there are HLS-unique directives such as 'component' (used to synthesize a C++ function down to RTL) and '#pragma max concurrency N' (used to pipeline loop instructions).*
   *In **emulation**, these directives are simply ignored in compilation because it is set to mimic the g++ compilation flow.*
4. Run the executable (named 'emulation' in this case) generated from the compilation performed in step 3.

a. Type

./emulation

```
<<<<---CURRENT STATUS--->>>>
**Initial state sets state assignments & schedules busy loop in simulation**
Initialization complete (~60 cycles). Entering state machine

<<<<---CURRENT STATUS--->>>>
North & South lights are GREEN, East & West lights are RED
60 seconds have passed, traffic light transitioning to next state

<<<<---CURRENT STATUS--->>>>
North & South lights are YELLOW, East & West lights are RED
3 seconds have passed, traffic light transitioning to next state

<<<<---CURRENT STATUS--->>>>
North & South lights are RED, East & West lights are GREEN
40 seconds have passed, traffic light transitioning to next state

<<<<---CURRENT STATUS--->>>>
North & South lights are RED, East & West lights are YELLOW
3 seconds have passed, traffic light transitioning to next state

<<<<---CURRENT STATUS--->>>>
North & South lights are GREEN, East & West lights are RED
60 seconds have passed, traffic light transitioning to next state
```

Figure 8: Terminal output after running 'emulation' executable

The output produced from emulation (shown in Figure 8) is nothing more than a functional verification of the C++ code. From the output above, we can deduce that the algorithm transitions to the next appropriate states, but have no notation of how many clock cycles a given state remains in. Because of its fast compilation time, emulation provides a quick turn-around in functional verification of code compared to RTL.

3.2. Observe the HLS cosimulation of the state machine

The following is generated when specifying cosimulation to the HLS compiler:

- An executable which will run the testbench (the main() function in C++). Any calls to functions from the testbench will be part of the simulation.
- All files necessary to include IP in an Intel Quartus software project (i.e. .qsys, .ip, .v, ect)
- Component hardware implementation report (estimation)
- Simulation testbench
- Intel Quartus software project (.qpf) to compile all the IP and generate a more accurate report)

1. Run a cosimulation compile on traffic_control.cpp

   * OPTIONAL: Compilation time takes ~ 3 minutes for this design. A waveform file has already been generated from a previous compile, so steps 1 and 2 may be skipped.

   a. Type

   i++ -ghdl -march=Arria10 traffic_control.cpp -o cosimulation

   This is when HLS directives placed within the C++ code take an effect.

*The **component** identifier placed in front of the function state_logic (line 35) specifies to the compiler to generate RTL for this function and simulate any outputs from it.*

***#pragma max concurrency 1** on line 39 specifies to iterate through the loop without pipelining. This specification is actually ignored due to the inner while-loop (line 74) being pipelined (called loop iteration ordering), but showcases common usage of a popular HLS directive.*

This compilation will generate RTL for any functions marked as a component, create a Quartus project, a Compilation Report, and a testbench.

2. Run the executable (named 'cosimulation' in this case) generated from the compilation performed in step 3. This executable runs the main() function and serves as the testbench in simulation.
    a. Type

    `./cosimulation`

    Execution will generate a .wlf waveform file

3. Launch the *vsim.wlf* file generated from step 2a through ModelSim-Intel.
    a. Type

    `vsim cosimulation.prj/verification/vsim.wlf &`

4. In the *Transcript* window located at the bottom of the ModelSim session, type in the following command to run the simulation script:
    a. `do hls_wave.do`

