ASIC Design Lab 5:
Introduction to Datapath-based Design in Verilog

In this lab you will:

- Synthesize, test, and verify the functionality of the mapped version of a Moore model state machine of finite impulse response (FIR) filter.
- Use Verilog file IO to feed a grayscale image through your finite impulse response 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 Lab5 directory like you have for the other labs. Then run this labs setup command.

setup5

Once you have committed your initial setup to your GIT repo, have a TA check off your work up to this point.

3. Overall Design

You will design an ASIC that implements a very small finite impulse response (FIR) filter. It will read in a series of data samples and output the filtered data, as well as keeping track of the number of samples processed. Each output sample will be computed as the weighted sum of the last four input samples. This design implements a configurable four sample finite impulse response filter, and is an example of a digital filter. The filter's design allows the FIR coefficients to be adjustable during operation. Such a design would be useful for smoothing an A/D conversion to reduce noise, for example. A diagram showing the operating theory of a general FIR filter is given in Figure 1. The design architecture diagram of the system is given to you in Figure 2. 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.

3.1. Module Declaration

```
module fir_filter
(
      input  wire clk,
      input  wire n_reset,
      input  wire [15:0] sample_data,
      input  wire [15:0] fir_coefficient,
      input  wire data_ready,
      input  wire load_coeff,
      output wire one_k_samples,
      output wire modwait,
      output wire [15:0] fir_out,
      output wire err
);
```

3.2. 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	100MHz system clock signal with a 50% duty cycle.
n_reset	Input	Active low reset
sample_data [15:0]	Input	16-bit unsigned data input.
fir_coefficient [15:0]	Input	16-bit fixed-point coefficient [0.0, 1.0] to load
load_coeff	Input	Active high signal that indicates when FIR coefficients should be loaded.
data_ready	Input	Active high signal that indicates when a sample is ready to be processed.
one_k_samples	Output	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
fir_out [15:0]	Output	The 16-bit unsigned FIR filter of the last four samples
err	Output	Active high signal indicating that an overflow occurred during averaging.

The **asynchronous** input data_ready and load_coeff 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. When the synchronized signal, lc, goes high, it signals to the control unit that 1 of a sequence of four new FIR coefficients are ready to load. Modwait should go high while loading the first FIR coefficient in the register file and desert to signal the module is ready to receive the next FIR coefficient. 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].

3.3. Design Usage Constraints

- The total time to process one sample must not exceed 25 clock cycles (250 ns). Within 25 cycles of data_ready assertion, the outputs must be valid and the modwait signal deserted.
- 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.

