CMPE-260
Digital System Design 2
Laboratory Manual

Department of Computer Engineering
Rochester Institute of Technology

2185

Contents

Timeline

The schedule for the labs is as follows:

Week	Lab	
1	Lab 1	Introduction to Vivado & Simple ALU
2	Lab 2	Register File
3	Lab 2	Register File
4	Lab 3	Instruction Fetch Stage
5	Lab 3	Instruction Fetch Stage
6	Lab 4	Instruction Decode Stage
7	Lab 5	Execution Stage
8	Lab 5	Execution Stage
9	Spring Break	
10	Lab 6	Memory
11	Lab 7	Write Back
12	Project 1	Complete MIPS
13	Project 1	Complete MIPS
14	Project 2	Programming MIPS
15	Project 2	Programming MIPS

Submission Guidelines

These instructions apply to all submissions in the course

1. Everything must be in a PDF, except code
2. You must include *all* of the required items for full credit, make it look professional
3. Everything must be typed and proofread. *NOTHING HANDWRITTEN WILL BE ACCEPTED FOR POINTS*
4. Submit all code to MyCourses
5. Submit grading sheets to MyCourses appended to the back of the report or worksheet for that exercise (scan it in). They must be signed by the lab instructor or a TA to get any credit on the exercise
6. Submit the worksheets and reports to MyCourses
7. All entries not submitted by the beginning of the next lab are considered late, unless otherwise specified
8. *DO NOT ZIP YOUR FILES.*
9. If you have any questions, email *ALL* TAs for your section

Exercise 1 Introduction to Vivado & Simple ALU

Introduction

This exercise will detail the basic functionality of the Xilinx Vivado Design Suite. This software suite provides an integrated development environment (IDE) for the design, synthesis, and implementation of custom designs for reconfigurable hardware. The students will learn how to create a custom design from source, simulate it, synthesize it, implement it, and load it onto a Basys3 FPGA. These objectives will be met by first completing a tutorial for a 4-bit ALU supporting two operations. The design will be expanded to support three additional operations and 32-bit width in a future exercise.

Procedure

1. Follow the Xilinx Vivado Tutorial in Appendix A to start the design for a partial ALU

2. To create the partial ALU, create a component for each of the operations in Table 1.1 and assign them to the corresponding opcode in the top-level ALU design. Make sure that each component is declared with generic bit width and is instantiated in the ALU with a width of 32 bits. Ensure that the following 4-bit opcodes are used. While only five operations will be supported now, this will allow many more operations to be added in later exercises.

Table 1.1: Operations

Opcode	**Operation**
1000	Logical OR
1010	Logical AND
1011	Logical XOR
1101	Shift Right Logical (SRL)
1110	Shift Right Arithmetic (SRA)

3. Write a testbench for the ALU, similar to those provided in the tutorial. It should spot-check **at least** four "generic" cases for **each** operation as well as the following "edge" cases:

Value1	Operation	Value2
0x6	SRL	0x2
0x6	SRA	0x1
0x6	SRA	0x2
0xF0000000	SRA	0x1
0x0	OR	0x0
0x0	OR	0xF
0xF	OR	0xF
0x5	OR	0xA
0xA	OR	0x5
0x0	XOR	0x0
0x0	XOR	0xF
0xF	XOR	0x0
0xF	XOR	0xF
0x5	XOR	0xA
0xA	XOR	0x5
0x0	AND	0x0
0x0	AND	0xF
0xF	AND	0x0
0xF	AND	0xF

Note: The proper syntax for hexadecimal in VHDL is: `x"F"`

4. Using the testbench, perform both a Behavioral Simulation and a Post-Implementation Timing Simulation. Check the results for accuracy.

Exercise 1: Introduction to Vivado & Simple ALU

Student's Name:____________________ Section:____________________

Demo		Point Value	Points Earned	Date
Part 1: 4-bit ALU	Behavioral Simulation	4		
	Post-Synthesis Timing Simulation	4		
	Schematic	4		
	Synthesis Report	4		
	Implementation Report	4		
	Post-Implementation Timing Simulation	4		
	Hardware Demonstration	4		
Part 2: 32-bit ALU	Behavioral Simulation	16		
	Post-Implementation Timing Simulation	16		

To receive any grading credit students must earn points for both the demonstration and the report.

