ECE337 Lab 4:
Introduction to FSM Controllers in Verilog

In this lab you will:

- Implement and test the functionality of the Mealy and Moore '1101' detector Finite State Machines originally discussed in the postlab of lab 1.
- Design, code, and test the functionality of the source version of a Moore model state machine of a sliding window average filter.
- Synthesize, test, and verify the functionality of the mapped version of a Moore model state machine of sliding window average filter.
- Use Verilog file IO to feed a grayscale image through your sliding window averaging filter.

1. Lab Rules and Teamwork Report

For this lab you will be working with a partner. Please see the Teamwork Report document posted with this handout for details on what is and is not acceptable collaboration. You will also need to fill out a teamwork report after the lab as outlined by the Teamwork Report document.

2. Lab Setup

Create your Lab4 directory like you have for the other labs, a copy of the commands are below for your reference.

```
mkdir ~/ece337/Lab4
cd ~/ece337/Lab4
dirset
setup4
```

Also make sure to import your fresh lab directory into SVN and checkout a working version before proceeding with the lab. **Once you have checked out a working copy of your lab 4 directory, have a TA check off your work up to this point.**

3. Introduction to Finite State Machine Design

In order to help get you more comfortable with designing finite state machine Verilog code like what will be used in the controller for the 4-sample slide widow averager design you will be building in this lab, you will start out implementing and verify the '1101' detector designs from lab 1's postlab.

As a reminder the '1101' detector's purpose is to **assert a logic '1' output whenever the sequence "1101" is detected** in a serial input data stream. *For example, the sequence "11011011" will output a logic '1' TWICE.*

3.1.1. Moore Machine '1101' Detector Design

3.1.1.1. Moore Machine '1101' Detector Specifications

The required module name is: moore
The required filename is: moore.sv
The required test bench module name is: tb_moore
The required test bench filename is tb_moore.sv

The module must have the following ports (case-sensitive port names):

```
input  wire clk,
input  wire n_rst,
input  wire i,
output reg o
```

3.1.1.2. Simulating and Verifying the Moore Machine '1101' Detector

You will need to create your own test bench for simulating and verifying your Moore '1101' detector design. This test bench will be graded via code coverage of a known properly working design during the automated grading submission.

3.1.1.3. Automated Grading of the Moore Machine '1101' Detector

To submit your Moore '1101' detector design and test bench for grading, issue the following command at the terminal (can be from anywhere).

```
submit Lab4MO
```

3.1.2. Mealy Machine '1101' Detector Design

3.1.2.1. Mealy Machine '1101' Detector Specifications

The required module name is: mealy
The required filename is: mealy.sv
The required test bench module name is: tb_mealy
The required test bench filename is tb_mealy.sv

The module must have the following ports (case-sensitive port names):

```
input  wire clk,
input  wire n_rst,
input  wire i,
output reg o
```

3.1.2.2. Simulating and Verifying the Mealy Machine '1101' Detector

You will need to create your own test bench for simulating and verifying your Mealy '1101' detector design. This test bench will be graded via code coverage of a known properly working design during the automated grading submission.

3.1.2.3. Automated Grading of the Mealy Machine '1101' Detector

To submit your mealy '1101' detector design and test bench for grading, issue the following command at the terminal (can be from anywhere).

```
submit Lab4ME
```

4. Overall Design

For this lab you will be designing an ASIC that reads in data samples and outputs the average of the last four samples, as well as keeping track of the number of samples processed. The algorithm this design implements is called a sliding-window average, and is an example of a simple digital filter. Such a design would be useful for smoothing an A/D conversion to reduce noise, for example. The design architecture diagram of the system is given to you in Figure 1. Although the hardware described in this lab is not the most efficient way to solve this simple problem, it is very extensible and the function of the system could be easily changed to do other tasks with very little redesign.

4.1. Module Declaration

```
module avg_four
(
  input  wire clk,
  input  wire n_reset,
  input  wire [15:0] sample_data,
  input  wire data_ready,
  output wire one_k_samples,
  output wire modwait,
  output wire [15:0] avg_out,
  output wire err
);
```

4.2. Design Interfaces

The inputs and outputs of the top level are given here. (A more detailed description will be given with the description of the top level file itself.)

Table 1: Design Port Descriptions

Signal	Direction	Description
clk	Input	200MHz system clock signal with a 50% duty cycle.
n_reset	Input	Active low reset
sample_data [15:0]	Input	16-bit unsigned data input.
data_ready	Input	Active high signal that indicating when a sample is ready to be processed.
one_k_samples	Output	Each active high signal indicating that 1,000 samples have been processed since the last assertion/power on.
modwait	Output	Active high signal indicating that the design is busy
avg_out [15:0]	Output	The 16-bit unsigned average of the last four samples
err	Output	Active high signal indicating that an overflow occurred during averaging.

