CprE 381 – Computer Organization and Assembly Level Programming

Lab #1

[Note: The goal of Lab #1 is to give you some initial familiarity with the ModelSim simulation environment, simple VHDL designs, and proper hardware design and test methodology. During subsequent labs you will continue to develop your hardware design skills and will have increasing freedom and responsibility for your work.

Throughout this lab, you will find it useful to have a VHDL reference. The detailed Mealy & Tappero Free Range VHDL *tutorial/eBook is provided on the CprE 381 website (under labs folder).* Free Range VHDL *is a relatively easy-to-read introduction to VHDL that emphasizes proper hardware design thinking. It does not aim to be a comprehensive guide to VHDL syntax, etc. The lab manual references specific sections that are particularly useful for the various parts of this lab. For a more complete description of standard VHDL syntax, I have found the archive of VDLANDE.com (https://www.ics.uci.edu/~jmoorkan/vhdlref/) to be very useful. If you are not used to reading language documentation this may be less accessible, but I would suggest becoming more familiar with this sort of documentation (especially for those of you who are software-oriented). Consulting the lengthy ModelSim User's Guide will probably not be necessary.]*

0. **Prelab.** Complete and submit the "Teaching Laboratory Safety Procedures" agreement. Ensure that you have access to ECpE VDI (vdi.engineering.iastate.edu using http://etg.ece.iastate.edu/vdi/connecting/). Watch the "Lab1 Intro" and "Design Cycle – Overview" videos.

1. **Design Cycle.** In order to practice the 381 design cycle, walk through the design cycle for a multiply and accumulate (MAC) unit. Note the tools and processes you use – you will continue to use these in each of the following labs. In particular, you should copy the use of testbenches and "do" files to perform efficient and thorough testing of each module you create.

    a. **Describe Design.** We are designing a pipelined multiply and accumulate (MAC) unit with included weight register for use in a TPU-like systolic array. *[If you want to know more about the context of the design, take a look at https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu. However, the actual use and context of the design is not critical to your understanding of the hardware design process.]* The design should be able to load a weight value. Once a weight value is loaded, each cycle a pair of inputs can be applied. The first is an activation to be multiplied by the weight with the result added to the second input which is a partial sum. After two cycles, the final result is output along with the activation. *[Such descriptions can be used in header comments in design files.]*

    b. **Define Ports.** Our MAC unit will need a weight data input, *iW*, and an associated weight load control signal, *iLdW*. It will also need a partial sum input, *iY*, and an activation input, *iX*. It will have two outputs, the MAC, *oY*, and the activation, *oX*. All numeric values are integers.

    c. **Example Inputs/Outputs.** We will explore a couple of specific examples of inputs and expected outputs that show the basic functionality of our design. While agreeing on the expected behavior of the design is very helpful for performing the detailed design work, this step conveniently

enumerates test cases (i.e., the *start* of a more complete test plan). We will describe each example case in plain English *[Although you may not understand what weight or activation or partial sum means, you should understand what underlying hardware behavior is being described.]*:

Case 1: Initialize weight

- Initialize a weight value register by setting *iW* to the desired value of 10 and *iLdW* to 1 prior to a positive edge of the clock and holding them until after the positive edge of the clock.
- I expect the intermediate internal signal, *s_W*, to take on my new value of 10 at the positive edge of the clock. The value of *s_W* prior to the clock edge may be anything since this is an initialization. I expect that *s_W* will not change unless I assert *iLdW* to 1 across a positive edge of the clock while *iW* is a different value.

Case 2: Perform one MAC operation with average-case values

- With the above weight initialization to 10, I set the activation input, *iX*, to 3 and the partial sum input, *iY*, to 25. I make sure the weight is not changed by setting *iLdW* to 0.
- I expect that, after two positive edges of the clock (i.e., two cycles), the activation output, *oX*, will be 3 and the partial sum output, *oY* will be 55=10*3+25.