Exercise 1: Introduction to Vivado & Simple ALU

Report	Point Value	Points Earned	Comments
Abstract	5		
Design Methodology	15		
Results & Analysis	10		
Conclusion	5		
Source Code (Style)	5		
Total for prelab, demo, and report	100		

Exercise 2 Register File

Objective

This exercise investigates the design of a register file for a FPGA. The objective of this exercise is to implement, using VHDL, a digital system that is capable of storing multiple words, or groups of bits, as well as understanding how to properly test such a design. VHDL files are created and tested, both in simulation and in hardware.

Background

Computers need to be able to store and access information very quickly. However, simple digital systems like D flip flops, by themselves, cannot hold enough information to be useful. Thus register files are created that allow for one bus to access multiple locations, each which holds multiple bits. In this exercise, there are two outputs, which allows for two addresses to be read from at the same time, increasing the speed at which the necessary data can be obtained.

Prelab

- Draw a schematic of the register file following the specifications laid out below. It is not necessary to show the inner works of each component, just how they are tied together to create the register file.
- Review the use of generics in both entity declarations and instantiations.
- Print the sign off sheet.

Hardware Specification

The design must meet the following specifications and use all components listed at least once. Some will be used multiple times.

Register Module

The register module has the following features (see Figure 2.1):

- Parameters:
  - The size of the word to store, in bits. To be called `n`.
- Inputs (One bit unless noted otherwise):
  - `in` (n bits): Parallel input to be stored.
  - `clk`: Clock signal. Register file is rising edge triggered.
  - `we`: Write enable. Enabling allows register contents to be written.
  - `rst`: Resets the register to `r`. Asynchronous.
- Outputs:

  - `out` (n bits): Parallel output of the register module.

Consider using the behavioral style for the register module.

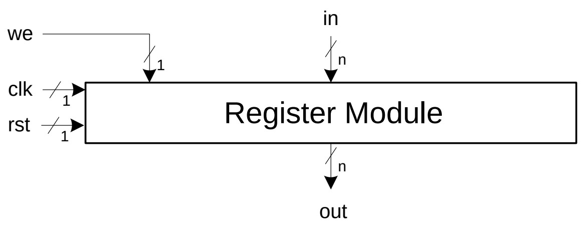


Figure 2.1: Register Module

Multiplexer

Create an `m`-to-1 MUX, whose eight inputs are `n` bits wide. `m` is the number of register modules in the file.

Decoder

Create an appropriately-sized decoder (i.e. if `m = 8`, the decoder would be 3-to-8). The input of this decoder should not be written as separate signals, but instead as one multi-bit signal. This is also true of the output.

Register File

The register file contains `m` register modules and follows a two-read, one-write format. See Figure 2.2.

- Parameters:
  - The size of the word to store, in bits. To be called `n`. Default to 8
  - The number of register modules. To be called `m`. Default to 8
- Inputs (One bit unless noted otherwise):
  - `rd1, rd2` ($\log_2$(`m`) bits): Read1 and read2. Selects which registers to read from.
  - `wr` ($\log_2$(`m`) bits): Write. Selects which register to write to.
    Note that the previous signals are $\log_2$*(`m`) bits long, allowing them to address all `m` registers.*
  - `in` (`n` bits): Data to be written to the appropriate register.
  - `clk`: Clock signal.
  - `we`: Write enable. Enabling allows register contents to be written.
  - `rst`: Resets all registers. Asynchronous.
- Outputs:
  - `out1, out2` (`n` bits): Parallel outputs containing the data from the selected registers.

Hints

- Use the output of the decoder in conjunction with the register file's `we` signal to drive the `we` on the register module.
- Use two MUXs, one to connect the output of the register specified by `rd1` to `out1`, and a second for `rd2` and `out2`.