The **asynchronous** input data_ready must first be synchronized in order to be used by the system. When the synchronized signal, dr, goes high, it signals to the control unit that valid data is on the data signal and that the system should process a new sample. The control unit is responsible for telling the datapath block which operations to perform, in order, and where to store the values. It also pulses the cnt_up signal to increment the counter block and keep track of the number of samples processed. Additionally, the control unit must manage two top-level outputs: modwait, which tells the user that a sample is being processed; and err, which indicates an error in processing the samples. The datapath itself outputs the result of the operation, stored in a special output register, reg[0]. Before being presented to the user, the result (the summation of the last four samples) is divided by four.

4.3. Design Usage Constraints

- The total time to process one sample must not exceed 20 clock cycles (100 ns). Thus the outputs must be valid and the modwait signal must be deasserted within 20 clock cycles (100ns) of when data_ready is asserted.
- The modwait signal must be glitch free
- The modwait signal must always be high while processing a sample. It cannot be ‘0’ while any of the outputs of the design are not valid.

4.4. Design Architecture

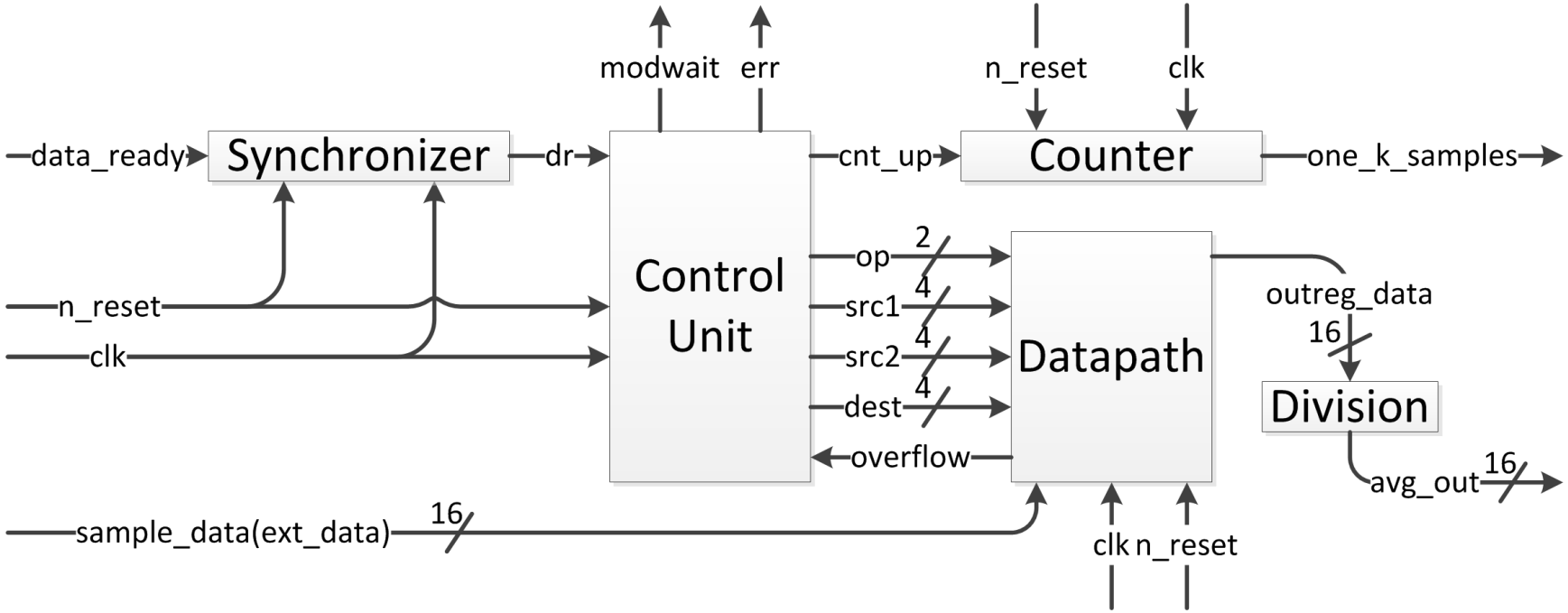

Figure 1: Sliding Window Average Filter Architecture

5. Detailed Design Specifications

5.1. Control Unit (controller.sv)

The control unit is the brain of your system. It has to regulate and control the operation sequence and input signals to the other components in the system so your system could operate as specified. A more in-depth description of the unit's operation is as follows: After receiving the dr signal, the new data should be stored in the register file, and the modwait signal should be raised. Then the data is reorganized and the oldest data thrown out, the data counter should be incremented, and any error signals should be de-asserted. Then the system will add the last four sampled data together (of course since you have only 1 adder, you have to do this in multiple cycles, therefore requiring you to use the register file as an accumulator). If at any point during the summation an overflow occurs, stop adding, assert the error signal, err, and continue on to the next step. (Cutting the addition short like this provides a small power savings.) If the dr signal goes low before the data is loaded the controller should stop processing it, assert the error signal, err, and not increment the counter. Once everything is done, modwait signal should be de-asserted to notify the external device that your system is done processing the data. Keep in mind that you need to store the final addition result in the output register in the register file, so that it will be displayed to via design output when the modwait signal is de-asserted. You may assume that the external device will know that **unless the modwait signal is low, the data in the output line might be invalid**. Also you may assume that the external device will attempt to send the design a new sample once the modwait signal transitions from high to low. This is why having a stable (i.e. no glitches) modwait signal is crucial. Signals "src1", "src2", and "dest" are the numbers or addresses of the registers that you would like to use as either sources (signals "src1" & "src2") of data for the operation or the destination (signal "dest") of its result. Section 9, which describes the datapath, discusses their use in more detail.

Hint: The easiest way to ensure that modwait is stable is to use a flip-flop. This is referred to as 'registering the output'.

5.1.1. Module Declaration

```
module controller
(
  input  wire clk,
  input  wire n_reset,
  input  wire dr,
  input  wire overflow,
  output reg cnt_up,
  output wire modwait,
  output reg [1:0] op,
  output reg [3:0] src1,
  output reg [3:0] src2,