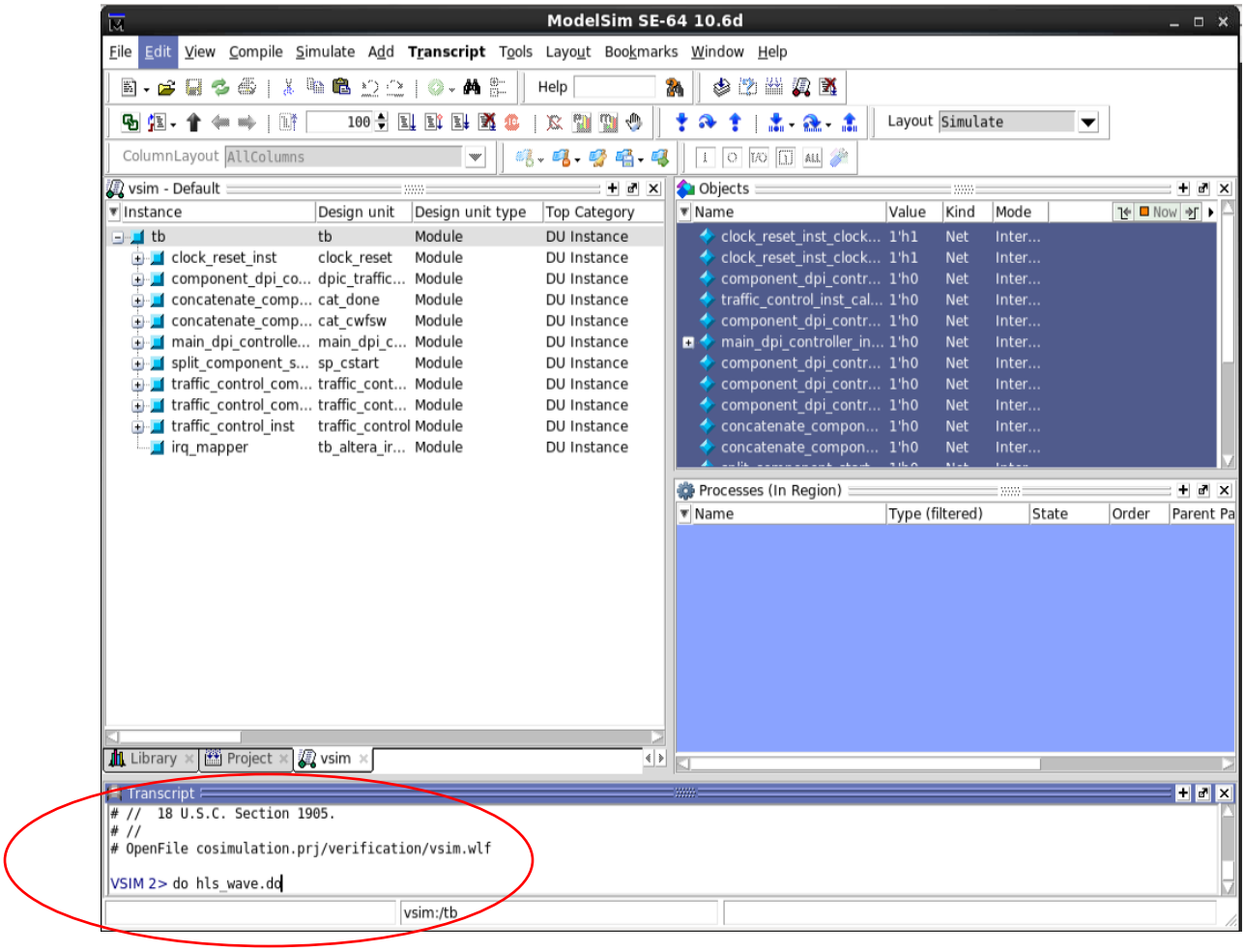


Figure 9: Run ModelSim Simulation

5. Maximize the *Wave* window which opened from the previous command.
6. Press F to view the simulation from start to finish. Recall you can zoom in/out with the I/O keys of your keyboard.

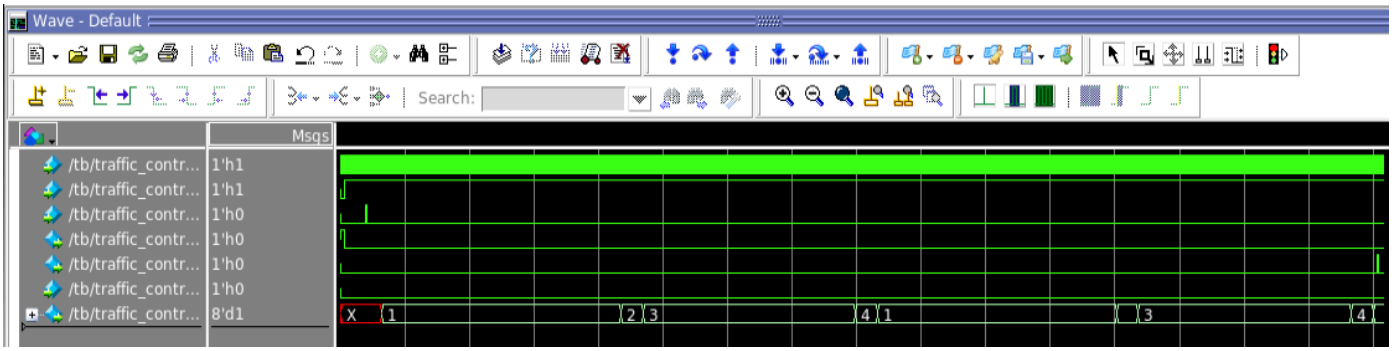


Figure 10: Traffic Control simulation (HLS)

Upon closer inspection, you will likely immediately notice three things:

1. *There is only one output (returndata) as opposed to four outputs seen in the RTL simulation. Referring to traffic_control.cpp, you can see that the states declared in lines 27 through 31 actually represent the decimal numbers seen.*
    a. *LIGHT_STATE_ENTRY (decimal 0 in simulation) represents the initial state (see explanation below)*
    b. *LIGHT_STATE_NS_GREEN_EW_RED (decimal 1 in simulation) represents the next state where north & south lights are green and east & west lights are red.*
    c. *LIGHT_STATE_NS_YELLOW_EW_RED (decimal 2 in simulation) represents the next state where north & south lights are yellow and east & west lights are red*
    d. *LIGHT_STATE_NS_RED_EW_GREEN (decimal 3 in simulation) represents the next state where north & south lights are red and east & west lights are green*
    e. *LIGHT_STATE_NS_RED_EW_YELLOW (decimal 4 in simulation) represents the next state where north & south lights are red and east & west lights are yellow*
2. *The return data is undefined for some time at the beginning of the simulation. This is why the extra state LIGHT_STATE_ENTRY was added to the state machine. The undefined output is mainly due to the way the compiler schedules the busy loop and state assignments, so the extra LIGHT_STATE_ENTRY state was added to give an accurate representation of the state machine in ModelSim.*
3. *A clock cycle offset of 10 was discovered after initial simulation. This means a value designed to hold for 1 clock cycle is actually held for 11 in simulation, due to the offset. To combat this, all state durations were scaled up by a factor of 4 to create correct cycle durations with respect to one another.*
    a. *N/S lights GREEN, E/W lights RED : 60 cycles → 240 cycles*
    b. *N/S lights YELLOW, E/W lights RED : 3 cycles → 12 cycles*
    c. *N/S lights RED, E/W lights GREEN : 40 cycles → 160 cycles*
    d. *N/S lights RED, E/W lights YELLOW : 3 cycles → 12 cycles*

4. Compare Compilation Reports

4.1. RTL Compilation Report

1. Bring Quartus session launched in Section 2 back up
    a. If closed, type **`quartus traffic_control.qpf &`** to re-open the RTL project
    b. Open up the Compilation Report by typing **`Ctrl + R`**
2. Observe the number of ALMs and registers used from the Verilog-compiled RTL.
    a. ALMS = 12, Registers = 8

    The register count comes from the following:

    - *6 registers for 'count' (line 18 in traffic_control.v)*
    - *2 registers for 'state' (line 30 in traffic_control.v)*

    Adaptive Logic Module (ALM) is the basic building block of supported FPGA device families and is designed to maximize performance and resource usage; the ALM count is relative to the number of logic elements (LEs) used in the design.