3.4. Implemented Algorithm

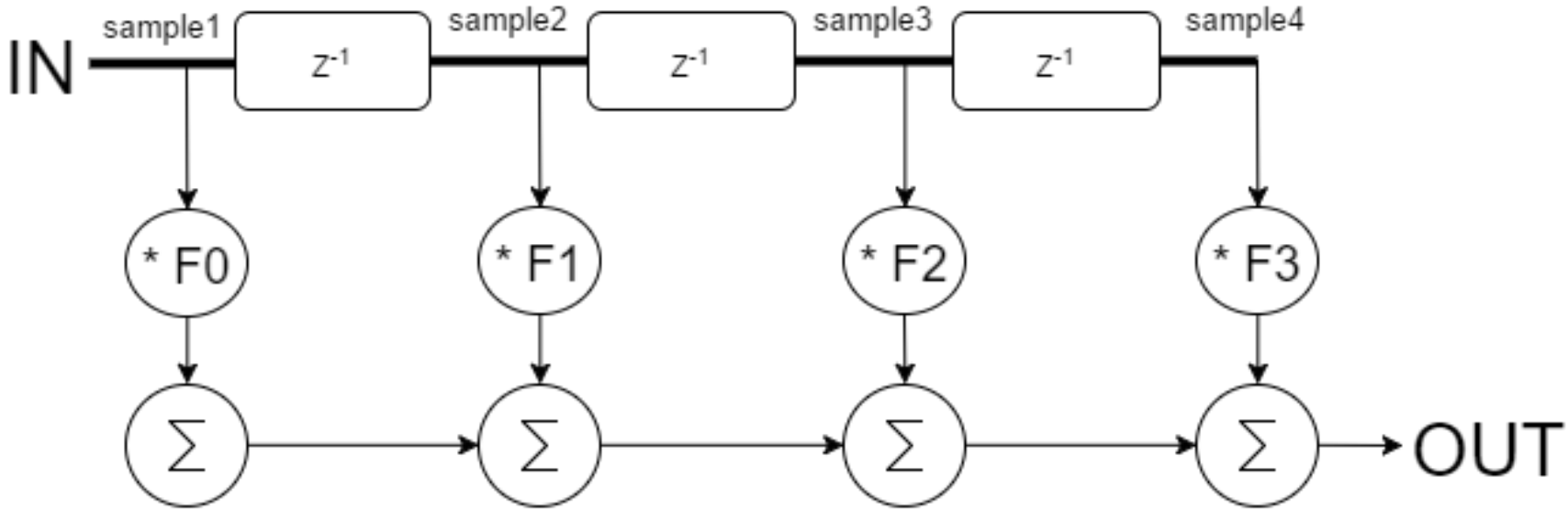

Figure 1: Finite Impulse Response Filter Theory of Operation

Z^{-1} is equivalent in circuit design to storing a value in a register for one clock cycle

Example: sample2 is sample1 from 1 cycle ago

3.5. Design Architecture

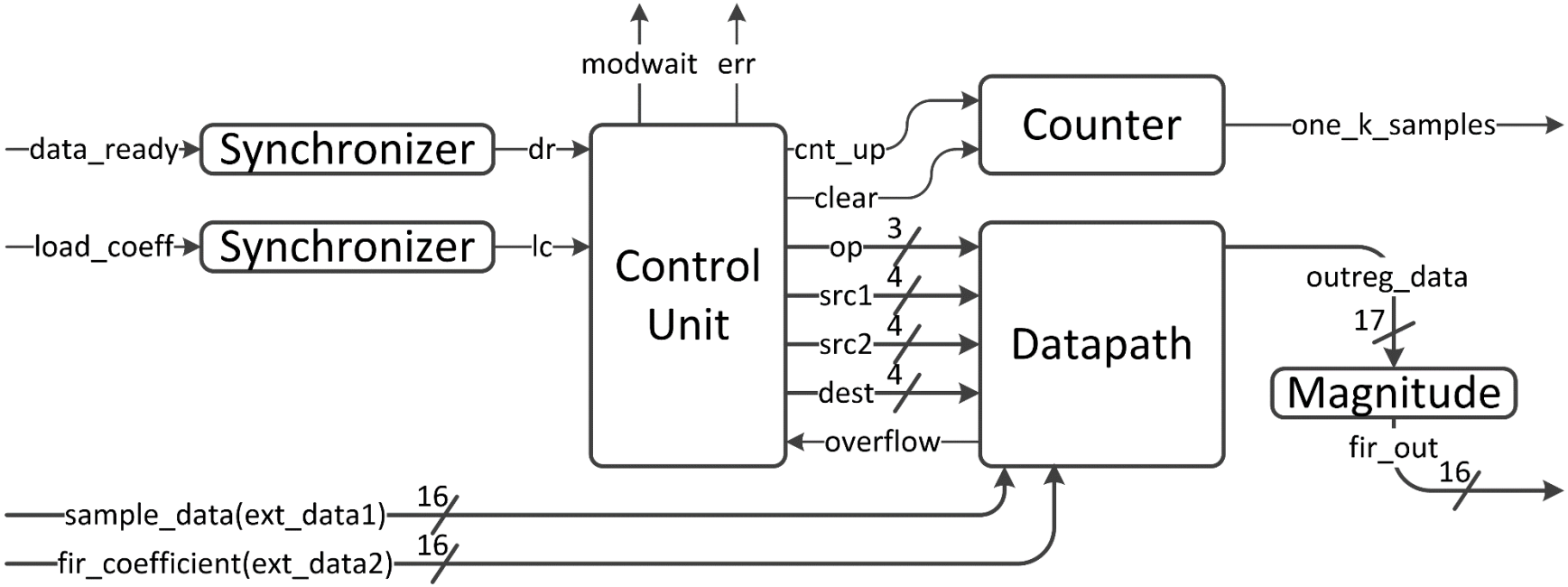

Figure 2: Finite Impulse Response Filter Architecture

Note: Clock and Reset signals are assumed to be sent to the appropriate blocks and are not shown above.