  output reg [3:0] dest,
  output reg err
);
```

5.1.2. Port Descriptions

Table 2: Controller Block Port Descriptions

Signal	Direction	DESCRIPTION
clk	Input	200MHz system clock signal with a 50% duty cycle.
n_reset	Input	Asynchronous active low system reset signal. When this signal is asserted, all Flip-Flop values in the design are set immediately to logic-0.
dr	Input	This is the synchronized form of the data ready signal that signifies that the next sample data is ready.
overflow	Input	Indicates that an overflow occurred in the Datapath.
cnt_up	Output	This signal is the count enable for the counter. This signal should only be pulsed for 1 clock cycle (i.e. asserted for only 1 clock cycle and then cleared).
modwait	Output	This signal is to tell the external device connected to your design that the system is still processing the new data and the external device should wait. This signal should be stable (i.e. no glitches or edges unless during transition). To achieve stability, think about the state machine style or logic devices that you could use.
op [1:0]	Output	Op-code for the Datapath. See description of Datapath.
src1 [3:0]	Output	Operand for Datapath. See description of Datapath.
src2 [3:0]	Output	Operand for Datapath. See description of Datapath.
dest [3:0]	Output	Operand for Datapath. See description of Datapath.
err	Output	Error flag. Asserted when an overflow from an addition is detected, and de-asserted when the next data sample is read in.

5.2. Counter Unit (counter.sv)

The counter unit simply counts how many samples have been processed and asserts its output signal high after 1000 samples have been processed since the last assertion or power on. It output must be held high and stable until the next sample is being processed, and should be cleared before the processing of that sample has finished. The cnt_up signal tells the counter a sample has been processed. This signal is supplied to the counter by the controller.

Hint: Since the requirements are really just a special case of those for the flexible counter design you created and test in lab 3, this should simply be a wrapper file that uses that design, which is why that design file was copied into your Lab4 folder by the setup script.

5.2.1. Module Declaration

```
module counter
(
  input  wire clk,
  input  wire n_reset,
  input  wire cnt_up,
  output wire one_k_samples
);
```

5.2.2. Port Descriptions

Table 3: Counter Block Port Descriptions

Signal	Direction	Description
clk	Input	200MHz system clock signal with a 50% duty cycle.
n_reset	Input	Asynchronous active low system reset signal. When this signal is asserted, all Flip-Flop values in the design are set immediately to logic-0.
cnt_up	Input	This signal is the count enable for the counter. This signal should only be pulsed for 1 clock cycle (i.e. asserted for only 1 clock cycle and then cleared).
one_k_samples	Output	Indicates that 1,000 samples have been processed since the assertion/power on. Must be an active-high signal and stable while modwait is low.

5.3. Synchronizer (sync.sv)

The top-level input, data_ready, needs to be synchronized before it can be used. See the end of this handout for a discussion of how to accomplish this.

Hint: Since the requirements are identical to those for the synchronizer design you created and test in lab 2, this should simply be a copy of that design, which is why that design file was copied into your Lab4 folder by the setup script.

5.3.1. Module Declaration

```
module sync
(
  input  wire clk,
  input  wire n_reset,
  input  wire async_in,
  output wire sync_out
);
```

5.3.2. Port Descriptions

Table 4: Synchronizer Block Port Descriptions

Signal	Direction	Description
clk	Input	200MHz system clock signal with a 50% duty cycle.
n_reset	Input	Asynchronous active low system reset signal. When this signal is asserted, all Flip-Flop values in the design are set immediately to logic-0.
async_in	Input	The unsynchronized input.
sync_out	Output	The synchronized output.