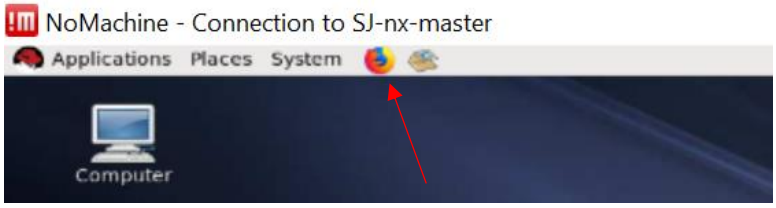
4.2. HLS Compilation Report

1. Navigate to the directory containing the HLS report generated from cosimulation
    1. Type

    `cd ../traffic_control_hls/cosim_quartus_compile.prj/reports`

    This .prj directory was generated with the command ' i++ -march=Arria10 --quartus-compile traffic_control.cpp -o cosim_quartus_compile ' which performs an Intel Quartus Prime software compilation on all the components in the cpp file.

2. Obtain the full path to this directory
    1. Type **`pwd`**
3. Open report.html in Firefox to view the cosimulation report
    1. Click on the Firefox icon located at the top left of your NoMachine session



4. Enter the path returned from the pwd command into the address line of Firefox, directly after *file:///* followed by report.html
    1. **`file:///<pwd path returned>/report.html`**
5. Observe the number of ALMs and FFs used from the HLS-compiled RTL under the section "Quartus Fit Resource Utilization Summary" (you might need to scroll down)

1. ALMs = 94.5, FFs = 152

Note: the "Estimated Resource Usage" was produced directly from the cosimulation performed in Section 3.2.1 but is only a rough estimate, as suggested. A more accurate resource count was retrieved using the command mentioned in Step 1 of this section. Details on the estimated resource utilization can be seen in the "Area Analysis" tab of the report.

An increase in resource utilization is expected due to the overhead in the tool we're using. Generally speaking, the higher level a tool is when designing, the more control is abstracted away from the designer. In software, this can be thought of in terms of an assembly language versus a high level language (like Java or C). In assembly, the designer has more control because they can write values directly to registers in the CPU. This is often described as the computer language, as the CPU is able to interpret instructions directly; there is no compiler or translator. In high level languages, common activities such as handling datatypes and complex syntax is simplified, but ultimately gives less control to the designer.

In terms of our traffic controller design, this overhead can most easily be seen in the durations we define to each state. In RTL, we can easily define the number of clock cycles to a given state because our counter is based on the positive edge of our input clock. This is made possible through use of 'always' blocks in hardware-descriptive languages. In C++, staying in a state for a definitive number of clock cycles is not as simple because we are not able to execute instructions based on the rising or falling edge of a clock cycle, as we are in RTL.

Certain constraints were put in place in the C++ domain to get the HLS compiler to synthesize the code down to the desired RTL. This came with the cost of greater resource utilization; mainly due to the computations & loop dependencies (seen in Area Analysis of the HLS report).

5. RTL Implementation of Parallel Multiplier

5.1. Review the Parallel Multiplier in Verilog

1. From the terminal, navigate to the directory where the Quartus project is stored and open it

   a. Type

   ```
   cd  ../../../../Lab2/Lab2_RTL
   ```

   b. Type

   ```
   quartus parallel_mult.qpf &
   ```

   to open the Quartus project.

   c. In the Restored Archive Project pop-up, leave the Archive name & Destination folder to their default paths. Click **OK**.

2. In the Project Navigator window located on the left, click the **Files** tab and double click *tb_parallel_mult.v* and *parallel_mult.v* open the Verilog files

```
module parallel_mult (
  input clock_200,
  input rst_n,
  input [255:0] from_mem_controller,
  output reg [255:0] to_mem_controller);

  wire clock_400_pll;
  reg [255:0] multiplicand, multiplicand_d1;
  reg [255:0] multiplier, multiplier_d1;
  reg [511:0] product_d1  /* synthesis preserve */ ;
  wire [511:0] product /* synthesis preserve */ ;
  pll i_pll (.refclk(clock_200), .rst(rst_n), .outclk_0(clock_200_pll), .outclk_1(clock_400_pll), .locked());

mult32_4stage u0( .dataa(multiplicand_d1[31:0]),  .datab(multiplier_d1[31:0]), .result(product[63:0]), .clock(clock_200_pll));
mult32_4stage u1( .dataa(multiplicand_d1[63:32]),  .datab(multiplier_d1[63:32]), .result(product[127:64]), .clock(clock_200_pll));
mult32_4stage u2( .dataa(multiplicand_d1[95:64]),  .datab(multiplier_d1[95:64]), .result(product[191:128]), .clock(clock_200_pll));
mult32_4stage u3( .dataa(multiplicand_d1[127:96]),  .datab(multiplier_d1[127:96]), .result(product[255:192]), .clock(clock_200_pll));
mult32_4stage u4( .dataa(multiplicand_d1[159:128]),  .datab(multiplier_d1[159:128]), .result(product[319:256]), .clock(clock_200_pll));
mult32_4stage u5( .dataa(multiplicand_d1[191:160]),  .datab(multiplier_d1[191:160]), .result(product[383:320]), .clock(clock_200_pll));
mult32_4stage u6( .dataa(multiplicand_d1[223:192]),  .datab(multiplier_d1[223:192]), .result(product[447:384]), .clock(clock_200_pll));
mult32_4stage u7( .dataa(multiplicand_d1[255:224]),  .datab(multiplier_d1[255:224]), .result(product[511:448]), .clock(clock_200_pll));

  always @(posedge clock_200_pll) begin
       multiplicand <= from_mem_controller; //even words
        multiplicand_d1 <= multiplicand;
        multiplier_d1 <= multiplier;
        product_d1 <= product;
  end

  always @(negedge clock_200_pll) begin
       multiplier <= from_mem_controller; //odd words
  end

  always @(posedge clock_400_pll) begin //repack into 4 64 bit products @ 400 MHz
      if (clock_200_pll == 0) to_mem_controller[255:0] <= product_d1[255:0]; // first 4 products
     else to_mem_controller <= product_d1[511:256];                          // second 4 products
  end

endmodule
```

