ECSE-325
Digital Systems

Lab Project – ***Complete SHA256 System*** **Winter 2021**

In this lab you will complete the SHA256 system.

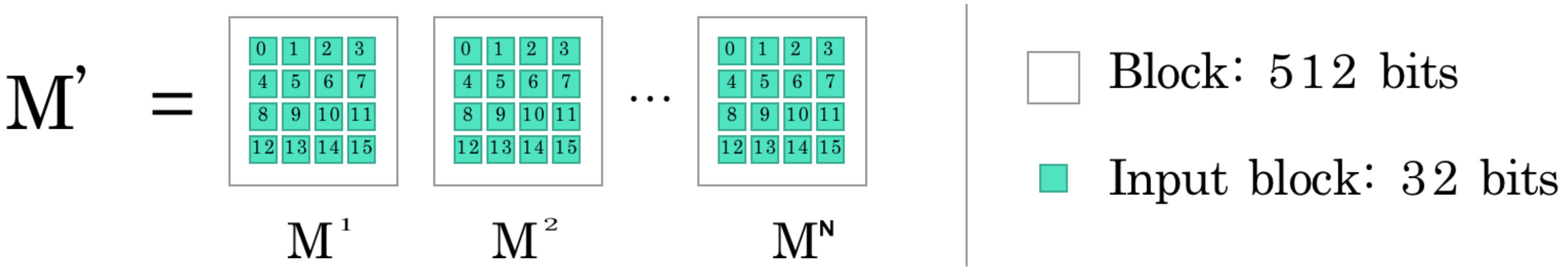
There will be four main tasks to do in the lab, some of which could be done in parallel (e.g., by different team members):

1. *Design of the message schedule circuit*
2. *Design of the Avalon Slave interface*
3. *Use Qsys to create the complete system, and connect to the HPS Arm CPU*
4. *Simulation of hashing a single message block*

Overall SHA256 system operation.

Take a look at the nice overview of the SHA256 system operation at:
https://medium.com/biffures/part-5-hashing-with-sha-256-4c2afc191c40

The first step in the overall hashing process is to break the message to be hashed into 512-bit blocks, structured as sixteen 32-bit words.



We will assume that the arrangement of the message into the blocks is done on the ARM CPU. Because of the pandemic restrictions, we will not actually write the program for the ARM CPU, and we will just focus on doing the hashing for each block.