Think of three more cases and record them in your lab report. *[Note that just three more test cases would not be sufficient to comprehensively test this design, but are used to help you get started thinking about the design functionality. For testing, I would start to think about what are the normal or average cases you expect? what are some edge cases? how do you expect the design to perform under those edge cases?]* One of your cases must result in your *oY* value being the same as the day of your birth. Describe these cases in your lab report using the above format.

d. **Draw Design.** We've included a drawing of our MAC unit at the end of the lab. We started with the inputs and outputs and then used several subcomponents to implement the high-level MAC unit (TPU_MV_Element). We then recursively designed each of the four subcomponents. Deciding where to break up a larger component is a challenging task for hardware designers and is part of the "art" of hardware design. We will provide some guidance throughout the semester on how to decompose your design into smaller components, but in general you should endeavor to make your designs easy to reason about, easy to test, and improve code *reuse [If you are copying the same complex code multiple times you should probably be using some form of modularity.]*

e. **VHDL Design.** Reference the circuit diagram at the end of this document (some parts simplified). There are 33 labeled areas in the diagram. For labels 1, 7, 22, and 28, specify where (VHDL file and line number) these values are located – some will be found in more than one place. Also attempt to explain the functionality of each label as it occurs in the code. *[Please format your responses similar to the following example: in the attached diagram area, (17) is the oC output of the multiplier module, which is defined as an integer on Line 31 of Multiplier.vhd and set on Line 41. For this specific instance of the multiplier module, the output oC is connected to a signal called s_WxX on line 105 of TPU_MV_Element.vhd.]*

f. **VHDL Testbench.** You've also been provided with a VHDL testbench and associated "do" file. It contains the instantiation of the design under test (DUT) and a process statement that includes sequential statements assigning input values at specific times. *[Note that, while process statements are useful for experienced VHDL programmers to conveniently describe intended hardware behavior, they are very easy to misuse.* ***Therefore, for this course, process statements should only be used in testbenches and by provided primitive components.****]*

g. **Modelsim Simulation.** We will simulate the MAC unit design using ModelSim.

i. First, create a new folder under the X drive (your home directory) titled "cpre381" and inside of this folder create another folder titled "Lab1". Extract the "Lab1.zip", which can be downloaded from the class webpage, in this location. In the newly created "cpre381" folder, right-click and click "Open in Terminal." Inside the terminal window, type "modelsim" and hit enter. The Modelsim GUI will now open up (note it is branded as "Questa Sim-64" at the top).

Changing the Directory:
Select "File -> Change Directory" and navigate to the Lab1 folder created before. Alternatively, use "pwd", "cd", and "ls" commands in the "Transcript" pane to navigate to the Lab1 folder you created.

To open the files select "File -> Open" or click on the open icon below the file drop down menu.

ii. Compile *Adder.vhd* into library "work". This can be done through the GUI (you'll have to find and select the file again) select "Compile -> Compile", or "Compile -> Compile Selected" if you created a project in the first step. Alternatively, use the "vcom -2008 -work work *.vhd" command in the transcript pane (see "vcom -help" for further usage information).

If a prompt appears asking about creating the library "work", select "Yes" to create this library.

Look in the "Transcript" pane to determine if the compilation was successful and if not, what the errors were (double-clicking on the red text should pop-up more details).

Repeat this compilation process for the other provided files listed above. *[There should be at least one error in the provided files. Before proceeding, read and address all warnings and errors* ***in order****.]*

iii. Simulate the design by selecting "Simulate -> Start Simulation", selecting the *work.tb_tpu_mv_element* design. You must then click "Optimization Options…" and select "Apply full visibility to all modules (full debug mode)" under "Visibility". Finally, click "OK" twice. Alternatively, use the "vsim -voptargs=+acc tb_tpu_mv_element" command in the transcript pane (see "vsim -help" for further usage guidance).

iv. The next screen that comes up will vary a bit based on your particular ModelSim installation and preferences, but typically you will have a "sim" pane with the instances of modules within your design. In this case your root is the testbench module, *tb_tpu_mv_element*. This pane is hierarchical, so you will be able to see the subcomponents such as *DUT0* and its subcomponents (click on the plus sign by *DUT0*). For whichever instance you have selected, you will be able to see the signals (e.g., *CLK*, *s_iX*, and *s_oY* are visible when you have *tb_tpu_mv_element*

selected) and constants in the "Objects" pane and the processes in the "Processes" pane (e.g., *P_TEST_CASES* is visible when you have *tb_tpu_mv_element* selected). You may make any of these signals visible in the simulation waveform by selecting them and choosing "Add -> Wave -> Selected Signals" to view the (empty) simulation waveforms. In the transcript pane, adding waveforms for all of the signals for the testbench module can be accomplished using the following command:

```
add wave sim:/tb_tpu_mv_element/*
```

v. Now we're finally ready to simulate! Select "Simulate -> Run -> Run 100 ns" to run the design for 100 nanoseconds. This isn't actually enough to see the result in the waveform clearly, so repeat this a few more times. In the future, you can simulate an arbitrary amount of time using the following command in the transcript:

```
run X
```