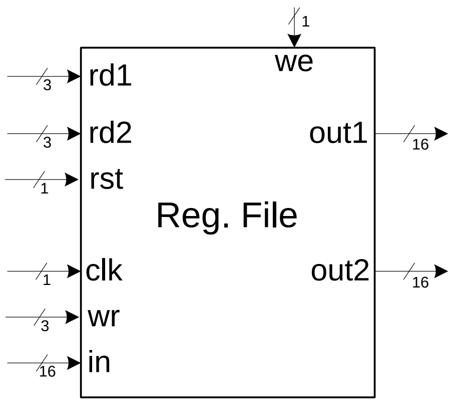


Figure 2.2: Register File

Testbench

Write a testbench to ensure that the register file is working properly. Some potential scenarios are listed:

- Resetting the register file
  - This should set all register modules to their reset value, `r`.
- Writing to each register module
  - A for loop can be used to write to every register module by modifying `wr`.
- Reading from each register module
  - Ensure that both `rd1` and `rd2` are tested.

This testbench must use assert statements to confirm proper functionality.

Lab Procedure

1. Create a new directory and Vivado project for this exercise.
2. Write a properly commented and properly formatted VHDL program according to the preceding specifications.
3. Test the design using a behavioral simulation.

4. Synthesize the design and test it using a Post-Synthesis Timing simulation.
5. Implement the design and test it using a Post-Implementation Timing simulation.
6. Prepare the design for hardware, following the procedure laid out in Exercise One.
7. Program the board and verify that it is working.
8. Demonstrate steps 3., 4., 5., and 7. to a TA or the instructor, who will sign the grading sheet.
9. Screen shot the appropriate waveforms for the report, making sure that they will be readable.
10. Submit the source code and report online to the appropriate myCourses dropbox.

Lab Report

Write a report that meets the rules of professional technical writing and follows the lab report format on myCourses. Additional items that must be included:

- A block diagram of the design, which is not hand drawn and is not taken from the RTL schematic.
- Marked simulation results for all simulations.

Exercise 2: Register File

Student's Name:________________________ Section:________________________

PreLab		Point Value	Points Earned	Comments
PreLab	Register File Schematic	10		

Demo		Point Value	Points Earned	Date
Demo	Behavioral Simulation	15		
	Post-Synthesis Timing Simulation	5		
	Schematic	5		
	Synthesis Report	5		
	Implementa-tion Report	5		
	Post-Implementation Timing Simulation	5		
	Hardware Demonstration	15		

To receive any grading credit students must earn points for both the demonstration and the report.

Exercise 2: Register File

Report		Point Value	Points Earned	Comments
Abstract		5		
Design Methodology	Block Diagram of Register File	3		
	Discussion of Register Module Functionality	3		
	Discussion of Register File Functionality	4		
Results & Analysis	Behavioral Waveform	3		
	Post-Implementation Waveform	3		
	Results Description	4		
Conclusion		5		
Source Code (Style)		5		
Total for prelab, demo, and report		100		

Exercise 3 Instruction Fetch Stage

Objective

In this exercise, students will be expected to design and create the Instruction Fetch stage of a pipelined MIPS architecture. This will require making several different modules and connecting them into a single Instruction Fetch stage. The Instruction Fetch stage is responsible for bringing the program instructions from memory into the processor.

Pre-lab Activities

- Describe in a brief paragraph the role of the Instruction Fetch stage in a MIPS processor.

- Create a block diagram of the Instruction Fetch stage with all signals labeled, including the internals of the Instruction Memory

Hardware Specification

Students will be creating the complete Instruction Fetch stage of the pipelined MIPS architecture, highlighted in Figure 3.2