4. Detailed Design Specifications

4.1. 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 can 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. Lowering the modwait signal indicates that the design is ready to receive another data sample and done processing the previous sample. After receiving dr, set the modwait signal high, 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 controller will manipulate the datapath to perform the FIR filtering algorithm presented in Figure 1 and explained in section 4.1.1. If at any point during the summation or multiplication an overflow occurs, stop processing, assert the error signal, err, and continue 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. See Figure 5's blue section for an example of correct behavior when dr is de-asserted before modwait is de-asserted.

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 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. Section 4 describes the datapath in detail.

Hint: The easiest way to ensure that modwait is stable is to have the output value come directly from a flip-flop. This is referred to as 'registering the output'.

4.1.1. Data Processing Operation

The data_ready input should be asserted until after the modwait output is asserted as shown in the first "Valid 3 cycle DR" portion of the following timing diagram. However due to the delay of the synchronizers between the data_ready input and the dr internal signal a 2-cycle pulse of data_ready will work as well. However, a 1-cycle pulse of data_ready should result in immediate processing termination and error reporting as shown in the "Invalid 1-cycle DR" section of the following timing diagram.

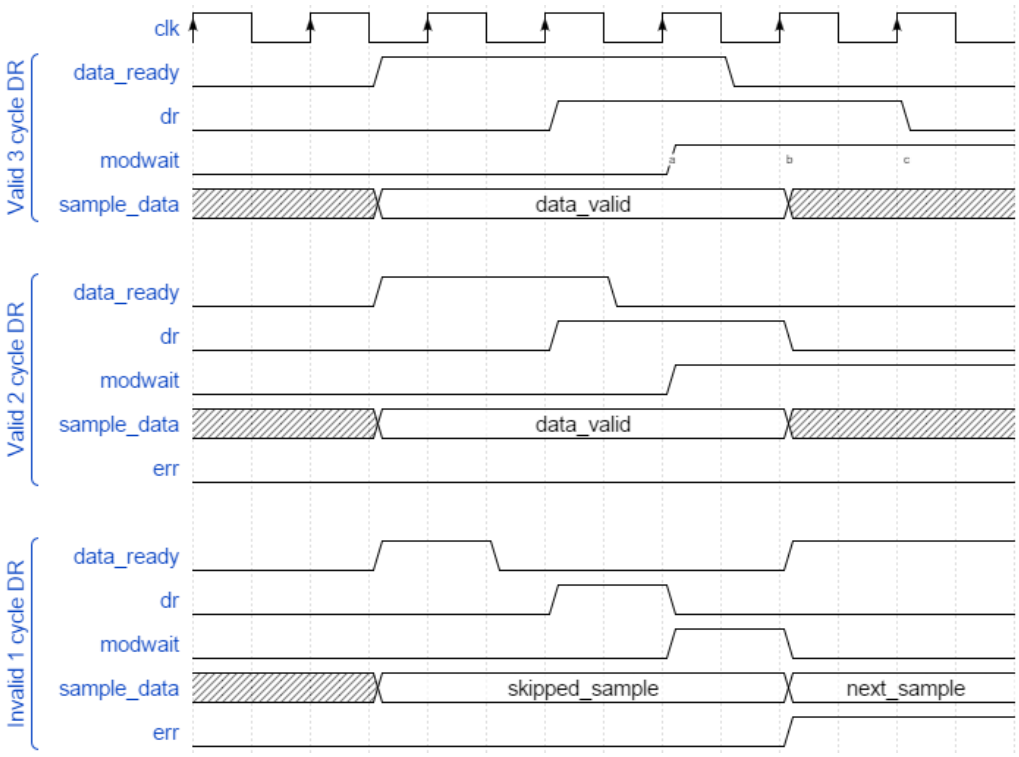

Figure 1: Data Processing Ignition Timing Diagram

4.1.2. FIR Coefficients and Algorithm

When in the IDLE state and the load_coeff signal is asserted, the control unit should handle loading the new four coefficient values into four registers of your choosing and clear the counter. Lower the modwait signal after storing each coefficient to indicate the control module is ready to receive the next coefficient. Four coefficients will be used in this design. F0 will be sent first and F3 will be sent fourth. The control module only needs to allow coefficient loading or data sample processing sequences to start while in IDLE. Once either of the sequences has started it must run to completion before a new sequence (either coefficient loading or sample processing) is allowed to start. FIR filters