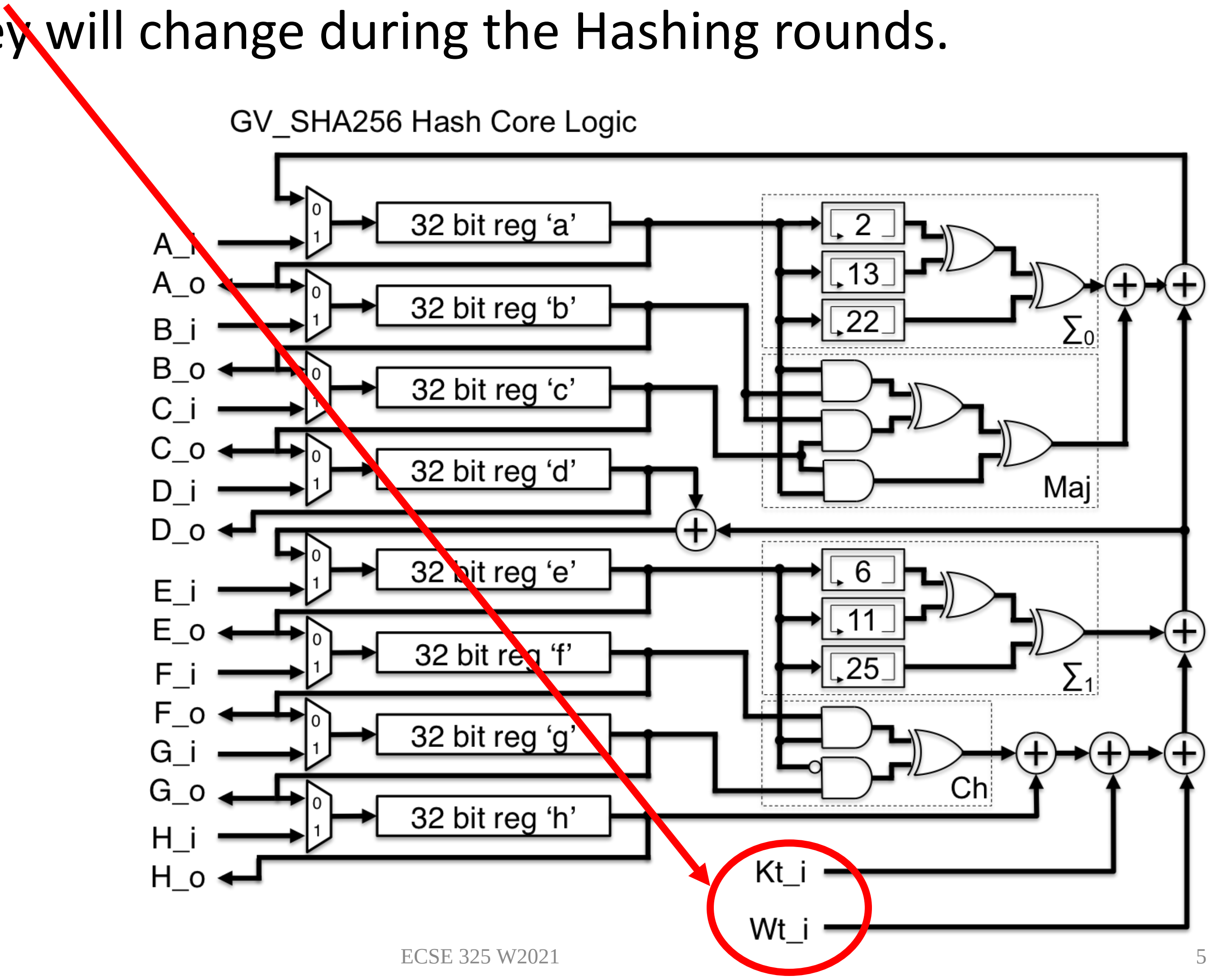
We will assume that the ARM CPU sends a 512-bit block to the SHA256 circuit, one 32-bit word at a time, to fill up the elements of a 16-element array of 32-bit vectors named ***M***.

```
type message_array is array(0 to 16) of std_logic_vector(31 downto 0);
signal M : message_array;
```

16 Avalon data transfers are needed to send a single message block to the hash circuit. Once these transfers have been done, the 64 rounds of the hashing operation can be done.

The Avalon interface will be designed later in this lab.
For now, we will focus on the hashing process that is applied to a single 512-bit message block.

Recall the SHA256 Hash Core Logic circuit that you created in earlier labs. In this lab you will create the logic to provide the ***Kt_i*** and ***Wt_i*** signals. Before, they were held constant, but now they will change during the Hashing rounds.



The ***Kt_i*** (i = 0,1,..63) are a set of 64 constant 32-bit values. The index ***i*** is what was called in Lab 3 as the "round_count".

These magical numbers are the first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers.

These can be stored in a constant signal array, as shown in the following VHDL snippet: (just copy this into your gNN_Hash_Core vhdl)

```
type constant_array is array(0 to 63) of std_logic_vector(31 downto 0);

constant Kt : constant_array := ( x"428a2f98", x"71374491", x"b5c0fbcf", x"e9b5dba5",
x"3956c25b", x"59f111f1", x"923f82a4", x"ab1c5ed5", x"d807aa98", x"12835b01",
x"243185be", x"550c7dc3", x"72be5d74", x"80deb1fe", x"9bdc06a7", x"c19bf174",
x"e49b69c1", x"efbe4786", x"0fc19dc6", x"240ca1cc", x"2de92c6f", x"4a7484aa",
x"5cb0a9dc", x"76f988da", x"983e5152", x"a831c66d", x"b00327c8", x"bf597fc7",
x"c6e00bf3", x"d5a79147", x"06ca6351", x"14292967", x"27b70a85", x"2e1b2138",
x"4d2c6dfc", x"53380d13", x"650a7354", x"766a0abb", x"81c2c92e", x"92722c85",
x"a2bfe8a1", x"a81a664b", x"c24b8b70", x"c76c51a3", x"d192e819", x"d6990624",
x"f40e3585", x"106aa070", x"19a4c116", x"1e376c08", x"2748774c", x"34b0bcb5",
x"391c0cb3", x"4ed8aa4a", x"5b9cca4f", x"682e6ff3", x"748f82ee", x"78a5636f",
x"84c87814", x"8cc70208", x"90befffa", x"a4506ceb", x"bef9a3f7", x"c67178f2"
);
```