5.4. Datapath (provided in Labs_IP Library)

This block has been provided to you in a library (Labs_IP). You can use objects from this library for both simulation and synthesis, but you do not have access to the source code. However, RTL and schematic level block diagrams have been provided for reference. A datapath is a term for the computational logic in a microprocessor. This datapath contains an ALU for arithmetic operations and a register file for storing data. There are 16 registers available for you to use, though you will not need them all for this design. **Register 0 is the output register, so any values assigned to this register will appear immediately on the output.** You will use this block to implement the arithmetic functions of your averaging filter. Signals “src1”, “src2”, and “dest” are the numbers or addresses of the registers that you would like to use as either sources (signals “src1” & “src2”) of data for the operation or the destination (signal “dest”) of its result.

5.4.1. Module Declaration

```
module datapath
(
  input  wire clk,
  input  wire n_reset,
  input  wire [1:0] op,
  input  wire [3:0] src1,
  input  wire [3:0] src2,
  input  wire [3:0] dest,
  input  wire [15:0] ext_data,
  output wire [15:0] outreg_data,
  output wire overflow
);
```

5.4.2. Port Descriptions

Table 5: Datapath Port Descriptions

SIGNAL	DIRECTION	DESCRIPTION
clk	Input	200MHz system clock signal with a 50% duty cycle.
n_reset	Input	Asynchronous active low system reset signal. When this signal is asserted, all Flip-Flop values in the design are set immediately to logic-0.
op<1:0>	Input	The operation to perform: 00 – NOP 01 – COPY: Copy from register 'src1' to register 'dest' 10 – LOAD: Store 'ext_data' in register' dest'. 11 – ADD: register 'dest' = register 'src1' + register 'src2'
src1<3:0>	Input	Source Operand #1 (register address/number)
src2<3:0>	Input	Source Operand #2 (register address/number)
dest<3:0>	Input	Destination Operand #3 (register address/number)
ext_data <15:0>	Input	Data to be loaded into a register (see op).
outreg_data <15:0>	Output	Value stored in the output register (register 0).
overflow	Output	Set if the current operation produces an overflow. (This signal is actually generated asynchronously and can be assumed to be valid whenever the input to op is ADD.)