operate on the current sample and several relative-to-time older data samples, 1 current and 3 older data samples for this filter. It is important to note that you have to perform this algorithm in multiple cycles since you have a datapath that can only perform 1 operation per cycle. Loading of the coefficient is allowed to take up to 5 cycles (5 cycle long active modwait) per coefficient but realistically it can simply be done in 1 cycle per coefficient.

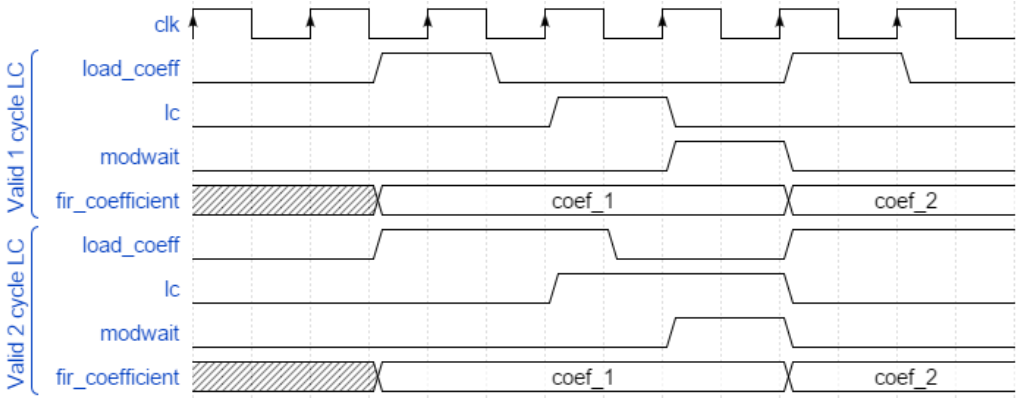

Figure 2: Allowed Load Coefficient Timing Sequences

4.1.3. FIR Coefficients

The FIR coefficients will be a small negative for F0, large positive for F1, large negative for F2, and a small positive integer for F3. The coefficients will be sent to the FIR filter module as 16-bit unsigned fixed-point decimals. The MUL op in the datapath is not a general-purpose multiplier. The MUL op does not support signed numbers and performs fixed point math expecting the coefficient in reg[src2] and the data sample in reg[src1]. Thus, it is necessary to make the product F0*sample1 and F2*sample3 negative by subtracting them from the accumulator, instead of adding them (SUB instead of ADD). It is ok for the reg[0] to have a negative value during or after the FIR algorithm is done.

4.1.4. Module Declaration

```
module controller
(
      input  wire clk,
      input  wire n_reset,
      input  wire dr,
      input  wire lc,
      input  wire overflow,
      output reg cnt_up,
      output reg clear,
      output wire modwait,
      output reg [2:0] op,
      output reg [3:0] src1,
      output reg [3:0] src2,
      output reg [3:0] dest,
      output reg err
);
```

4.1.5. Port Descriptions

Table 2: Controller Block Port Descriptions

Signal	Direction	DESCRIPTION
clk	Input	100MHz 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.
lc	Input	This is the synchronized form of the load coeff signal that signifies that the next coefficient is ready load.
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).
clear	Output	This signal is used to clear the counter to zero, when lc is asserted.
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 [2:0]	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.

4.2. 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 for the current set of coefficients 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. Also, the counter should be cleared upon a change of coefficients.

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 Lab5 folder by the setup script.