Figure 11: Parallel Multiplier RTL Code

```
`timescale 1 ns / 1 ps
module tb_parallel_mult;
  reg clock_200 = 1'b0;
  reg [255:0] from_mem_controller;
  wire [255:0] to_mem_controller;

  wire [31:0] multiplier_0, multiplier_1, multiplier_2, multiplier_3, multiplier_4, multiplier_5, multiplier_6, multiplier_7;
  wire [31:0] multiplicand_0, multiplicand_1, multiplicand_2, multiplicand_3, multiplicand_4, multiplicand_5, multiplicand_6, multiplicand_7;
  wire [63:0] product_0, product_1, product_2, product_3, product_4, product_5, product_6, product_7;

  assign multiplier_0 = DUT.multiplier_d1[31:0];
  assign multiplicand_0 = DUT.multiplicand_d1[31:0];
  assign product_0 = DUT.product[63:0];

  assign multiplier_1 = DUT.multiplier_d1[63:32];
  assign multiplicand_1 = DUT.multiplicand_d1[63:32];
  assign product_1 = DUT.product[127:64];

  assign multiplier_2 = DUT.multiplier_d1[95:64];
  assign multiplicand_2 = DUT.multiplicand_d1[95:64];
  assign product_2 = DUT.product[191:128];

  assign multiplier_3 = DUT.multiplier_d1[127:96];
  assign multiplicand_3 = DUT.multiplicand_d1[127:96];
  assign product_3 = DUT.product[255:192];

  assign multiplier_4 = DUT.multiplier_d1[159:128];
  assign multiplicand_4 = DUT.multiplicand_d1[159:128];
  assign product_4 = DUT.product[319:256];

  assign multiplier_5 = DUT.multiplier_d1[191:160];
  assign multiplicand_5 = DUT.multiplicand_d1[191:160];
  assign product_5 = DUT.product[383:320];

  assign multiplier_6 = DUT.multiplier_d1[223:192];
  assign multiplicand_6 = DUT.multiplicand_d1[223:192];
  assign product_6 = DUT.product[447:384];

  assign multiplier_7 = DUT.multiplier_d1[255:224];
  assign multiplicand_7 = DUT.multiplicand_d1[255:224];
  assign product_7 = DUT.product[511:448];


  parallel_mult DUT (.clock_200(clock_200), .from_mem_controller(from_mem_controller), .to_mem_controller(to_mem_controller));

  initial
    begin
      clock_200 = 0;
       forever #2.5 clock_200 = ~clock_200;
    end

  initial
    begin
      #400; // This will give time for PLL to lock
      #2.5  from_mem_controller = 256'h0000000E_0000000C_0000000A_00000008_00000006_00000004_00000002_00000000; //multiplier
      #2.5  from_mem_controller = 256'h0000000F_0000000D_0000000B_00000009_00000007_00000005_00000003_00000001; //multiplicand
      #2.5  from_mem_controller = 256'h00000010_0000000E_0000000C_0000000A_00000008_00000006_00000004_00000002; //multiplier
      #2.5  from_mem_controller = 256'h00000011_0000000F_0000000D_0000000B_00000009_00000007_00000005_00000003; //multiplicand
      #2.5  from_mem_controller = 256'h00000012_00000010_0000000E_0000000C_0000000A_00000008_00000006_00000004; //multiplier
      #2.5  from_mem_controller = 256'h00000013_00000011_0000000F_0000000D_0000000B_00000009_00000007_00000005; //multiplicand
      #2.5  from_mem_controller = 256'h00000014_00000012_00000010_0000000E_0000000C_0000000A_00000008_00000006; //multiplier
      #2.5  from_mem_controller = 256'h00000015_00000013_00000011_0000000F_0000000D_0000000B_00000009_00000007; //multiplicand
      #2.5  from_mem_controller = 256'h00000016_00000014_00000012_00000010_0000000E_0000000C_0000000A_00000008; //multiplier
      #2.5  from_mem_controller = 256'h00000017_00000015_00000013_00000011_0000000F_0000000D_0000000B_00000009; //multiplicand
      #2.5  from_mem_controller = 256'h00000018_00000016_00000014_00000012_00000010_0000000E_0000000C_0000000A; //multiplier
      #2.5  from_mem_controller = 256'h00000019_00000017_00000015_00000013_00000011_0000000F_0000000D_0000000B; //multiplicand
      #2.5  from_mem_controller = 256'h0000001A_00000018_00000016_00000014_00000012_00000010_0000000E_0000000C; //multiplicand
      #2.5  from_mem_controller = 256'h0000001B_00000019_00000017_00000015_00000013_00000011_0000000F_0000000D; //multiplier
      #2.5  from_mem_controller = 256'h0000001C_0000001A_00000018_00000016_00000014_00000012_00000010_0000000E; //multiplicand
      #10000 $stop;
    end
endmodule
```

Figure 12: Parallel Multiplier RTL Test-Bench Code

3. Open the Compilation Report located under the Compilation section of the Tasks tab in Quartus
    a. If the tasks tab is not visible upon opening Quartus, go to **View → Tasks** to display it
    b. Next, observe the RTL syntax and construction (if you see checkmarks then its already compiled if not then you need to compile it)
    c. Then go to Tools → RTL Viewer

The following an example of the generated diagram used for the RTL implementation of the parallel multiplier.