5.4.3. Datapath Architecture

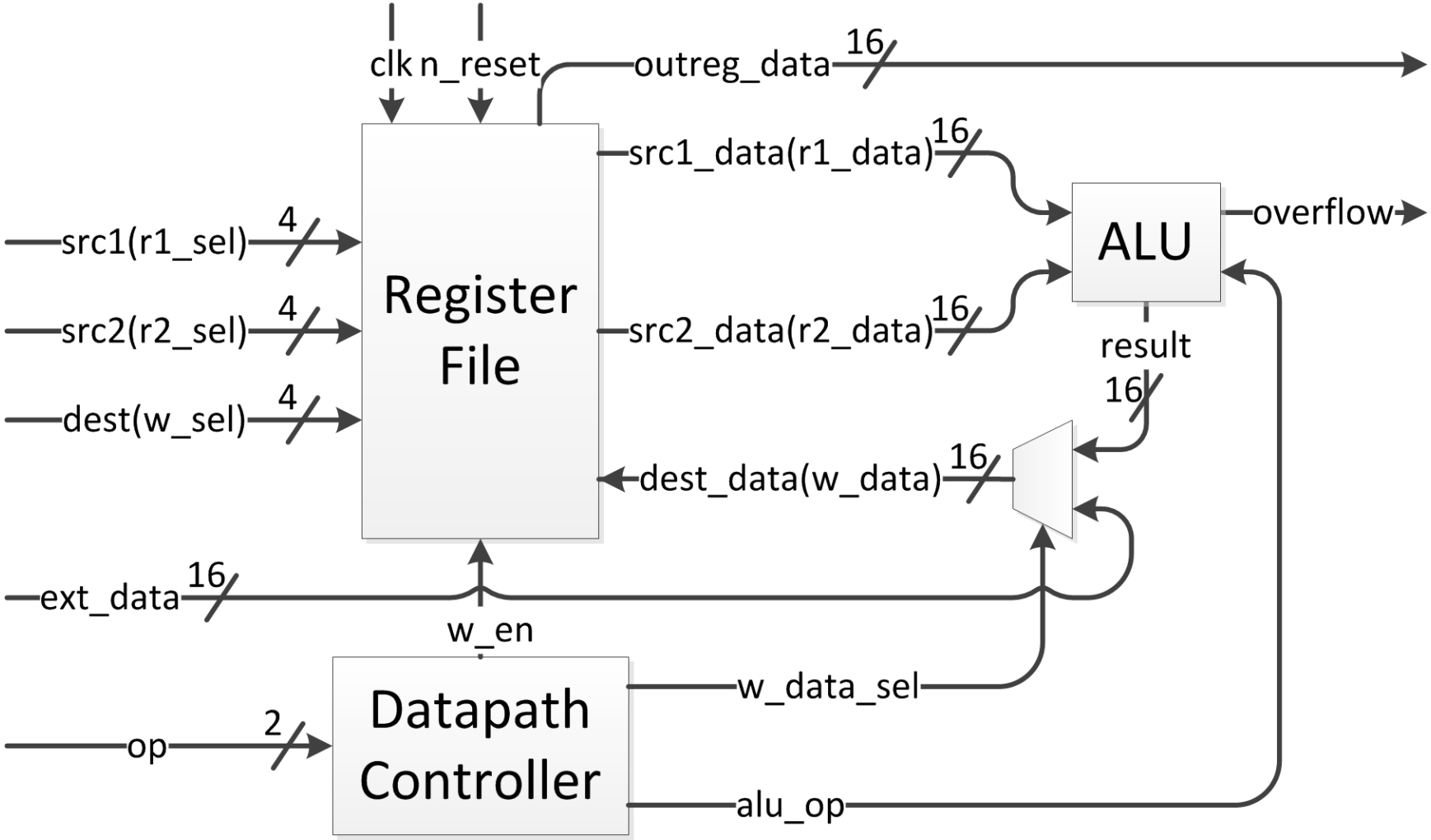

Figure 2: Datapath Architecture Diagram

5.4.4. Register File Architecture

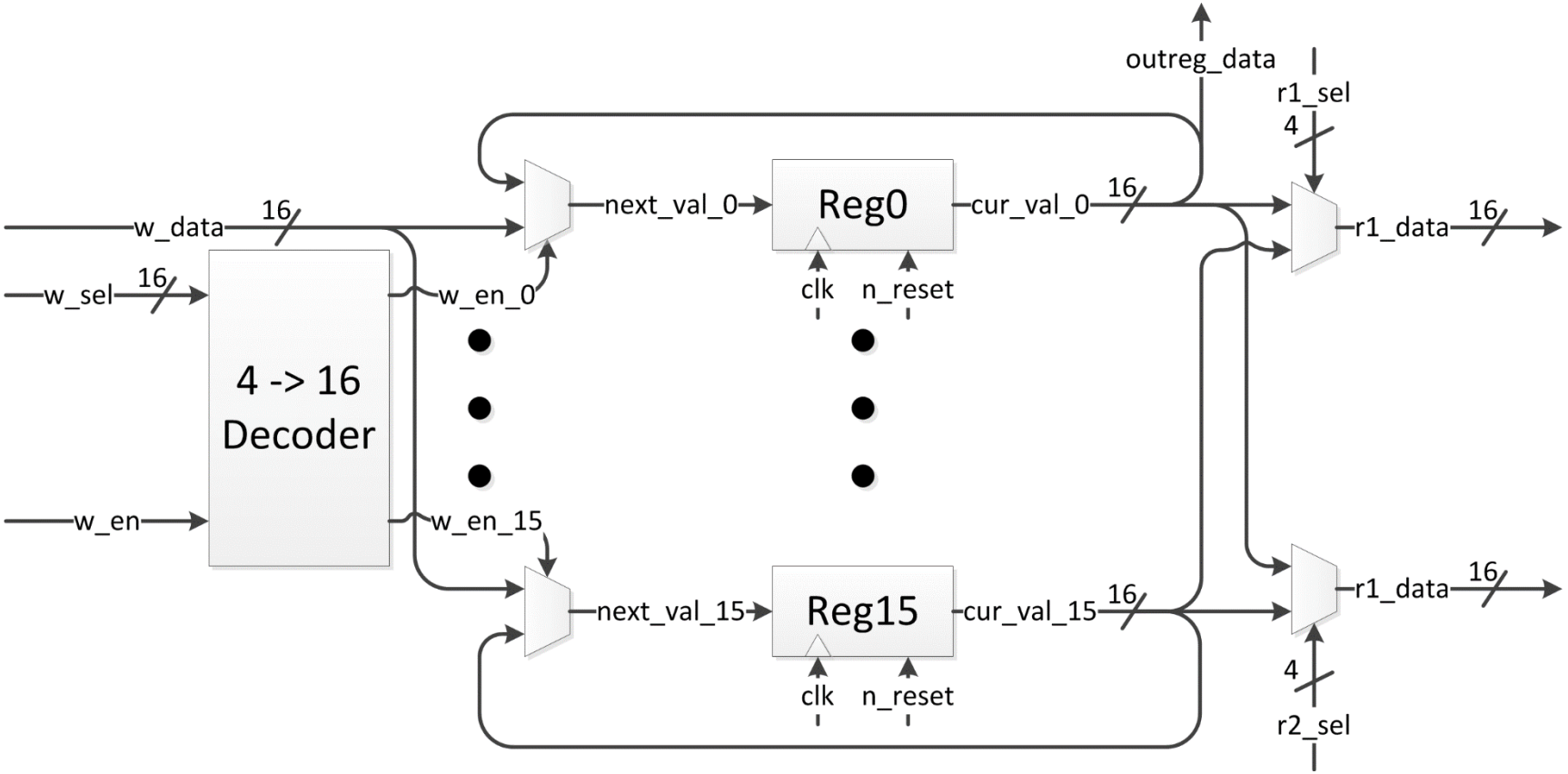

Figure 3: Register File Architecture Diagram

5.5. Top Level (avg_four.sv)

This is the top level module which will connect all the individual components. The division by four will be performed in the top level.