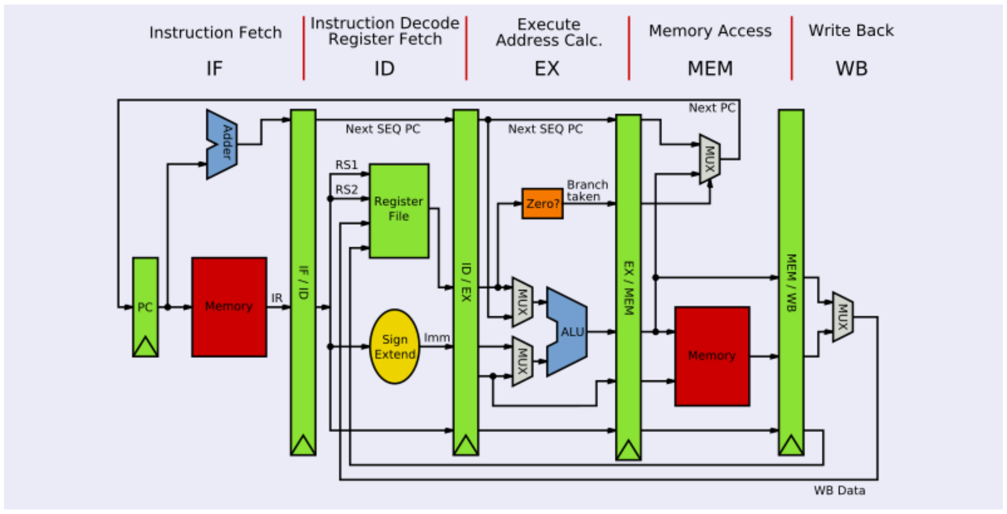


Figure 3.1: Pipelined MIPS Architecture

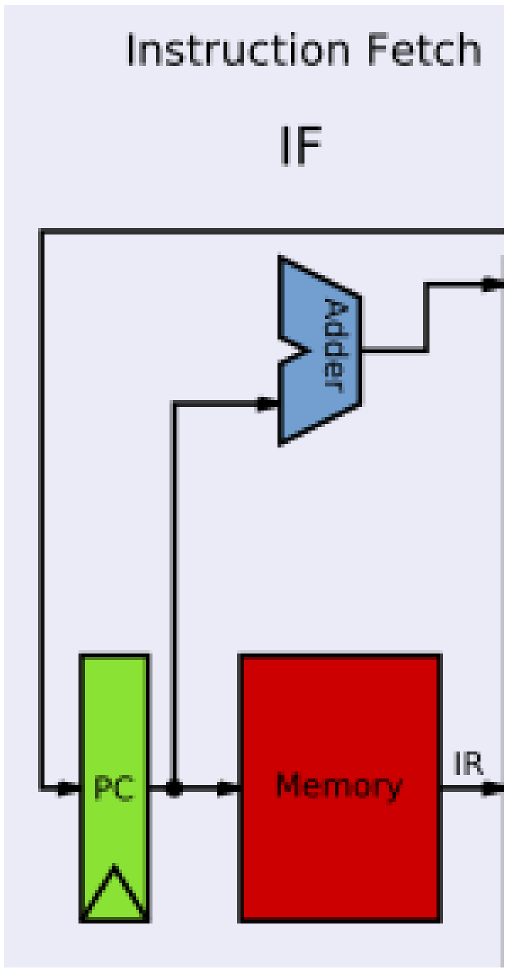


Figure 3.2: Pipelined MIPS Instruction Fetch

Instruction Memory

The Instruction Memory module will hold all of the instructions which will be fetched by the Instruction Fetch.

- Inputs (One bit unless noted otherwise)
  - `clk` The system clock
  - `addr` (28 bits) Address to read from
- Outputs (One bit unless noted otherwise)
  - `d_out` (32 bits) The instruction to be read from an instruction file at a specific address

Instruction Fetch

The Instruction Fetch module will incorporate the Instruction Memory module. The fetch will fetch an instruction from memory at the address determined by the program counter, which should increment and over time fetch all instructions from the memory.

- Inputs (One bit unless noted otherwise)
  - `clk` The system clock
  - `rst` Active high reset. Asynchronous.
  - `Jump` Active high bit which determines if the program is jumping
  - `JumpAddr` (28 bits) Address to jump to if the program is jumping
  - `PCSrc` (28 bits) Program Counter Source - the address that the Program Counter will start at
- Outputs (One bit unless noted otherwise)
  - `PCNext` (28 bits) Next Program Counter location
  - `Instruction` (32 bits) The instruction which was fetched from memory