4.2.1. Module Declaration

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

4.2.2. Port Descriptions

Table 3: Counter Block Port Descriptions

Signal	Direction	Description
clk	Input	100MHz 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).
clear	Input	This signal is a synchronous reset, set the counter to zero
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.

4.3. Synchronizer (sync_low.sv)

The top-level inputs, data_ready and load_coeff¸ need to be synchronized before they can be used. See the end of this handout for a discussion of how to accomplish this.

Since the requirements are identical to those for the sync_low design you created and tested in lab 2, this should simply be a copy of that design, which is why that design file was copied into your Lab5 folder by the setup script.

4.4. Magnitude (magnitude.sv)

If the FIR algorithm produces negative output from the datapath, this block will output the positive magnitude on fir_out instead. Otherwise, this block will pass the datapath's out_reg value on to fir_out. This is desired as this block works with negative coefficients and the samples are unsigned, thus the output should be unsigned as well. The negative output will be in 2's compliment format and will have to be converted to the positive form of 2's compliment should the outreg_data of the datapath have a negative number on it.

4.4.1. Module Declaration

```
module magnitude
(
      input  wire [16:0] in,
      output wire [15:0] out
);
```

4.4.2. Port Descriptions

Table 4: Magnitude Block Port Descriptions

Signal	Direction	Description
in<16:0>	Input	Input port to be filtered for negatives, connect to outreg_data
out<15:0>	Output	Outputs magnitude filtered form of input.

4.5. 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.

4.5.1. Module Declaration

```
module datapath
(
      input  wire clk,
      input  wire n_reset,
      input  wire [2:0] op,
      input  wire [3:0] src1,
      input  wire [3:0] src2,
      input  wire [3:0] dest,
      input  wire [15:0] ext_data1,
      input  wire [15:0] ext_data2,
      output wire [16:0] outreg_data,
      output wire overflow
);
```

4.5.2. Port Descriptions

Table 5: Datapath Port Descriptions

SIGNAL	DIRECTION	DESCRIPTION
clk	Input	100MHz system clock signal with a 50% duty cycle.
n_reset	Input	Asynchronous active low system reset signal. All Flip-Flop values in the design are set immediately to logic-0.
op<2:0>	Input	The operation to perform: 000 – NOP 001 – COPY: Copy from register 'src1' to register 'dest' 010 – LOAD1: Store 'ext_data1' in register 'dest'. 011 – LOAD2: Store 'ext_data2' in register 'dest'. 100 – ADD: register 'dest' = register 'src1' + register 'src2' 101 – SUB: register 'dest' = register 'src1' – register 'src2' 110 – MUL: 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_data1 <15:0>	Input	Data to be loaded into a register (see op).
ext_data2 <15:0>	Input	FIR coefficient to be loaded into a register (see op).
outreg_data <16:0>	Output	Value stored in the output register (register 0).
overflow	Output	Set if the current operation produces an overflow. (Generated asynchronously and valid when the op is ADD or MUL.)