An external source puts an unsigned 16-bit word on the data input, and asserts data_ready to indicate to the design that the data is valid. (Note that data and data_ready are asynchronous signals.) At this point the design will store the contents of data in a register file. Then it will add the last four samples together and store the result in the accumulator (one of the registers in the datapath). The samples will need to be moved to make way for the next cycle, discarding the last data point, e.g., sample4 is discarded, sample3 becomes sample4, and so on. While the processing occurs, the modwait signal is asserted to indicate that the system is not yet ready to process a new sample. If an overflow occurs during the averaging process, err is asserted to indicate an error and remains asserted until the beginning of the next averaging operation. The data on avg_out is not valid during this time. Below is a timing diagram showing this operation, although the 1,000 sample signal is not shown as a 1,000 samples being processed would not have legible waveforms when confined to tolerable physical size.

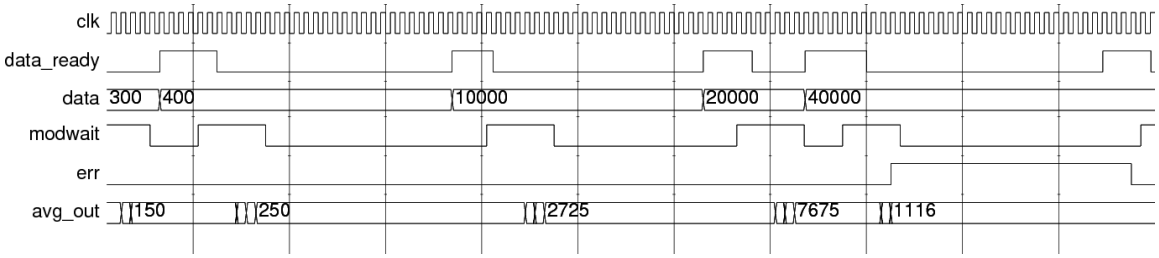

Figure 4: Example Waveform

6. Design Preparation

Before starting to code up/implement a design is incredibly important to plan it out in detail first, otherwise extra-long debugging sessions and confusion will abound. This is why we have been having you practice your RTL diagraming and state transition diagraming skills in the prior labs. Make sure to always diagram out your design in both block diagrams as shown in the prior sections of this lab manual and RTL diagrams for each component. Also, make sure to structure your code to match your diagrams so that you can use them as aids in designing and debugging. Additionally, try to keep your 'register' process as simple as possible and keep all of the combinational 'next value/state' logic and 'output' logic in separate always blocks or dataflow statements. This helps in debugging as you have more points/information at your disposal to check and debug the design and it increases the likelihood of the synthesizer interpreting your code the way you want it to. Messy/unorganized code can easily increase your design + debugging time in a super-linear (sometimes even exponential) relationship.

6.1. Two Sample Sliding Window Averaging Filter

To start things out, let us consider a simpler averaging filter design that only averages two samples instead of four. The following pseudo code describes the operation of the whole design for a two sample sliding window averaging filter using the same architecture as detailed in sections 4 and 5:

```
idle:  if (data_ready=0) goto idle  ; wait until data_ready=1
store: if (data_ready=0) goto eidle
       reg[3] = data                 ; Store data in a register
       err = 0                       ; reset error
sort1: reg[1] = reg[2]              ; Reorder registers
sort2: reg[2] = reg[3]              ; Reorder registers
add:   reg[0] = reg[1] + reg[2]     ; Add data, store result
       if (V) goto eidle            ; On overflow, err condition
         goto idle
eidle: err = 1
       if (data_ready=1) goto store  ; wait until data_ready=1
       if (data_ready=0) goto eidle
```

The setup4 script provided you with a Dia file containing state transition diagram template. Fill in the template state transition diagram for to describe the Finite State Machine for the controller block needed to implement the above pseudo code using the architecture described in sections 4 and 5. **Once you have completed the two sample state transition diagram, have a TA verify and check off your work up to this point.**

6.2. Extending the Two Sample Filter Design

Now extend the provided pseudo code for the two sample filter to work descript the operation of a four sample filter, and store this pseudo code in a file called 'pseudo_code.txt' in your 'docs' folder. **Once you have completed the four sample pseudo code, have a TA verify and check off your work up to this point.**