Lab Procedure

1. Create a new Project in Xilinx Vivado for the exercise
2. Write a properly commented and formatted VHDL program according to the preceding specifications.
3. Write a testbench which properly tests the full stage. The testbench must use assert statements to be self-checking.
4. Test the design with a Behavioral Simulation. Make sure it includes the following:
   - PC Incrementation
   - Response to a Jump & behavior afterwards
5. Synthesize the design and test it using a Post-Synthesis Timing Simulation
6. Implement the design and test it using a Post-Implementation Timing Simulation
7. Program the board to verify its functionality using the provided constraints file
8. Demonstrate the above steps to a TA or Lab Instructor - ensure there are no warnings in the Synthesis or Implementation.
9. Screenshot the appropriate waveforms for the report, making sure they are readable
10. Submit the source code and report online to the appropriate MyCourses dropbox

Lab Report

Write a report that meets the rules of professional technical writing and follows the lab format on MyCourses. Additional items that must be included:

- A block diagram of the design, which is not hand drawn and is not taken from the RTL schematic
- Marked simulation results for all simulations

Exercise 3: Instruction Fetch Stage

Student's Name:____________________________ Section:____________________________

PreLab		Point Value	Points Earned	Comments
PreLab	Description of Instruction Fetch	10		
	Instruction Fetch/Memory Block Diagram	10		

Demo		Point Value	Points Earned	Date
Demo	Behavioral Simulation	10		
	No Warnings in Synthesis	5		
	Post-Synthesis Timing Simulation	5		
	Schematic	5		
	Synthesis Report	5		
	No Warnings in Implementation	5		
	Implementation Report	5		
	Post-Implementation Timing Simulation	5		

To receive any grading credit students must earn points for both the demonstration and the report.

Exercise 3: Instruction Fetch Stage

Report		Point Value	Points Earned	Comments
Abstract		5		
Design Methodology	Block Diagram of Instruction Fetch	4		
	Discussion of Instruction Memory Functionality	3		
	Discussion of Instruction Fetch Functionality	3		
Results & Analysis	Behavioral Waveform	3		
	Post-Implementation Waveform	3		
	Results Description	4		
Conclusion		5		
Source Code (Style)		5		
Total for prelab, demo, and report		100		

Exercise 4 Instruction Decode Stage

Objective

In this exercise, the students will be expected to design and create the Instruction Decode stage of a pipelined MIPS architecture. The Instruction Decode stage is responsible for parsing the instructions passed from the Instruction Fetch stage into commands which the Execute step can put into action.

Pre-lab Activities

- Describe in a brief paragraph the role of the Instruction Decode stage in a MIPS processor.
- Create a block diagram of the Instruction Decode stage with all signals labeled

Hardware Specification

Instruction Decode

The Instruction Decode stage takes a 32-bit op-code and decodes that code into full instructions.

- Inputs (One bit unless noted otherwise)
  - `Instruction` (32 bits) The op-code for the instruction being decoded
  - `RegDataA` (32 bits) Data from the first Register file
  - `RegDataB` (32 bits) Data from the second Register file
- Outputs (One bit unless noted otherwise)
  - `Jump` Bit determining whether or not to jump
  - `JumpAddr` (28 bits) Address to jump to
  - `ALUOp` (4 bits) Op-code specific to the ALU
  - `ValA` (32 bits) First value for the ALU to act on
  - `ValB` (32 bits) Second value for the ALU to act on
  - `MemWr` Bit to tell if we are writing to the Memory
  - `MemRd` Bit to tell if we are reading from Memory
  - `RegIdxA` (5 bits) Index to access in first Register
  - `RegIdxB` (5 bits) Index to access in second Register
  - `RegIdxWb` (5 bits) Register index for WriteBack, also functions as destination register
  - `RegEnWb` Bit that determines if a register is being written to

MIPS Instructions

The MIPS instruction set is made of a number of 32-bit instructions, which can be broken down by the Instruction Decode stage into smaller segments for different components to handle. There are three types of instructions you will be asked to handle, R-type (Register Type), I-type(Immediate Type), and J-type (Jump Type). Each has a slightly different format for the instructions.
The type of instruction is determined by the op-code of each instruction, which is always the first 6 bits.