Where X is the amount of time you'd like to simulate for (default is in nanoseconds). *[Note: your ModelSim simulation time-step resolution may be set to picoseconds (ps) by default. You can change it to ns to follow the instructions literally, but either resolution will work as long as you are aware of it.]*

In your lab report, include a screenshot of the waveform. Describe, in plain English, any differences between what you expected and what the simulation showed.

vi. To exit the simulation to edit and compile the source files (e.g., for debugging), type the following in the transcript:

```
quit -sim
```

h. **Debug.** If you used the files provided to you, you must have encountered several issues throughout the process. If you haven't already done so, fix them. In general, you should not have any compile, load, or simulate warnings. While there are some benign warnings (e.g., warnings about the number of processor cores used by the CAD software), most warnings are a sign of an error in your design or your description of your design. Address them before moving on since, throughout the semester, you will design components that are used repeatedly in more complex designs. If you do not fix design issues early, they may manifest as indecipherable, hard-to-track-down errors in the more complex designs.

Once you have eliminated all errors and warnings, you will find that the provided VHDL files still do not function as expected. Using what you've already seen of VHDL as well as the specific results of simulation, edit the design files to produce the expected results. In your lab report, include a screenshot of the waveform. Describe, in plain English, how your waveform matches the expected result (e.g., reference the specific cycles and times). In your submission zip file, provide the completed *TPU_MV_Element.vhd* file in a folder called 'MAC'.

In order to verify that your updated design works correctly, you will have to perform at least one more simulation (and possibly several more). As you begin to simulate more complicated designs

and have to perform multiple simulates to debug it, you may begin to be annoyed by having to repeatedly add signals to a waveform and format them to display in your desired format (e.g., hex). Good. This is tedious and, worse, error-prone. These tasks are prime candidates for automation in the form of a "do" file. We've included an example do file, *tb_TPU_MV_Element.do*, which adds and formats signals for our design and then runs the simulation. Take a moment to read this text file. We've annotated the file with comments (all text on a line following the '#' character are comments). You can run a do file once you've started simulation by entering the following command into the transcript pane:

```
do tb_TPU_MV_Element.do
```

Note that a do file can reproduce almost any action you take in the simulation GUI. There are two ways to learn the corresponding commands. One is to perform an action in the GUI and see the resulting command or commands in the transcript pane. Copy the commands into your .do text file and save it. Alternatively, for waveform commands that may not appear in the transcript (yes, this is annoying), you can save the waveform configuration as a do file by clicking "File -> Save Format" and clicking "Ok". Then you can look inside of wave.do and try to pick out the relevant commands or command arguments. This can be tricky and takes some up-front time, but can pay off throughout the semester in significantly reducing your debug cycle time.

i. **Getting Debug Help.** One of the most challenging (and time-consuming) parts of your 381 lab experience will be debugging your design. It may feel like you are on your own, but you are NOT! Your lab mates, TAs, and instructors are all available to help. To access that help, you have to be able to accurately describe the problem you are struggling with. Enter the "Bug Report":

1. What was the system setup?
2. What are the inputs (files, signal values, etc.)?
3. What were the expected results (print outputs, signal/waveform behavior, etc.)?
4. What were the actual observed results (print outputs, signal/waveform behavior, etc.)?
   * Include screen captures of waveforms ("Applications -> Utililies -> Screenshot"). Note: Make sure to capture only the areas relevant to the waveform (i.e. don't include extraneous information by capturing the whole window).
   * Include log transcript as a file (specify how to for specific parts of the process)
   * Include code or minimum working example? (this may be too much)
5. What, if any, errors exist?
6. What, if any, warnings exist? Why are they ok to ignore?
7. Any other behavior you see that you think may be relevant
8. Attempts made to solve/debug this problem -- you should have tried at least one, and include an explanation of your thought process

Filling out the bug report forces you to concretely and comprehensively specify the unexpected behavior you are seeing and hypothesize about what may have caused the bug.