Now using this pseudo code for a four sample filter design, make a copy of the two sample filter controller state transition diagram and extend it describe a controller for implementing a four sample filter. **Once you have completed the four sample state transition diagram, have a TA verify and check off your work up to this point.**

6.3. Component Diagraming

Now that you have functional state transition diagrams, it's time to think about how the FSM will translate to hardware in terms of flip-flops and combinational logic, as well as time to think about how the division block should be implemented. Create a RTL diagram for the controller block and **have a TA verify and check off your work up to this point.**

Next create a schematic level diagram of the division block. Now remember, this is not an arbitrary division block, as it is always/only dividing the output of the datapath by four. Think about what is special about division by a power of two. **Once you have completed the division block's schematic level diagram, have a TA verify and check off your work up to this point.**

7. Structural Verilog model for the sliding window average.

As mentioned before, after you have created the internal blocks, now you have to connect them together to make the top level design avg_four.sv.

You will need to declare the module for the top level block. In this case, your module declaration should look like as follows (although any outputs may be declared as either wire or reg):

```
module avg_four
(
  input  wire clk,
  input  wire n_reset,
  input  wire [15:0] sample_data,
  input  wire data_ready,
  output wire one_k_samples,
  output wire modwait,
  output wire [15:0] avg_out,
  output wire err
);
```

You must declare any intermediate nets and variables you want to use. An intermediate net is any wire that is not a top-level input or output. For example, most of the nets used to connect to the Datapath are intermediate signals. Make sure to give them relevant names to aid you working with your code.

After you declare the intermediate nets and variables you will need, you can start connecting the components. As a reminder, the recommend syntax for a port map is:

```
<component module name> <instance name/tag>
(
  .<port name>(<connecting signal name>),
  …
  .<port name>(<connecting signal name>)
);
```

8. Simulating the design

This design is a hierarchical design, due to the use of sub-modules, and is the style of designs that you will most commonly work with as designs that have no sub-modules are either not going to be very interesting or are going to be incredibly difficult and time consuming to design, test, and maintain. From now on the overall designs you will be making in this course will be hierarchical and this means that in order to efficiently simulate these designs, and have them properly graded during submissions, you will need to become familiar with populating the first three variables in your makefiles and using the "sim_full_source" and "sim_full_mapped" make rules.

When simulating the design, you may notice that you can see the hierarchy of the datapath in you ModelSim design window even though you do not have access to the source. Can you identify the sub-blocks within the datapath? Can you identify any internal signals?

For this lab you have been provided with a gold model of the top-level design, called gold_avg_four. This gold model is a source only functional implementation of the design. If your design output matches the gold model, then you know it is working correctly. However, your design may still be correct if you do not exactly match the gold model but have very close behavior to where the two are still functionally equal.

9. Grading Procedure

Important: The top three variables in your makefile must be properly filled out or the grading system will be unable to properly grade your design and will usually result in a grade of 0 points. If you can run "make veryclean sim_full_mapped" and your simulation works as expected then you have properly filled out these variables.

The command for the automated grading script runs of your design is:

```
submit Lab4AVG
```

The automatic grading script itself is multithreaded to permit for simultaneous submissions from multiple students. It may take a minute, or more during peak load times, to grade your submission and grades will be displayed when it has finished. Once you run the submit command, do not close the terminal, terminate the command, or logout of your session until the results have been displayed. Doing either of these actions will lock your connection to the server and you will not be able to submit anything to it for any of the submit commands until you notify a TA to have them reset the access for your account. If for some reason you need to leave the terminal before it is finished, and are on one of the white thin clients, simply lock your session and it will run in the background until you login again.

If you would like to check the status of prior grades, as well as see how many attempts you have used, run

```
submit Lab4AVG -c
```

The above command will not count against your submission attempts.

The automatic grading script will do two things. First it will test the SOURCE version of your design. Second, it will re-synthesize and then test the MAPPED version of your design. For this lab, 30/50 of your grade will be determined 100% from your MAPPED version grade (the last 20/50 comes from the preparation and post lab). This is to reinforce the importance of getting your mapped version working. The source version score is included just as a reference.

In order to complete this lab's course objective, you have to score at least 50% (15/30) on your mapped version test.