Register Type Register Type instructions act upon 3 different registers, a source, target, and destination. The function, which is the final 6 bits, determines which ALU operation is to be executed with these registers. The shift amount (`sh_amt`) is the amount to shift by when using a shift command.
The Op-code for an R-Type instruction is "000000"

Table 4.1: R-Type

Op-code	rs	rt	rd	sh_amt	function
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Table 4.2: ALU Operations for R-Type

Function	Code
ADD	100000
AND	100100
MULTU	011001
OR	100101
SLL	000000
SRA	000011
SRL	000010
SUB	100011
XOR	100110

Immediate Type Immediate Type instructions are functions which act upon a number stored in a register and a 16 bit immediate. Unlike R-Type operations, the instruction op-code determines both that this is an I-Type instruction as well as what operation is to be performed.

Table 4.3: I-Type

Op-code	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Table 4.4: ALU Operations for I-Type

Function	Code
ADDI	001000
ANDI	001100
ORI	001101
XORI	001110
SW, ADD	001111
LW, ADD	100011

The `SW, ADD` and `LW, ADD` commands determine whether or not you are storing or loading a value as well as adding.

Jump Type Jump Type instructions instruct the processor to jump to a specified address.

Table 4.5: J-Type

Op-code	Addr
6 bits	26 bits

Table 4.6: Op-Codes for J-Type

Function	Code
Jump	000010
Jump	000011

Both op-codes for the jump command will behave the same way in this case. This is not always necessarily true, though.

ALU Codes The `ALUOp` output is only 4 bits; however, the part of the op-code which determines the ALU operation is 6 bits. Table 4.7 contains the 4 bit op-codes for the ALU which will be needed for R and I Type Instructions.

Table 4.7: 4-Bit ALU Op-Codes

Function	Code
ADD/ADDI/SW,ADD/LW,ADD	0100
AND/ANDI	1010
MULTU	0110
OR/ORI	1000
XOR/XORI	1011
SLL	1100
SRA	1110
SRL	1101
SUB	0101

Procedure

1. Create a new Project in Xilinx Vivado for the exercise
2. Write a properly commented and formatted VHDL program according to the preceding specifications.
3. Write a testbench which properly tests the stage. The testbench must use assert statements to be self-checking
4. Test the design with a Behavioral Simulation
5. Synthesize the design and test it using a Post-Synthesis Timing Simulation
6. Implement the design and test it using a Post-Implementation Timing Simulation
7. Demonstrate the above steps to a TA or Lab Instructor
8. Screenshot the appropriate waveforms for the report, making sure they are readable
9. Submit the source code and report online to the appropriate MyCourses dropbox

Lab Report

Write a report that meets the rules of professional technical writing and follows the lab format on MyCourses. Additional items that must be included:

- A block diagram of the design, which is not hand drawn and is not taken from the RTL schematic
- Area used, which is reported via number of occupied slices, FFs used, and LUTs used
- Marked simulation results

Exercise 4: Instruction Decode Stage

Student's Name:____________________ Section:____________________

PreLab		Point Value	Points Earned	Comments
PreLab	Description of Instruction Decode	10		
	Instruction Decode Block Diagram	10		

Demo		Point Value	Points Earned	Date
Demo	Behavioral Simulation	15		
	Post-Synthesis Timing Simulation	5		
	Schematic	5		
	Synthesis Report	5		
	Implementation Report	5		
	Post-Implementation Timing Simulation	5		

To receive any grading credit students must earn points for both the demonstration and the report.

Exercise 4: Instruction Decode Stage

Report		Point Value	Points Earned	Comments
Abstract		5		
Design Methodology	Instruction Decode Block Diagram	5		
	Discussion of Functionality	8		
Results & Analysis	Behavioral Simulation	3		
	Post-Implementation Timing Simulation	3		
	Discussion of Results	6		
Conclusion		5		
Source Code (Style)		5		
Total for prelab, demo, and report		100		

Exercise 5 Execute Stage

Objective

In a MIPS processor, once an instruction has been fetched from memory and decoded it needs to be executed. The execute stage consists of an ALU and several multiplexors, which determine how the instruction being passed in is executed. For the purposes of this design, these multiplexors will not need to be explicitly created.
First, in this exercise, the ALU that was begun in an earlier exercise will need to be completed with a Ripple-Carry full adder/subtractor and a Carry-Save Multiplier. Next, once the ALU has all of the required functionality, the complete execute stage must be created so that it can properly interface with the previous stages.