You can access individual array elements just as you would individual bits in a vector, e.g.   X <= Kt(3); would assign `x"e9b5dba5"`   to X.

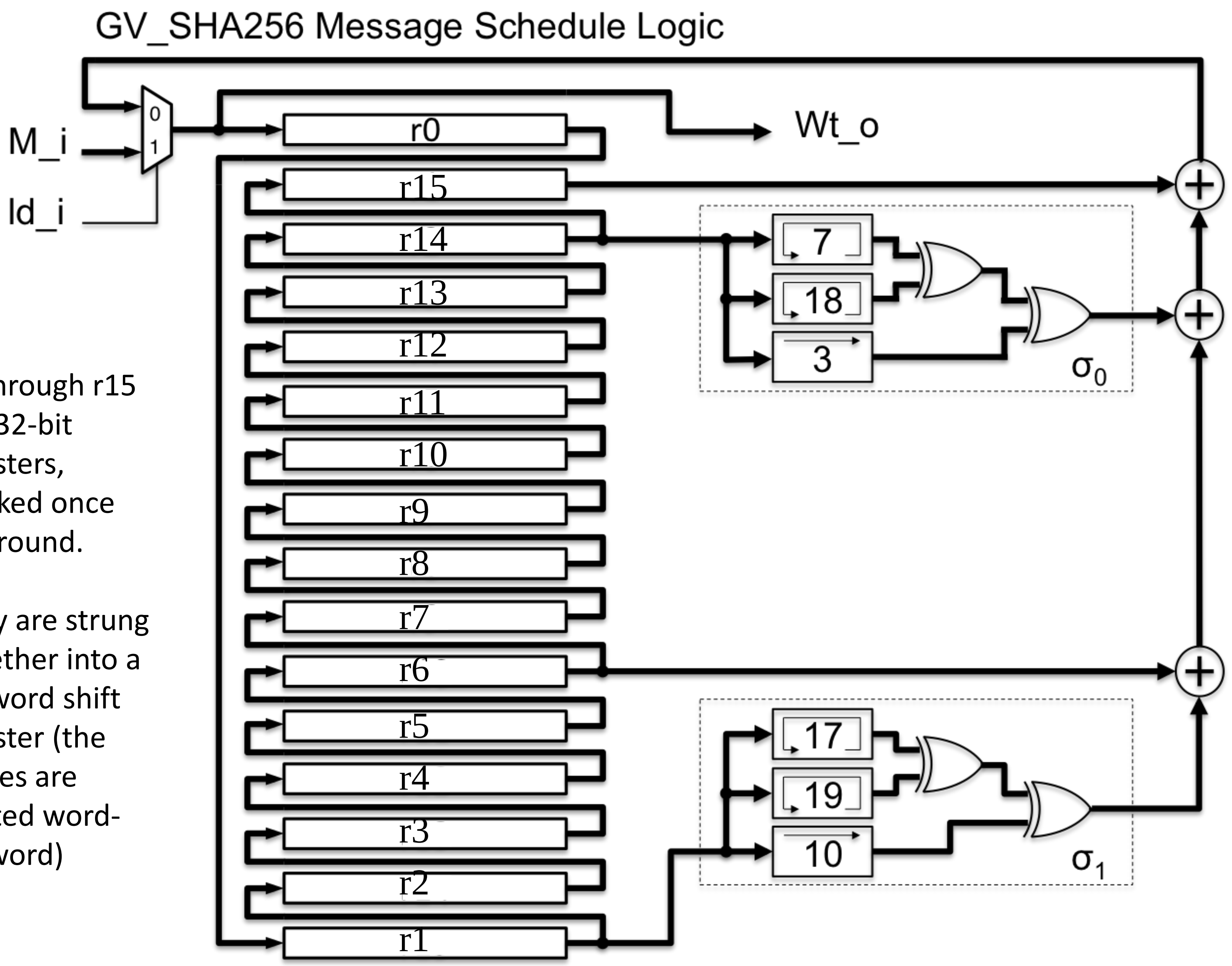
The **Wt_i** array of 64 32-bit values is called the ***Message Schedule***.