10. Comments

- You are required update the variables in the makefile provided to you by dirset so that it can compile and synthesize your design. The grading script will parse your makefile for the values assigned to the variables and will use those to compile and synthesize everything during the testing. If your variables are not up to date for your design it will result in the grading system not being able to compile and synthesize your design or sections of it, which will effectively waste one of your submission attempts.
- Your design is required to run at a minimum of 200 MHz or it will fail the automatic grading script. **In order to have your synthesized design run at this frequency add the following delay optimization commands to Step 2 of synthesis commands variable in your makefile.**
    - set_max_delay 1 -from "DP/ALU_MAP /CLA_ADDR/A" -to "DP/ALU_MAP /CLA_ADDR/S"
    - set_max_delay 1 -from "DP/ALU_MAP /CLA_ADDR/B" -to "DP/ALU_MAP /CLA_ADDR/S"
    - set_max_delay 1 -from "DP/ALU_MAP /CLA_ADDR/Cin" -to "DP/ALU_MAP /CLA_ADDR/S"
    - set_max_delay 1 -from "DP/ALU_MAP /CLA_ADDR/A" -to "DP/ALU_MAP /CLA_ADDR/V"
    - set_max_delay 1 -from "DP/ALU_MAP /CLA_ADDR/B" -to "DP/ALU_MAP /CLA_ADDR/V"
    - set_max_delay 1 -from "DP/ALU_MAP /CLA_ADDR/Cin" -to "DP/ALU_MAP /CLA_ADDR/V"

  Note: "DP" in the above paths is the tag/label for the portmap of the datapath in your design and each command should be typed as one line. Also, it is recommended that you type out the commands instead of copying them from here, as copying them often also adds invisible formatting characters or slightly different formatting that will cause them to not work correctly and be relatively difficult to debug.

- You must set the clock period constraint in the 'CLOCK_PERIOD' variable in your makefile in addition to the above delay optimizations, for the design to be guaranteed to run at 200 MHz. *Note: that given the complexity of the design if you only use a constraint of 5 ns for the clk constraint the synthesizer will stop working once it has "met" it even if it results in a critical path of exactly 5 ns. We have observed that the delays in Modelsim tend to be slightly longer than those reported by Design Compiler in your mapped reports folders. To adjust for this, use time constraints smaller than 5 ns, such as 4ns or less, to force it to optimize further and give your design the cushion it needs to handle these longer simulation delays.*
- The code for the grading test bench used by the grading script will not be disclosed to students nor will the specifics of the test cases be told to the student.
- The majority of the points come from successfully passing the synthesized design portion of the grading test bench; therefore make sure you have an error-free synthesis.
- You will be allowed a maximum of 5 passes through the Lab 4 grading test bench (i.e. 5 chances to run 'submit Lab4AVG') – note: this number will decrease in future labs. **Only the last submission will count!** So use SVN! If a previous submission is better than your most recent submission, restore it from SVN and submit again. SVN is your friend.

11. Postlab

11.1. Design Questions

Place answers to the following questions in a file called 'Lab4.txt' inside your 'docs' directory.

Question: What is the minimum amount of time that data_ready must remain asserted to ensure correct operation? What is the minimum amount of time, in clock cycles, that data must remain valid after data_ready is asserted in order to ensure correct operation? (You may assume that all setup and hold times, as well as any propagation delays, are negligible.)

Question: Assume that the datapath was extended to support a multiplication instruction defined by dest = src1 * src2. Write the pseudocode for a state machine that would calculate the dot product of the last four samples. (Remember the dot product is (sample1*sample3) + (sample2*sample4). Also assume that the division code has been removed from the top level.

11.2. Verilog File IO Demo

Part of what the "setup4" script did was copy over a customized makefile, customized test bench file, 24-bit bitmap image, and a waves formatting .do script. These files were copied over in order to support this quick demo on how to use the file IO syntax in Verilog in test bench designs. For this, the provided test bench uses Verilog file IO syntax to open the bitmap image file, extract the file header information, feed the image data through the filter design you completed in this lab, and then store the filtered image data and appropriate file header in a new bitmap file. The test bench is heavily commented and you are expected to look through it and try to use it to help you understand how to utilize the file IO syntax in Verilog to work with files for testing purposes. Upon analyzing the test bench file (tb_avg_four_image.sv) you should notice that the file IO syntax in Verilog is very similar to file IO in C.

At this point in the course you are not expected to fully understand how to use Verilog File IO, however it is often very useful during testing project designs later in this course so it is in your best interest to look over the code provided in the test bench, conduct online searches about Verilog File IO and try to create some simple test benches with that use file IO to feed values to your design from this lab, as well as ask the course staff questions you have about using Verilog File IO.

If you were unable fully complete the design for this lab, simply change the "avg_four" port maps in the test bench file to be for the gold model (gold_avg_four) instead.

To run the file IO demo, run the following command in your Lab4 directory:

```
make sim_image
```

After running the simulation has finished open both the "test_1.bmp" and "filtered_1.bmp" files in image viewers and compare them.

Place answers to the following questions in a file called 'Lab4.txt' inside your 'docs' directory.

Question: How are the image files different? Does this make sense given the filter design built in the lab? Why or why not?

Question: What is the general syntax for each of the file IO functions used in the provided test bench (tb_avg_four_image.sv)?

Question: What are the different format specifies available for use in file functions like $fscanf(…)?

Once you have answered all of the postlab questions and have finished the team report, run the following command on terminal window to submit your postlab:

```
submit Lab4Post
```