Pre-lab Activities

- Draw a block diagram of the complete ALU, including the Ripple-Carry Adder/Subtractor and Carry-Save Multiplier.
- Become familiar with the functionality of a multiplier. Consider what might need to be modified to make it a generic size.
- Print the sign of sheet.

Hardware Specification

ALU

The full ALU has several different components. Most of these were created in previous exercises; however, there are two more components which need to be added for full functionality: the adder/-subtractor and the multiplier.
Once all operations have been created, Table 5.1 should be used to determine which operation is being performed.

Table 5.1: 4-Bit ALU Op-Codes

Function	Code
ADD/ADDI/SW,ADD/LW,ADD	0100
AND/ANDI	1010
MULTU	0110
OR/ORI	1000
XOR/XORI	1011
SLL	1100
SRA	1110
SRL	1101
SUB	0101

- Inputs (One bit unless noted otherwise)
  - `in1` (n bits) First input to the ALU

- `in2` (n bits) Second input to the ALU
- `control` (4 bits) The ALU Op-code

- Outputs (One bit unless noted otherwise)
  - `out1` (n bits) Output of the ALU

Ripple Carry Full Adder The full adder has the following features:

- Functions:
  - Multi-bit addition.
  - Multi-bit subtraction. Thus all multi-bit inputs and outputs are in two's complement.
- Parameters:
  - The number of bits of the two addends. To be called `n`.
- Inputs:
  - `A` (`n` bits): First addend, or minuend if subtracting
  - `B` (`n` bits): Second addend, or subtrahend if subtracting
- Outputs:
  - `Sum` (`n` bits): Sum, or difference if subtracting

The ripple carry full adder must be structural, however, the full adders it is made up of can be behavioral, dataflow, or structural. Use the for generate syntax to instantiate the individual full adders inside of the ripple carry adder.

Carry Save Multiplier The multiplier has the following features:

- Functions:
  - Multi-bit multiplication.
- Parameters:
  - The output length, in bits. To be called `n`.
- Inputs:
  - `A` (`n`/2 bits): First factor.
  - `B` (`n`/2 bits): Second factor.
- Outputs:
  - `Product` (n bits): Product. Note that this is twice as long as the factors.

The carry save multiplier must be structural, however, the components of the multiplier do not have to be. Figure 5.1 shows a four-bit carry save multiplier. Note that the rightmost adder could be a half adder, as the carry in is set to zero. This multiplier functions in a similar way to humans performing multiplication with two decimal numbers on paper. In order to implement such a mutliplier, consider using an array of `std_logic_vectors` to hold the intermediate results such as the results of the AND operation, sums, and carry outs. The use of the `+` and `*` operators is not allowed.