It is created by mixing up and transforming the 32-bit words in the message block as the hashing rounds proceed.
(*the purpose of all these manipulations is to make it very hard to undo the hashing process*).

As seen in the diagram on the next page, the first 16 words of ***Wt_i*** are set to the 16 words of the message block.

For ***i*** (*round_count)* from 0 to 15, the ***ld_i*** signal is set to 1, which just passes through ***M(i)*** to ***Wt_i***. For *i* from 16 to 63, ***ld_i*** *is* set to 0, which means that ***Wt_i*** is taken from the output of the last adder.

Modify your gNN_SHA256 vhdl code from lab 3 to implement the message schedule creation and combine it with the hash core logic.
It would be best to make another component to do the message schedule logic and instantiate it in the gNN_SHA256 entity.



r0 through r15 are 32-bit registers, clocked once per round.

They are strung together into a 32-word shift register (the values are shifted word-by-word)

Finally, there is one more modification you need to do to your Hash logic.

This is to add the initial Hash value to the final Hash value:

h0 <= h0 + A_o (*each addition is a 32-bit unsigned addition with no carry*)
h1 <= h1 + B_o
h2 <= h2 + C_o
h3 <= h3 + D_o
h4 <= h4 + E_o
h5 <= h5 + F_o
h6 <= h6 + G_o
h7 <= h7 + H_o

The final hash is then: (*change this in your gNN_SHA256 VHDL*)

```
    hash_out <= h0 & h1 & h2 & h3 & h4 & h5 & h6 & h7;
```

If there is more than one message block, these hash blocks are used as the initial state for processing the next block (but we will just limit ourselves to one block in this lab).

The second part of the lab is to define the overall system using Qsys.

This is fairly straightforward, but there are a lot of steps!

1. Copy the file *Altera_Cyclone_V_SOC_Development_Kit_HPS_Presets.qprs* (posted on myCourses along with this lab document) to the project directory
2. Start Quartus
3. Use the "New Project Wizard" to start up a new project. Call the project "gNN_SHA256_system". Don't add any files (in the Wizard). In the "Family, Device & Board Settings" window, click on the Board tab. Choose the DE1-SoC Board. Uncheck "Create top-level design file" (this will be defined later). Click "Finish".
4. Start Qsys from Quartus Tools menu
5. Under Qsys Tools/Options add the project directory to the IP Search Path. Click Finish. You shouldn't get any warnings and errors at this point.
6. From the Qsys IP Catalog, in the Library section, select the Processors and Peripherals/Hard Processor System/Arria V/Cyclone V Hard Processor System and click "Add".
7. In the window that pops up, select the Preset "*Altera_Cyclone_V_SOC_Development_Kit_HPS_Presets".* This will load in various settings for the hard processor device. Click "FINISH".
8. From the Qsys IP Catalog Library, select Processors and Peripherals/Peripherals/PIO (Parallel I/O) and click "Add". Set the bit-width to 10. Set direction to OUTPUT. This will be used to turn on/off the single element LEDs on the board. Rename the component to "LEDS".
9. Add another PIO component, but this time with direction set to INPUT. Set the bit width to 10. Rename to "SWITCHES". This will be used to read the state of the slide switches on the board.
10. Add another PIO component, with bit-width 32 and direction OUTPUT. Rename to HEX3-HEX0. This will be used to control the four rightmost of the six 7-segment LEDs on the board.
11. Add another PIO component, with bit-width 16 and direction OUTPUT. Rename to HEX5-HEX4. This will be used to control the two leftmost of the six 7-segment LEDs on the board.