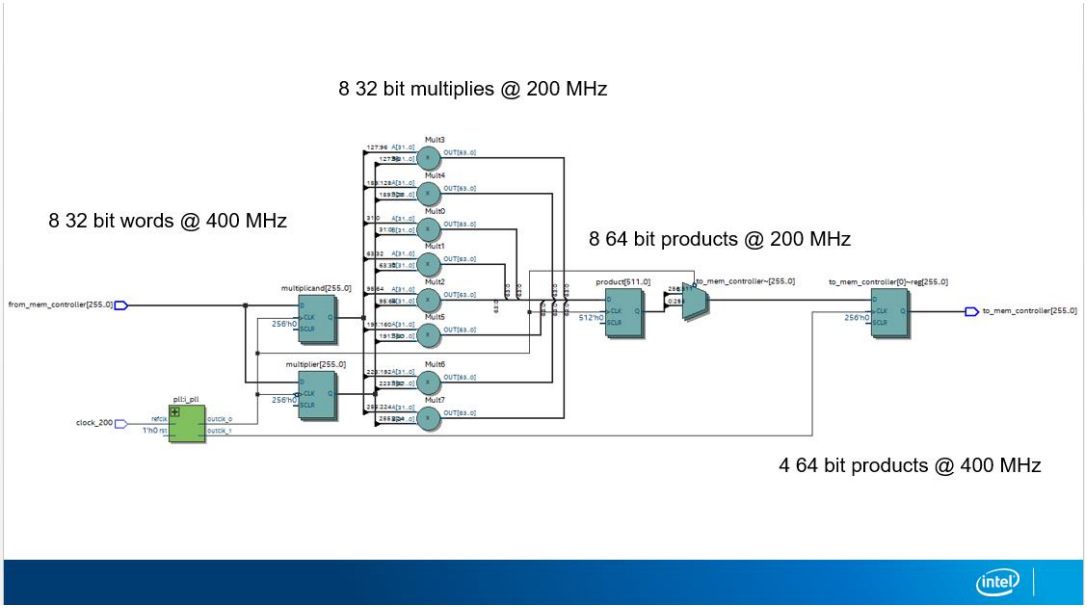


Figure 13: Parallel Multiplier RTL Diagram

5.2. * Observe the Parallel Multiplier Simulation * (Don't Do! Skip To Figure)

6. From the terminal, type the following command to open the ModelSim project for the traffic controller simulation
    a. **vsim simulation/parallel_mult.mpf &**
7. In the Project tab on the ModelSim session, double click *hdl_simulation* located under the two green checks (signifying properly compiled Verilog files).
8. In the Objects plane (dark blue window in ModelSim session), select all of the signals, right click, and select *Add Wave* to add the signals to your waveform.

    If you are unable to see the Objects plane, click View → Objects to open it.
9. Move to the Wave tab and expand it to full screen by clicking the small '+' button on the top right of the window.
10. Click **Simulate → Run → Run-All** (you may have to return to the Wave tab after doing this).
11. You will need to change the radix of your symbols from Binary to Decimal. Do this by selecting all the signals located to the left of your waveforms
    a. **Right click all the symbols → Radix → Decimal**
    b. Expand the window containing the signal names to display the entire name of all the signals
12. Press F to get a full view of the simulation from start to finish.

Note In the interest of time we will not be able to manually arrive at the simulation environment. Instead, a completed waveform will be provided.