While filling out a bug report will let you access a wealth of collective knowledge, it also forces you to systematically examine the conditions that caused your unexpected behavior. Often times going through the process of filling out the template can cause you to identify and solve the bug on your own (e.g., identifying implicit assumptions, missed warnings, errors in VHDL description of your drawing). It also helps you to get a more complete understanding of your design.

If you get stuck on one of the above bugs or any other bug throughout the semester, write up bug report and post to *your* lab section channel. As you are working through the labs, read the bug reports others post, ask questions about their bugs, and provide suggestions to others within your lab section.

*[Phew, you've already learned a lot! Take a little mental break. You're about to dive into designing your own units using the process you've just learned. This should reinforce the process and the tools skills you've learned. To help you manage your time, I would expect you to have gotten here roughly by the end of your first lab section or shortly afterwards. I expect that you would have already started on the following designs **before** you set foot in the section lab section! Note that this means you have worked on the lab outside of the lab section during the first week.*

As you embark on your first VHDL designs, it would be helpful for you to learn a little bit more about VHDL. In particular, you should read at least Chapter 9 in Free Range VHDL and go back and reference any other sections you may need to understand Chapter 9. This chapter covers how to describe a design in structural modeling –instantiating modules and connecting input signals and output signals. You've already seen a bunch of examples of this modeling approach in the testbench and high-level design of the MAC unit. This approach is most analogous to the physical hardware you are describing and as such is a good place to begin. However, it can be tedious and error-prone to describe all of your hardware from gate components up, so there are other modeling approaches that one can use. Specifically, the dataflow approach described in Chapter 4 offers more convenient modeling approaches.

Another modeling approach, behavioral modeling with process statements, may appear very familiar to programmers in imperative languages (e.g., C, Java, and Python). However, this advanced modeling approach requires a very strong understanding of hardware design intent and a disciplined hand. Reasoning about sequential statements interacting with concurrent statements has befouled many a student project. If you don't believe me, take a look at Chapter 5, especially 5.5. As new VHDL users, do not use process statements outside of testbenches and the provided dffg.vhd file.]

2. **Debug Report.** Throughout this portion of the assignment, you must post at least one bug report to your lab section channel. If you are flying through this without having any bugs, that's great (and pretty rare), but practice filling out a bug report by coming up with a challenging bug and posting it for others to think about. Additionally, you must make at least one substantive comment on a lab mate's bug report. Finally, once you solve the bug, post which solution work to your conversation.

3. **A First Design.** For your first VHDL design, we will revisit our familiar friend, the Two-Input Multiplexor (or 2:1 mux). A 2:1 mux takes two bits as input (call them i_D0 and i_D1), and forwards one of them to its output (call this o_O) based on the value of a control signal (call this i_S). Provide your solution to this problem in a folder called 'Mux/'.

    a. Draw the truth table, Boolean equation, and Boolean circuit equivalent (using only two-input gates) that implements a 2:1 mux. Include this in your lab report.

    b. Implement a 2:1 mux using ***structural*** VHDL (name the entity mux2t1). You may use any or all of *invg.vhd* (NOT gate design), *andg2.vhd* (two-input AND gate design), *org2.vhd* (two-input OR gate design), and *xorg2.vhd* (two-input XOR gate design) as the basic building blocks. If your solution to part a uses other two-input Boolean gates, then feel free to design and use those structural components as well. *[I suggest you copy the general structure of the VHDL files you've been provided, including the header comments. This is also true for the required testbench files.]*

    c. Implement an appropriate testbench for this component. Note that for such a small combinational module, you can exhaust all test cases you enumerated in part a.

d. In your lab report, include a screenshot of the waveform. Make sure to label the screenshot with which module it is testing.

e. Now re-implement your 2:1 mux using ***dataflow*** VHDL. Don't use the AND, OR, and NOT operators, but rather a conditional signal assignment statement (see Section 4.4 in *Free Range VHDL*). Note that testing is identical to the structural mux. However, I would suggest using a different testbench and different entity name in order to avoid name ambiguity in the ModelSim library. Again, in your lab report, include a labeled screenshot of the waveform showing the dataflow mux implementation working.