12. Add another PIO component, but this time with direction set to INPUT. Set the bit width to 4. Check the “Synchronously Capture” box, and set “Edge Type” to FALLING. Check the “Generate IRQ” box and set IRQ Type to EDGE. In Qsys rename to “PUSHBUTTONS”. This will be used to read the state of the pushbuttons on the board.
13. In the Systems Contents pane of the Qsys window, make the connections between the components. Connect all of the clk inputs of the added components to the clk clock output signal of the clk_0 component by clicking on the appropriate bubbles in the Connections column. Also connect this clk to the h2f_axi_clock, f2h_axi_clock and h2f_lw_axi_clock inputs.
14. Connect the clk_reset signal of the clk_0 component to the f2h_cold_reset_req, f2h_debug_reset_req, and f2h_warm_reset_req of the hps_0 component and to the rest inputs of the other added components. Do the same with the h2f_reset output of the hps_0 component (just to the other component reset inputs). This will OR the two reset signals together.
15. Connect the irq signal of the PUSHBUTTONS component and the SYSTEM_CONSOLE component to the f2h_irq0 of the hps_0 component.
16. Connect the Avalon Memory Mapped Slave (bus) signals (these are labeled s1) of all components to the h2f_lw_axi_master on the hps_0 component.
17. Double click on the “Double-click-to-export” text in the “Export” column for the port named “external_connection” for the LEDS component. Type in “rled”. This tells Qsys that this component will have some signals that will be connected externally (in this case to the red LEDs on the board). Similarly, export an external connection for the SWITCHES, HEX3_HEX0, HEX5_HEX4, and PUSHBUTTONS components (name these “switches, hex3_hex0, hex5_hex4, and pushbuttons, respectively).

18. In the IRQ column, change the PUSHBTTONS IRQ to 1.
19. Notice the “Base” column, these entries show the base addresses of the Avalon interface for each component. Currently they are all set to the same value, 0x0000_0000. We need to make them different. To do this, select System/Assign Base Addresses from the Qsys menu bar.
20. There should be no more error messages in the Qsys Message pane. There may be some warnings.
21. Save the Qsys settings file as “gNN_SHA256_system.qsys”. Close the save system window when it finishes saving.
22. Click on “Generate HDL”. In the “Create HDL design files for synthesis” box select “VHDL”. Uncheck the “Create timing…” box and select “None” for Create simulation model:, since we will not be doing any simulation. The Output Directory Path should be your project directory. Click on “Generate”. This will produce the VHDL file which basically instantiates the components specified in your Qsys system.
23. Exit Qsys.

24. In Quartus, select Project/Add/Remove Files in Project. Then add the *.pip file in the directory qsys_lab/synthesis (which was created when you ran Generate in Qsys).
25. In Quartus, open the file /synthesis/gNN_SHA256_system.vhd (which should have been generated by Qsys). In the /Project menu tab, first select “Add Current File to Project”, then select “Set as Top-Level Entity”.
26. We will need to map the FPGA pins to the DE1-SoC board connections. Before doing this you should run the Analysis and Synthesis (under Processing/Start/Start Analysis and Synthesis/). This will determine what pins are unassigned and need to be mapped.
27. When Qsys generates any HPS component, Qsys also generates the pin assignment TCL Script File (.tcl) to perform pin assignments for the memory connected to the HPS. The script file name and location is at: ../qsys_lab/synthesis/submodules/hps_sdram_p0_pin_assignments.tcl
28. Run this script, by selecting Tools > Tcl Scripts... Once the script has run, we can assign the rest of the standard DE1-SOC pins.
29. To assign the rest of the pins, select Assignments/Import Assignments and choose the file qsys_tutorial.qsf to load in some pre-defined assignments (otherwise it would quite tedious to enter them all yourself using the Pin Planner). Note that in this assignment, KEY(0) (the rightmost pushbutton) is mapped to a reset signal, and thus will not be accessible by the HPS via the Avalon bus (i.e. the signal PUSHBUTTONS[0] is not connected to anything).
30. Re-compile the design using Quartus. There should be no errors (may be some warnings).