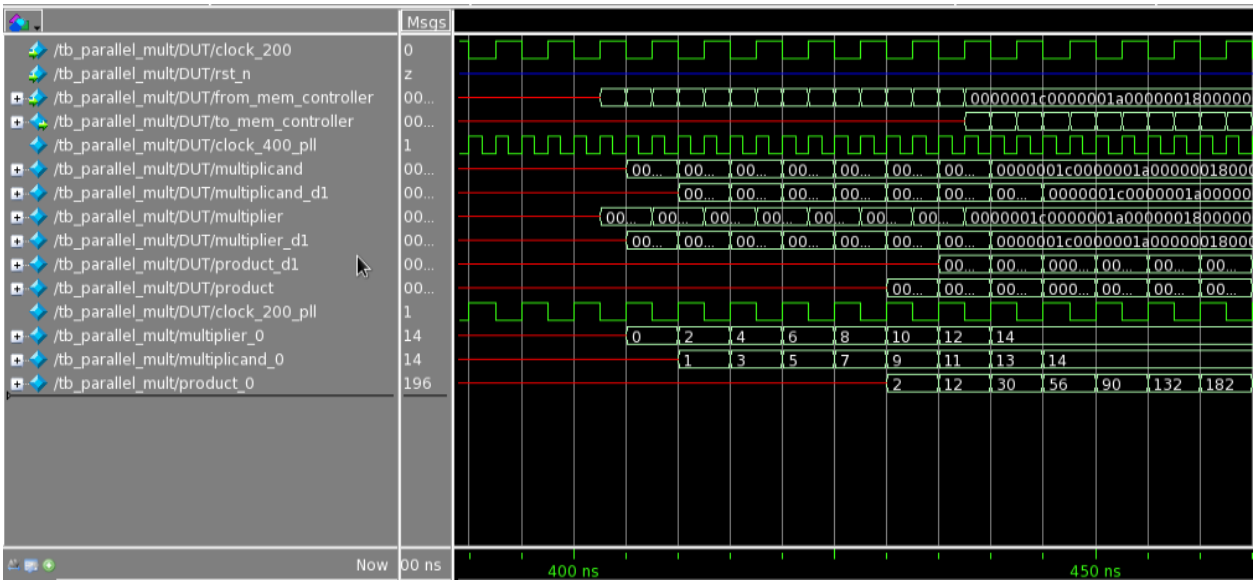


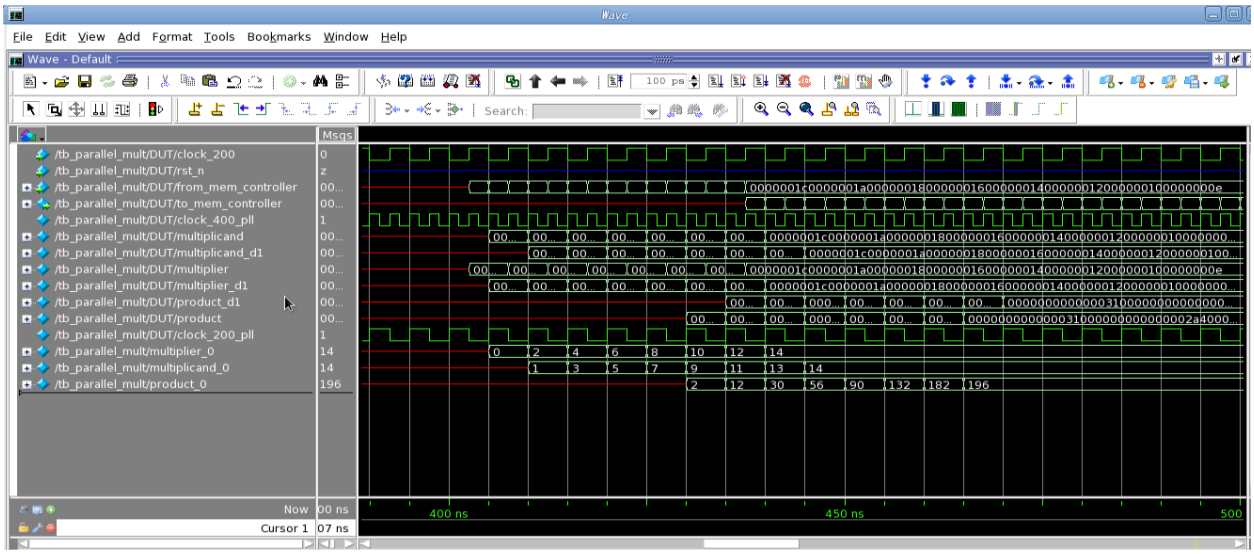
Figure 14: Parallel Multiplier RTL Test-Bench Simulation

As can be seen by the Figure 13, the multiplier inputs named “multiplier_0” and “multiplicand_0”, act as 256-bit registers which obtain a different value at the rising edge of each cycle of the “clock_200_pll” signal. After an initial latency of 4-cycles, the results of the multiplication operations are seen, in the 512-bit bus value named “product_0”. The simulation is organized in a manner as to facilitate the multiplication of the following pairs of numbers:

(1, 2), (3, 4), (5, 6), (7, 8), (9, 10), (11, 12), (13, 14).

The corresponding results should be as follows:

(0), (12), (30), (56), (90), (110), (132), (182).


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Search:
Msgs
/tb_parallel_mult/DUT/clock_200
/tb_parallel_mult/DUT/rst_n
/tb_parallel_mult/DUT/from_mem_controller
/tb_parallel_mult/DUT/to_mem_controller
/tb_parallel_mult/DUT/clock_400_pll
/tb_parallel_mult/DUT/multiplicand
/tb_parallel_mult/DUT/multiplicand_d1
/tb_parallel_mult/DUT/multiplier
/tb_parallel_mult/DUT/multiplier_d1
/tb_parallel_mult/DUT/product_d1
/tb_parallel_mult/DUT/product
/tb_parallel_mult/DUT/clock_200_pll
/tb_parallel_mult/DUT/multiplier_0
/tb_parallel_mult/DUT/multiplicand_0
/tb_parallel_mult/DUT/product_0
Now
Cursor 1
400 ns
450 ns
500

6. HLS Implementation of Parallel Multiplier

6.1. Observe the HLS emulation of the parallel multiplier

1. From the terminal, navigate to the Lab2 directory
    a. Type **cd ../../../../Lab2**
2. Inspect the C++ implementation of the parallel multiplier
    a. Type **vim mult2.cpp &**
    b. Observe that there exists both a “main” section” and a “component” section. For the “component” code/function, the compiler will interpret it as a normal C++/C function during the emulation compile (shown in the following step).

    Additionally, the section of the code preceded by “main” will behave as the default C++/C driver function.
3. Run an emulation compile on mult2.cpp
    a. Type **i++ -march=x86-64 mult2.cpp –o mult2_out**
    b. An “emulation” compile will simply compile the “.cpp” or “.c” files using the default GNU compiler, generating the corresponding executable files (.exe) and running the code like a normal C/C++ program.
4. Execute emulation
    a. Type **./mult2_out**