4. **Generics.** Look at the provided N-bit 2:1 module (*mux2t1_N.vhd*). This module structurally instantiates an N-bit 2:1 mux from the 2:1 mux you've already designed using a generate for statement. Note how it defines a generic port. Create a testbench for this module and confirm that it works as you expect. For further help on generics you can refer back to Section 9.2 of *Free Range VHDL*. Include a waveform screenshot and corresponding description demonstrating it is working correctly.

5. **One's Complementor.** A One's Complementor unit is a combinational functional unit that takes a single input and negates each individual bit. Provide your solution to this problem (i.e., VHDL code) in a folder called 'OnesComp/'.

   a. Implement an N-bit one's complementor using structural VHDL, with the included *invg.vhd* NOT gate design as the basic building block.

   b. Create a single testbench design that uses structural VHDL to test your design. Use ModelSim to test that both designs give consistent results for the N=32 configuration. Include a waveform screenshot and description in your lab report.

6. **Adder.** An adder takes three inputs (two N-bit operands and one carry-in bit) and calculates both an N-bit sum and a single bit carry-out value. Provide your solution to this problem in a folder called 'Adder/'.

   a. A full adder takes three single-bit inputs and produces two single-bit outputs – a sum and carry for the addition of the three input bits. Draw the truth table, Boolean equation, and Boolean circuit equivalent (using only two-input gates) that implements a 1-bit full adder. Include this in your report.

   b. Implement this full adder using **structural** VHDL, with the included gate designs as the basic building blocks.

   c. Create an N-bit, ripple-carry version of this structural full adder design. Write at least three input and expect output cases. Then draw a schematic of the intended design, including inputs and outputs and at least the 0, 1, N-2, and N-1 stages. Include this in your report. *[This is slightly less trivial than the previous two designs due to carries across bit positions.]*

   d. Create a testbench for your N-bit adder using at least your above three test cases *[a more thorough testing now will prove useful when you use this design in later labs and, possibly, your term*

project]. Use ModelSim to test that both designs give consistent results for the N=32 configuration. Include an annotated waveform screenshot in your write-up.

7. **Adder-Subtractor.** An **Adder/Subtractor with Control** takes two values (A, B) as input, plus a control bit (nAdd_Sub), and calculates A+B when nAdd_Sub = '0', and A-B when nAdd_Sub = '1'. Provide your solution to this problem in a folder called 'AddSub/'.