4.5.3. Datapath Architecture

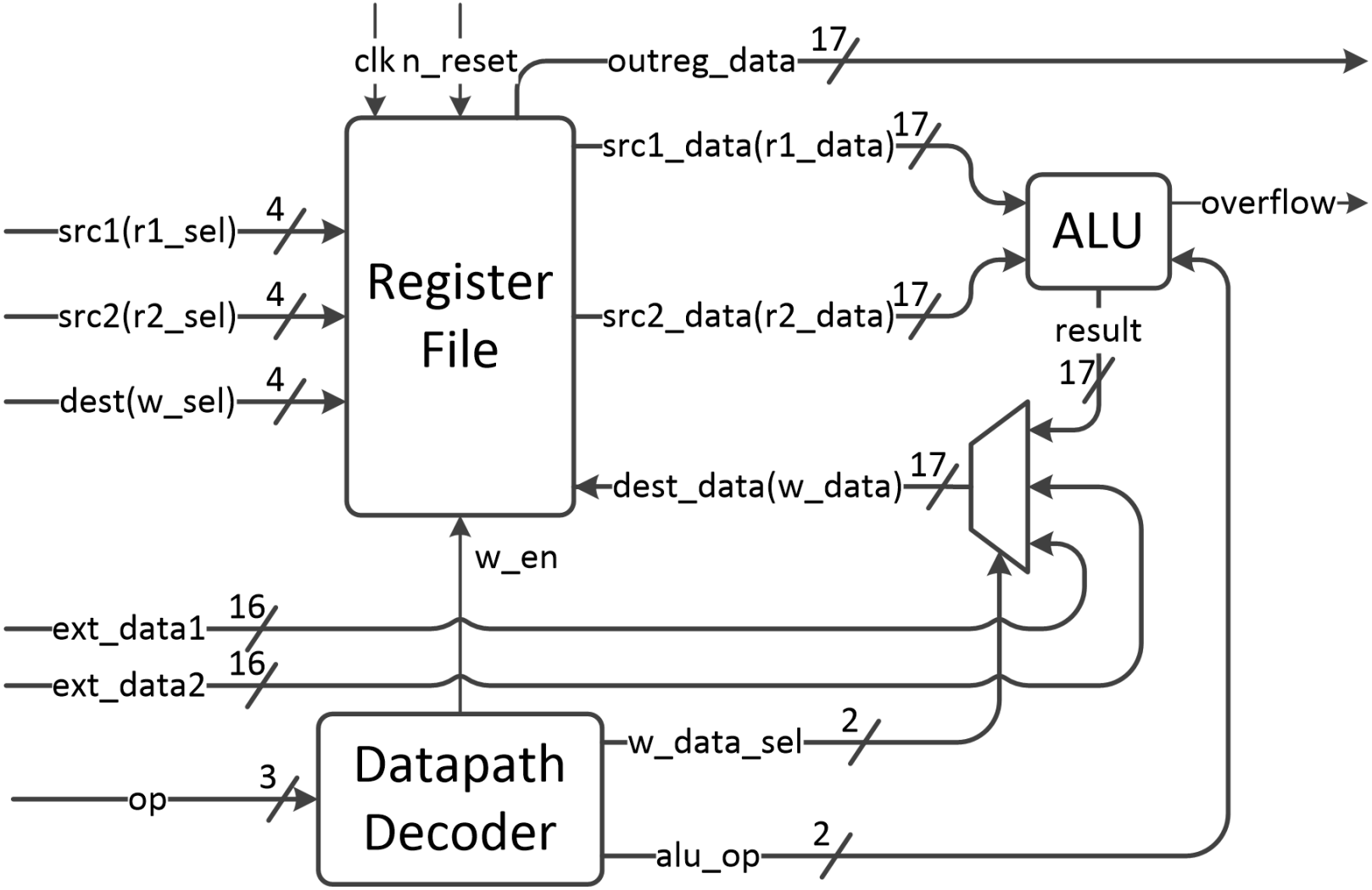

Figure 3: Datapath Architecture Diagram

4.5.4. Register File Architecture

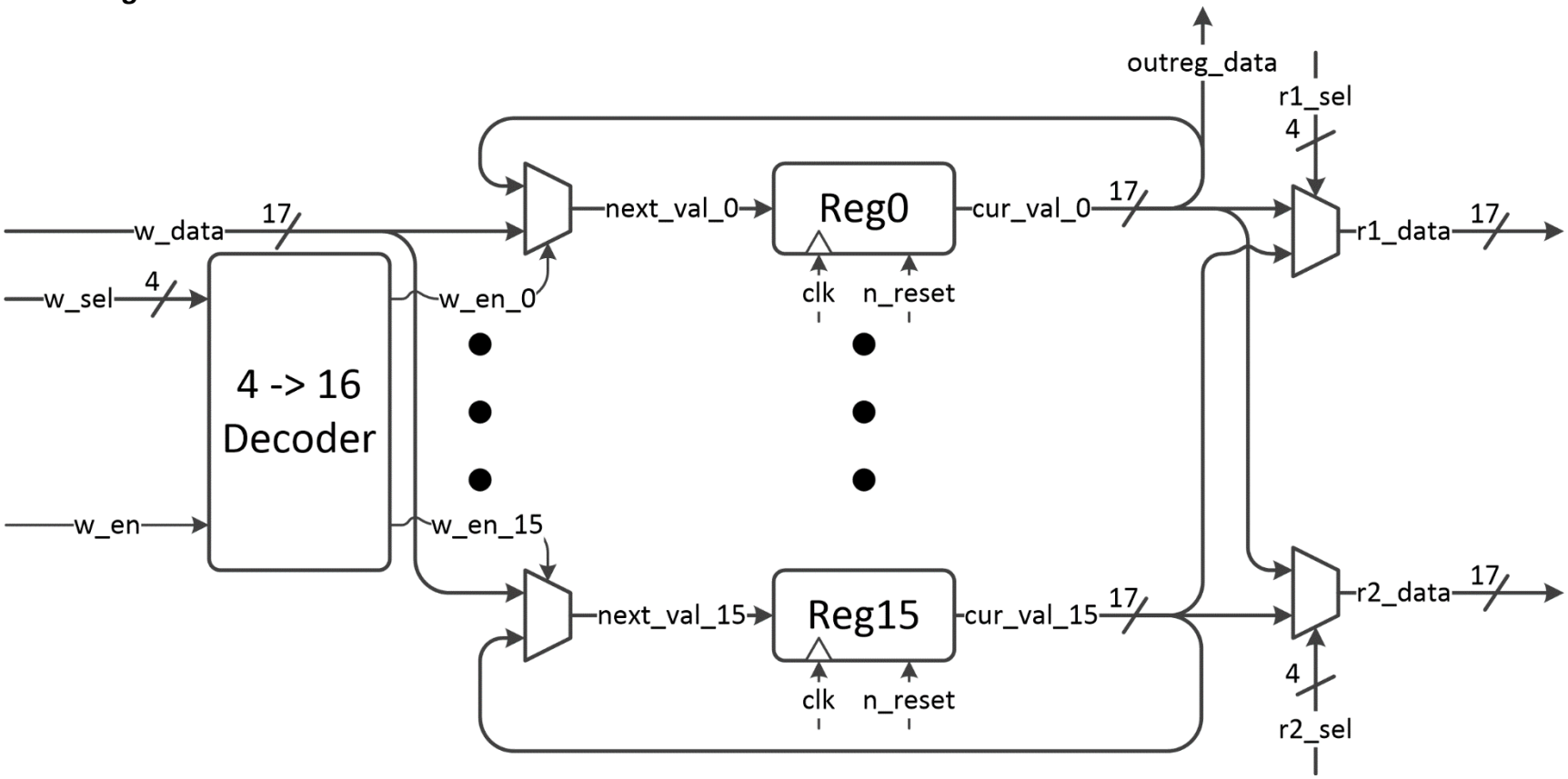

Figure 4: Register File Architecture Diagram

4.6. Top Level (fir_filter.sv)

This is the top level module which will connect all the individual components. 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 multiply the oldest sample by FIR coefficient F3, through to multiplying the newest sample by FIR coefficient F0, as seen in Figure 1. Then it will sum those intermediate sample products and store the result in the accumulator (one of the registers in the datapath). Sample products that had 'negative' coefficients must be subtracted from the accumulator instead of added to it. 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 multiplying sampleX by its relevant FX coefficient, do not overwrite the original sampleX with the result as it will still be needed for the next calculation where it will become sample(X+1). While 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 filtering process, err is asserted to indicate an error and remains asserted until the beginning of the next filtering operation. The data on fir_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.