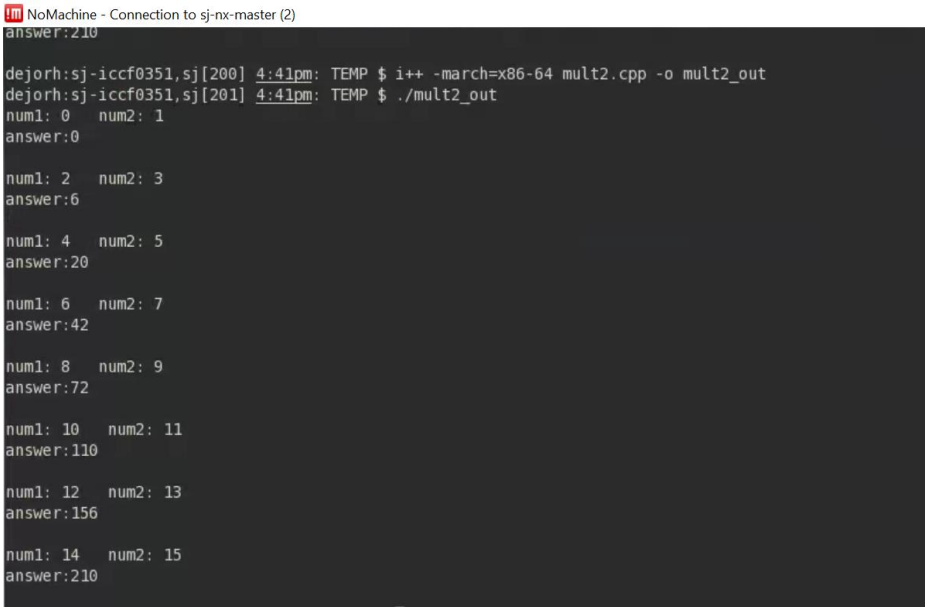


Figure 16: 8 bit Parallel Multiplies of 8-pairs of 32-bit integers

This is an example of Parallel Multiplication performed on incoming data stream. A 512-bit data stream is read, partitioned into 8-pairs of 32-bit multiplier inputs, and then multiplied. The multiplication occurs in parallel (via 8 32-bit multipliers) and the 8 64-bit products will be, subsequently, recombined/repackaged into a 512-bit outgoing word.

6.2. Observe the HLS cosimulation of the Parallel Multiplier

1. Run a cosimulation compile on mult2.cpp
    a. Type **i++ -ghdl -march=Arria10 mult2.cpp -o mult2_fpga**
2. Execute cosimulation
    a. Type **./mult2_fpga**

    Execution will generate a .wlf waveform file
3. Launch ModelSim-Altera to view the parallel multiplier simulation generated through HLS
    a. Type **vsim mutl2_fpga.prj/verification/vsim.wlf &**
4. On the vsim tab located on the left of the ModelSim instance, right click *mult_pair_inst* and click Add Wave
5. Maximize the window, change the Radix of all signals, and press F for a full view

Note: When performing the cosimulation compile, and subsequent waveform generation, it is worth noting that the "main" and "component" sections of the mult2.cpp file take on different roles than described in the previous emulation section 3.1. Specifically, the "component" code/function will be converted into a quartus-recognizable, HDL (Verilog) file. The "main" section will then take on the role of serving as a testbench for the RTL simulation. When viewed, the waveform will simulate the functional behavior of the "component" code using the interactions encountered by each instantiation of the "component" as expressed within the code for the "main" section.

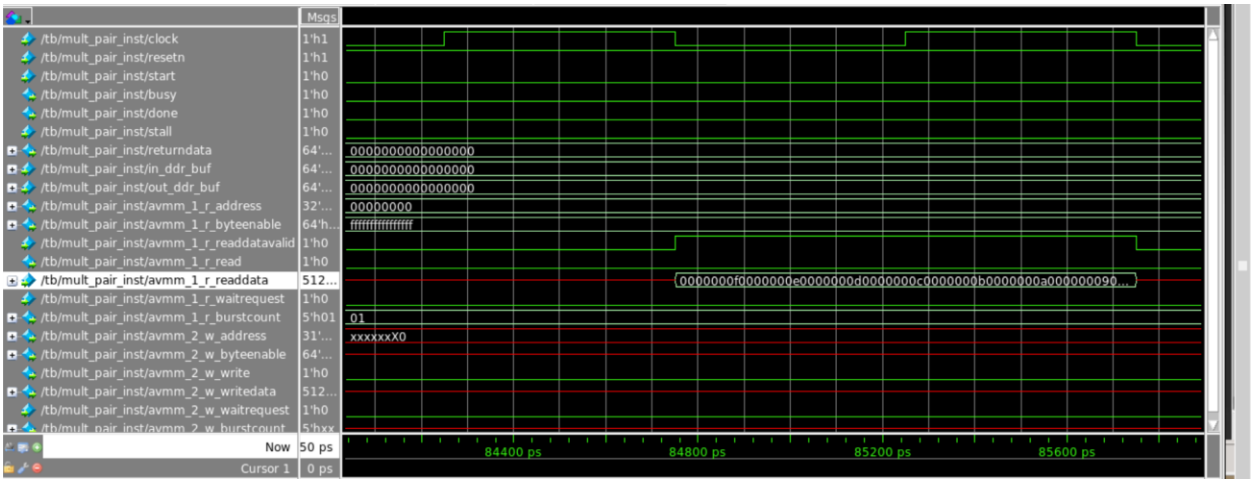


Figure 17: Parallel Multiplier (memory reads) (zoomed in view)

As can be seen by the above waveform, the "**avmm_1_r_readdata**" bus/signal represents the 512-bits of incoming data from the I/O stream. The packing of the data is done in such a way as to enable the bus to be easily partitioned into 8 pairs of 32-bit multiply inputs.
Quick Maths: (512 **bits incoming stream**/ 2 **multiply inputs**) / (**32** bits multiplied) = 8 **pairs**

Note that the 512- bit bus signal is represented by a hexadecimal value (4 bits binary). The total length of the signal in the hexadecimal domain is 128 digits long.
Quick Maths: (512 binary bits / 4 bits per hexadecimal digit) = 128 **hex digits**

These are the expected multiplier pairs coming from the stream:

(0, 1), (2, 3), (4, 5), (6, 7), (8, 9), (10, 11), (12, 13), (14, 15)

Notice that the most significant 8- hexadecimal digits is "0000000f" represents the value of $(15)_{10}$ and the second most significant 8 – hexadecimal digits "0000000e" represents the value of $(14)_{10}$.