31. Next, you will use Qsys to add a new component of your own design. This will also be connected to the Avalon bus and communicate with the HPS via the light-weight bridge (h2f_lw_axi).
32. In Quartus, create a new VHDL component called "gNN_SHA256_custom_component.vhd" (where NN is your group number). This file will just contain a component instantiation for your gNN_SHA256 design entity from lab 3 (with some changes). We do it this way so that any changes will just be done to the second file. Since the first file will not be changing we will not require any re-generations with Qsys (after the initial one to tell it about the new component). The gNN_SHA256_custom_component design entity should provide the necessary signals for connecting as a slave to an Avalon memory-mapped master. As such, it needs the following port signals:
    - clock (clock signal)
    - resetn (active low reset)
    - address (8 bit address to be used internally by the component to direct data)
    - readdata (32 bit data read from the component)
    - writedata (32 bit data to be sent to the component)
    - read (active when a read transaction is to be performed)
    - write (active when a write transaction is to be performed)
    - chipselect (active when a transaction is being performed)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY gNN_SHA256_custom_component IS
PORT (
   clock, resetn : IN STD_LOGIC;
   read, write, chipselect : IN STD_LOGIC;
   address : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
   writedata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
   readdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
   );
END gNN_SHA256_custom_component;

ARCHITECTURE Structure OF gNN_SHA256_custom_component IS
SIGNAL to_component, from_component : STD_LOGIC_VECTOR(31 DOWNTO 0);

COMPONENT gNN_SHA256 - from lab 3 with changes to port map
PORT ( clock, resetn : IN STD_LOGIC;
   read, write, chipselect : IN STD_LOGIC;
   in_data : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
   out_data : OUT STD_LOGIC_VECTOR(31 DOWNTO 0) );
END COMPONENT;

BEGIN
to_component <= writedata;
component_inst: gNN_SHA256 PORT MAP (clock, resetn, read, write,
chipselect, address, to_component, from_component);
readdata <= from_component;
END Structure;
```

You will need to edit the VHDL code for gNN_SHA256 from lab 3.

First, change the ports in the entity declaration to include the Avalon port signals:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY gNN_SHA256 IS
PORT ( clock, resetn : IN STD_LOGIC;
   read, write, chipselect : IN STD_LOGIC;
   address : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
   in_data : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
   out_data : OUT STD_LOGIC_VECTOR(31 DOWNTO 0) );
END gNN_SHA256;
```

Then, you need to add in the code to handle the Avalon transfers. Remember, these are used to fill the message block array, ***M***. Use a different address for each word in the ***M*** array. *You can use a write to some other address to initiate a hash operation* (we have 8 address bits so 256 different addresses available).

You can look at the example VHDL code for the Mandelbrot fractal accelerator to see how to handle Slave writes. Reads are similar (for the Master to read the final Hash value).