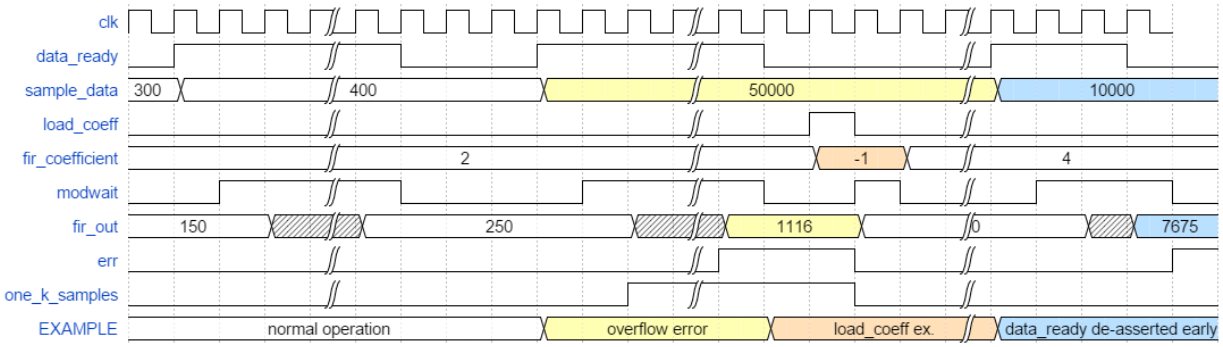

Figure 5: Example Waveform

5. 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_comb' 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 generally will increase your design and debugging time in a super-linear (sometimes even exponential) relationship.

5.1. Two Sample High-Pass Filter

To start things out, let us consider a simpler FIR filter design that only uses two samples instead of four. There should be an initial state that the FSM in the controller module goes to out of reset that handles loading in the four FIR coefficients and storing them in two registers that will not be overwritten for the duration of filter operation, reg[4] and reg[5] here. 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 3 and 4, excluding the initial coefficient load:

```
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
zero:       reg[0] = 0                  ; zero out accumulator
sort1:      reg[1] = reg[2]             ; Reorder registers
sort2:      reg[2] = reg[3]             ; Reorder registers
mul1:       reg[6] = reg[1] * reg[4]    ; sample2 * F2
add:        reg[0] = reg[0] + reg[6]    ; add Large pos. coefficient
            if (V) goto eidle               ; On overflow, err
condition
mul2:       reg[6] = reg[2] * reg[5]    ; sample1 * F1
sub:        reg[0] = reg[0] - reg[6]    ; sub Large neg. coefficient
            if (V) goto eidle               ; On overflow, err
condition
            else goto idle
eidle:      err = 1
            if (data_ready=1) goto store  ; wait until data_ready=1
            if (data_ready=0) goto eidle
```

Create a state transition diagram to describe the Finite State Machine for the controller block needed to implement the coefficient loading process and the above pseudo code for the data processing using the architecture described in sections 3 and 4. **Once you have completed the two-sample state transition diagram, have a TA verify and check off your work up to this point.**

5.2. Extending the Two Sample Filter Design

Now extend the provided pseudo code for the two-sample filter to work for the operation of four sample filter described above, and store this pseudo code in a file called 'pseudo_code.txt' in your 'docs' folder. Remember the coefficient order of small negative, large positive, large negative, small positive for a four-bit FIR filter. **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.**

5.3. 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.**

6. 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 fir_filter.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 fir_filter
(
      input  wire clk,
      input  wire n_reset,
      input  wire [15:0] sample_data,
      input  wire [15:0] fir_coefficient,
      input  wire load_coeff,
      input  wire data_ready,
      output wire one_k_samples,
      output wire modwait,
      output wire [15:0] fir_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>)
);
```

7. 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?

8. Grading Procedure

Important: The top two 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 Lab5
```

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 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 Lab5 -c
```

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, 60/75 of your grade will be determined 100% from your MAPPED version grade (the last 15/75 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.

To complete this lab's course objective, you must score at least 50% (30/60) on your mapped version test.

9. 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.
- You must set the clock period constraint in the 'CLOCK_PERIOD' variable in your makefile, for the design to be guaranteed to run at 100 MHz. *Note: that given the complexity of the design if you only use a constraint of 10 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 10 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 10 ns, such as 9ns 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 3 passes through the Lab 5 grading test bench (i.e. 3 chances to run 'submit Lab5'. **Only the last submission will count!** So use git! If a previous submission is better than your most recent submission, restore it from git and submit again.

10. Postlab

10.1. Design Questions

Place answers to the following questions in a file called 'Lab5.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.)

10.2. Verilog File IO Demo

Part of what the "setup5" 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_fir_filter_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.

To run the file IO demo, run the following command in your Lab5 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 'Lab5.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_fir_filter_image.sv)?

Question: What are the different format specifiers 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 Lab5post
```