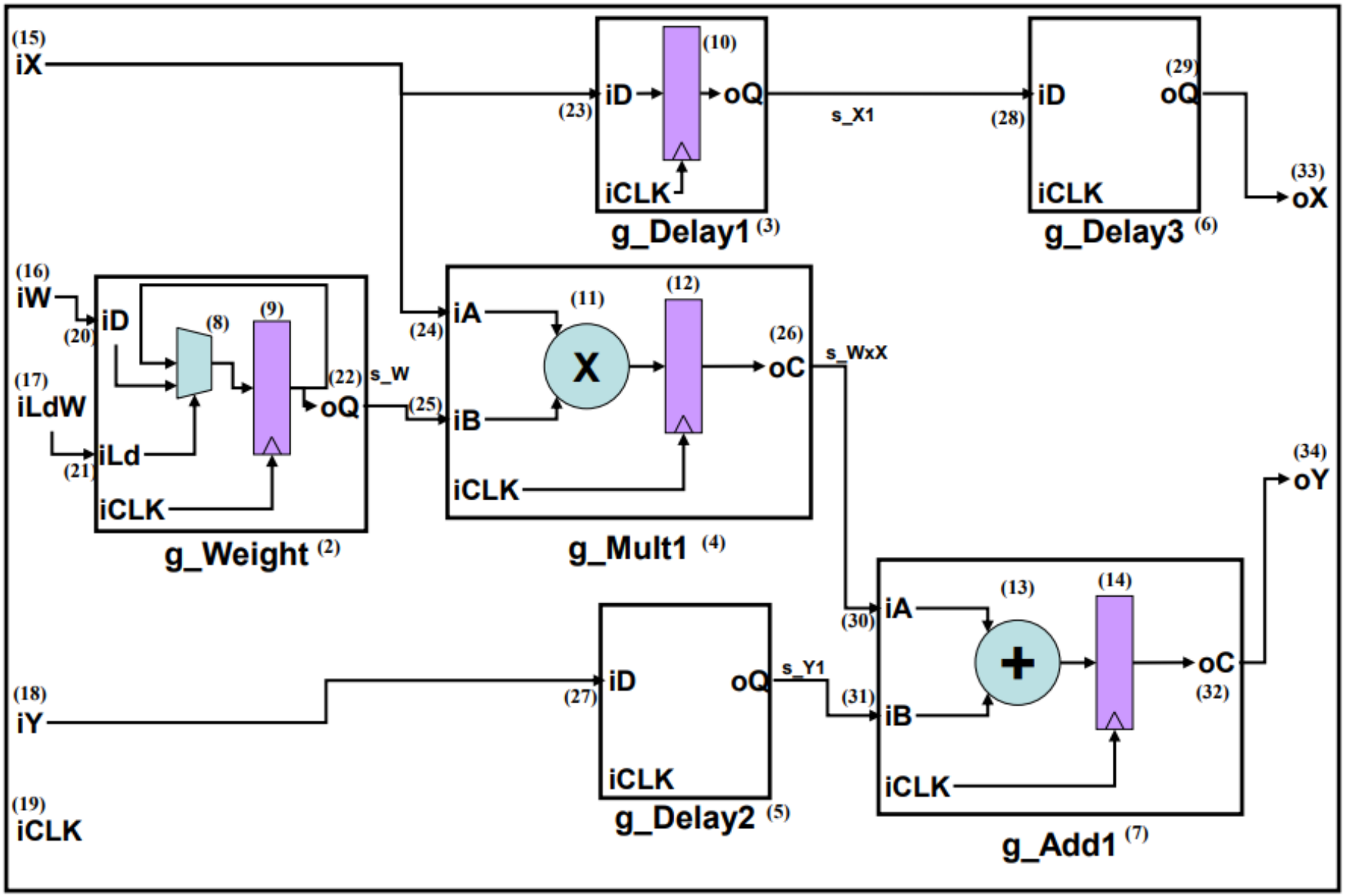
   a. Draw a schematic (don't use a schematic capture tool) showing how an N-bit adder/subtractor with control can be implemented using only the three main components designed in earlier parts of this lab (i.e., the N-bit inverter, N-bit 2:1 mux, and N-bit adder). How is the 'nAdd_Sub' bit used? Include this in your report.

   b. Implement this adder/subtractor with control using **structural** VHDL.

   c. Thoroughly test this design for correctness. Provide multiple waveform screenshots in your write-up to confirm that this component is working correctly. What test-cases did you include and why?

SUBMISSION:

- Complete the lab report (using template *Lab1_report.doc* in *Lab1.zip*).
- Create a zip file *Lab1_submit.zip*, including the following completed code and documents
  1. TPU files folder.
  2. MUX files folder.
  3. OnesComp files folder.
  4. Adder files folder.
  5. AddSub files folder
  6. Lab report (*Lab1_report.doc*)
- The file names in your zip file should be self-explained.
- Submit the zip file on Canvas under "Lab1: Orientation" assignment. **Even though the report should be included in the zip file, you must *also* submit it as a pdf separately from the zip on Canvas.**


(15)
iX
(23)
iD
(10)
oQ
iCLK
g_Delay1 (3)
s_X1
(28)
iD
(29)
oQ
iCLK
g_Delay3 (6)
(33)
oX
(16)
iW
(20)
(17)
iLdW
(21)
iD
(8)
(9)
(22)
oQ
iLd
iCLK
g_Weight (2)
s_W
(24)
(25)
iA
(11)
(12)
X
(26)
oC
iB
iCLK
g_Mult1 (4)
s_WxX
(34)
oY
(30)
iA
(13)
(14)
+
oC
(32)
(31)
iB
iCLK
g_Add1 (7)
(18)
iY
(27)
iD
oQ
s_Y1
iCLK
g_Delay2 (5)
(19)
iCLK
TPU_MV_Element (1)