33. Start Qsys again, and load the qsys_lab.qsys file.
34. In the IP Catalog select “New Component” and click on “Add”. An information window will appear, in which you will provide details about the new component. Give it the name “gNN_SHA256_custom_component” with Display name “custom_component”. Leave the Group blank. For the description enter “gNN_SHA256 custom component”. Enter your group student names in the Create by tab.
35. Select the “Files” tab in the Component Editor window. In the “Synthesis Files” section click on “Add File”. Select the top-level VHDL file you just created (gNN_SHA256_custom_component.vhd). Then click on “Analyze Synthesis Files”. This will look at the design entity descriptions to see what the input and output ports are and match them to Avalon interface signals. (if you get a “Info: Error: No modules found when analyzing null.” message then you probably have a typo or syntax error in your VHDL file).
36. There will be some error messages in the Messages pane. This is because we still have some work to do!
37. Click on the “Signals & Interfaces” tab at the top of the Component Editor window. This shows the and interfaces in your top level component, and what Qsys thinks they are for relative to the Avalon bus interface. Most of these guesses are wrong, hence the errors in the Message window. Under the clock[1] signal, click on *<<add interface>>*. Select “new Clock Input”. The entry will change to “clock_sink”. Drag the clock[1] signal to move it under the clock_sink interface. In the right hand part of the window, change Signal Type by selecting “clk”. Drag the resetn signal from the Avalon_slave_0 interface to be under the “clock_reset” interface. Change the Signal Type to “reset_n”. The other signal settings will probably be correct and don’t need to be changed.
38. There will still be a few remaining error messages. On the left, click on “avalon_slave_0”, and under “Associated Clock” select “clock_sink” and under “Associated Reset” select “clock_reset”. Finally, for “reset_sink” on the left side, select “clock_sink” for the “Associated Clock”. Finally, click on the “clock_reset” interface. Then set its Associated Clock to “clock_sink”. All the error messages should now be gone!
39. Click Finish and choose “Yes, Save” to save the qsys related component description files. The new component will now show up in the IP Catalog, in the Project group.
40. We can now add the new component to our Qsys system, just like we did with the other components earlier. Connects its ports just as you did for the PUSHBUTTONS component.
41. The address map will now overlap the other component addresses, so we have to redo the /System/Assign Base Addresses command. The addresses may have changed for the components, so you should make sure to check what the new addresses are so that you can change your c-code if needed. Note the memory address range for the new component. It actually has a 10-bit address range (0000-03FF) as the address indicates 8-bit words, whereas the data width is 32-bits.
42. Re-run the Generate HDL in Qsys. Take a screenshot for your report, then you can exit Qsys.

Finally, go back to Quartus and do a full compilation of the top-level design (which should be gNN_SHA256_system).

Before doing the full compile, set the target clock frequency in the sdc file to something relatively slow, say 10MHz.

After compilation, check the timing report to see what the maximum clock frequency is. Change the target clock frequency in the sdc file and re-compile.

Also check the compilation flow summary to see the FPGA resource usage. How many copies of the system do you think you could fit into the FPGA?

Unfortunately, there is no way to simulate the running of a program on the ARM processor with connection to the FPGA, and because of the pandemic you will not be able to download your complete system to the DE1-SoC development board.

But you can do a simulation of the *gNN_SHA256* circuit (not *gNN_SHA256_system*), creating a VHDL Testbench to send the 16 words that make up one message block, and then run the *gNN_SHA256* for one complete hashing cycle (of 64 rounds).

As a test, try to simulate the example shown in:
https://qvault.io/cryptography/how-sha-2-works-step-by-step-sha-256/
which goes through the hashing of “hello world” step-by-step.

If you can get to the point where you can simulate your entire system and replicate the results shown in that example, then kudos to you!

If not, hopefully you have gained some understanding of what's involved in doing cryptographic operations such as underly today's blockchain technology, as well as digital system design techniques.

And no matter how far you got in this lab, you are all to be congratulated for hanging in there during a very difficult time.

Write up the Lab Report .

Write up a short report describing the ***gNN_SHA256_system*** circuit that you designed in this lab. This report should be submitted in pdf format.

The report must include the following items:

- A header listing the group number, the names and student numbers of each group member.
- A title, giving the name (i.e., ***gNN_SHA256_system***) of the circuit.
- A description of the circuit's function, listing the inputs and outputs.
- Also provide the following:
    - A screenshot of the final Qsys system contents pane.
    - A screenshot of the Flow Summary showing the FPGA resource usage.
    - Static timing analysis report, showing maximum clock frequency.
    - Functional simulation results for the test case.
- Discussion of the any design decisions made and problems faced during the design.

Submit the Lab Report to myCourses .

The lab report, and all associated design files (.vhd, .vht, .qsf and .sdc files) must be submitted, as an assignment to the myCourses site.

Only one submission need be made per group (all students in the group will receive the same grade).

Combine all of the files that you are submitting into one *zip* file and name the zip file gNN_Lab_Project.zip (where NN is your group number).

The report is due on Tuesday April 20, at 11:59 PM.

There will be no extensions granted, as the course marks have to be submitted by April 23.