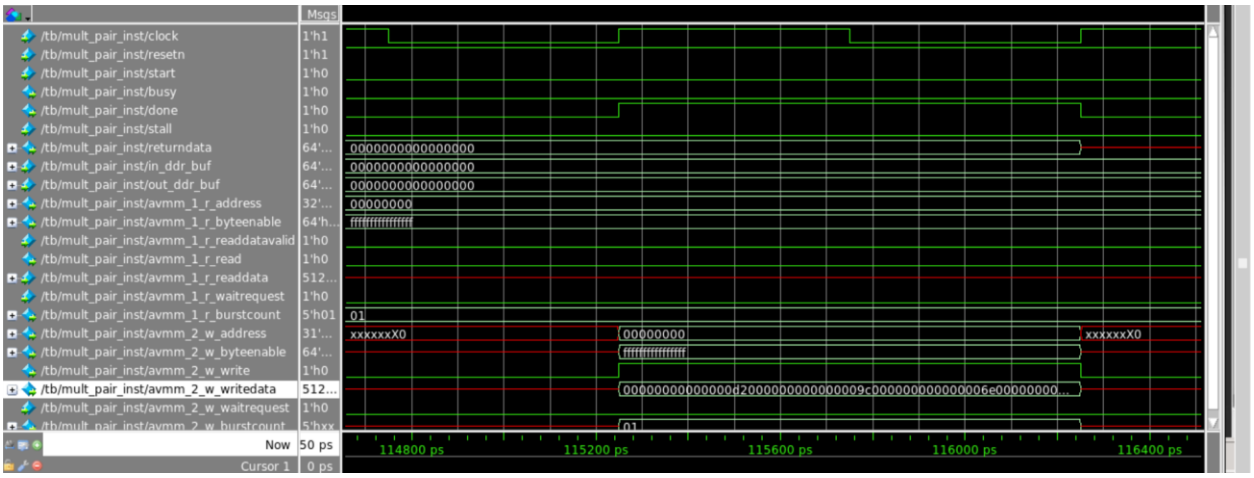


Figure 18: Parallel Multiplier (memory write-back) (zoomed in view)

The above Figure 5, illustrates the write-out of the resulting 512-bit product. The "**avmm_2_w_writedata**" bus/signal represents the 512-bits of outgoing data. The packing of the data is done in such a way as to enable the bus to be easily partitioned into 8 groupings of 64-bit multiply outputs.

These are the expected multiplier pairs coming from the stream:

(0), (6), (20), (42), (72), (110), (156), (210)

Notice that the most significant 16- hexadecimal digits is "00000000000000d2" represents the value of $(210)_{10}$

7. Compare Compilation Reports

7.1. RTL Compilation Report

1. Bring Quartus session launched in Section 2 back up
    a. If closed, type **quartus parallel_mult.qar &** to re-open the RTL project
    b. Open up the Compilation Report by clicking **Processing → Compilation Report**
2. Observe the number of ALMs and FFs used from the Verilog-compiled RTL.
    a. ALMS = 466, Registers = 1280 (note that 1024 are in the DSP blocks)

; Fitter DSP Block Usage Summary ;

Statistic	Number Used
Two Independent 18x18	16
Sum of Two 18x18	8
Total number of DSP blocks	24
Fixed Point Unsigned Multiplier	32

; Fitter Resource Usage Summary ;

Resource	Usage	%
Logic utilization (ALMs needed / total ALMs on device)	466 / 427,200	< 1 %
ALMs needed [=A-B+C]	466	
[A] ALMs used in final placement [=a+b+c+d]	225 / 427,200	< 1 %
[a] ALMs used for LUT logic and registers	75	
[b] ALMs used for LUT logic	96	
[c] ALMs used for registers	54	
[d] ALMs used for memory (up to half of total ALMs)	0	
[B] Estimate of ALMs recoverable by dense packing	19 / 427,200	< 1 %
[C] Estimate of ALMs unavailable [=a+b+c+d]	260 / 427,200	< 1 %
[a] Due to location constrained logic	0	
[b] Due to LAB-wide signal conflicts	0	
[c] Due to LAB input limits	3	
[d] Due to virtual I/Os	257	
Difficulty packing design	Low	
Total LABs: partially or completely used	29 / 42,720	< 1 %
-- Logic LABs	29	
-- Memory LABs (up to half of total LABs)	0	
Combinational ALUT usage for logic	340	
-- 7 input functions	0	
-- 6 input functions	0	
-- 5 input functions	68	
-- 4 input functions	0	
-- <=3 input functions	272	
Combinational ALUT usage for route-throughs	74	
Dedicated logic registers	256	
-- By type:		
-- Primary logic registers	256 / 854,400	< 1 %

Figure 19: Parallel Multiplier (Case 1: 8 – 32-bit multiplies) RTL Resource Usage

7.2. HLS Compilation Report

1. From the terminal, navigate to the directory where the Quartus project is stored and open it.
2. Use the cursor to manually navigate to the “reports” folder. From your Desktop within NoMachine:

   Lab2 → mult2_fpga.prj → reports
3. Double click on the “reports.html” file and you should be re-directed to the reports page

Figure 20: Parallel Multiplier (Case 1: 8 – 32-bit multiplies) Resource Summary

As can be seen by the resource summary report for the 8 32-bit multiplies, there were 36 RAMs used, 5461 FlipFlops used, 1607 ALMs used, and 16 DSPs used by the HLS as opposed to 0 RAMs, 256 Registers, 466 ALMs, and 24 DSPs used by the RTL compiler. This can be seen as due to additional Avalon-Interface signals included with the HLS component, and additionally, more resources are used (possibly in pipelining) to improve speed of the execution. In the HLS code, 2 Avalon memory masters (of large size) are used to serve as inputs to the component, which could serve as an explanation for the high amount of Flipflops and RAMs.

Figure 21: Parallel Multiplier (Case 2: 4 – 32-bit multiplies) Resource Summary