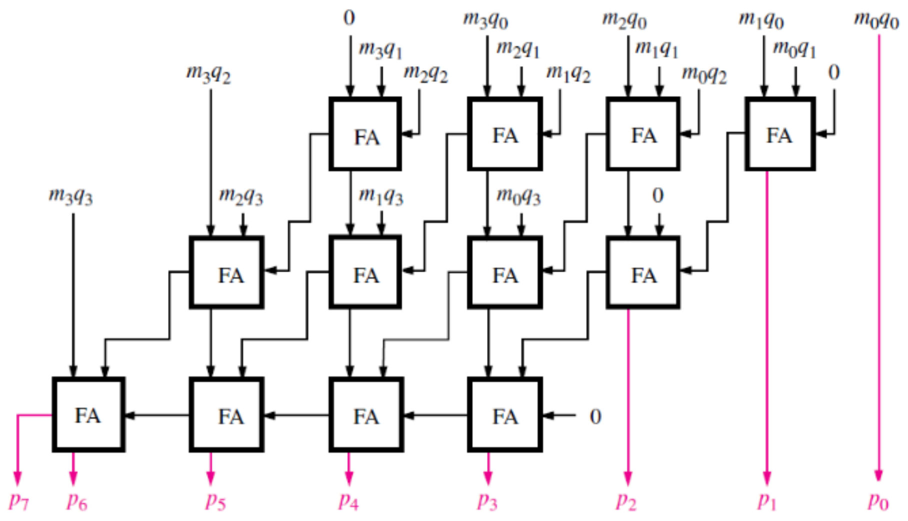


Figure 5.1: 4-bit Carry Save Multiplier

Execute Stage

The execute stage of the MIPS processor takes the outputs from the Instruction Decode stage and executes the appropriate instruction. A major part of this stage is the ALU - however, there can be other components involved, depending on the design.

- Inputs (One bit unless noted otherwise)
  - `ALUOp` (4 bits) Op-code specific to the ALU
  - `IdExA` (32 bits) First value for the ALU to act on
  - `IdExB` (32 bits) Second value for the ALU to act on, or - when not performing an ALU Operation - it is the data which interacts with the Memory stage.
  - `IdExWbIdx` (5 bits) Index for Memory WriteBack
  - `IdExWbEn` Memory WriteBack Enable bit
  - `IdExMemRd` Memory read bit
  - `IdExMemWr` Memory write bit
- Outputs (One bit unless noted otherwise)
  - `MemWrData` (32 bits) Data to be written to the memory stage

  - `ALUResult` (32 bits) Result of the ALU Operation
  - `ExMemWbIdx` (5 bits) Memory WriteBack Index
  - `ExMemWbEn` Memory WriteBack Enable bit
  - `ExMemRd` Memory read bit
  - `ExMemWr` Memory write bit

Procedure

Part 1 - Ripple-Carry Adder & Execute Stage

1. Create a new Project in Xilinx Vivado for the exercise
2. Write a properly commented and formatted VHDL program according to the preceding specifications.
3. Write a testbench which properly tests the ALU **except the Carry-Save Multiplier**
4. Test the design with a Behavioral Simulation
5. Synthesize the design and test it using a Post-Synthesis Timing Simulation
6. Implement the design and test it using a Post-Implementation Timing Simulation
7. Demonstrate the above steps to a TA or Lab Instructor
8. Reduce the number of bits for input to 4, and program the Basys3
9. Demonstrate the proper functionality to a TA or the Lab Instructor.
10. Screenshot the appropriate waveforms for the report, making sure they are readable
11. Create the Execute Stage for the MIPS Datapath - make the the ALU is the full size when adding it to the stage
12. Write a testbench which properly tests the current functionality of the Execute Stage
13. Test the design with a Behavioral Simulation
    - It is not required to synthesize or implement the design at this stage, though it is recommended to ensure the correct functionality

This must be completed within the first week of the lab

Part 2 - Carry-Save Multiplier

1. Create the Carry-Save Multiplier
2. Write a testbench which properly tests the functionality of the Multiplier
3. Test the design with a Behavioral Simulation
4. Add the Multiplier to the rest of the ALU

5. Test the completed design with a Behavioral Simulation by modifying the testbench created for the ALU in Part 1
6. Synthesize the design and test it using a Post-Synthesis Timing Simulation
7. Implement the design and test it using a Post-Implementation Timing Simulation
8. Behaviorally test the completed Execute Stage
9. Demonstrate the above steps to a TA or Lab Instructor
10. Screenshot the appropriate waveforms for the report, making sure they are readable
11. Submit the source code and report online to the appropriate MyCourses dropbox

Lab Report

Write a report that meets the rules of professional technical writing and follows the lab format on MyCourses. Additional items that must be included:

- A block diagram of the design, which is not hand drawn and is not taken from the RTL schematic
- Area used, which is reported via number of occupied slices, FFs used, and LUTs used
- Marked simulation results

Exercise 5: Execute Stage

Student's Name:_____________________________ Section:_____________________________

PreLab		Point Value	Points Earned	Comments
PreLab	ALU Block Diagram	10		

Demo		Point Value	Points Earned	Date
Part 1 – Adder& Execut e Stage	ALU Behavioral Simulation	10		
	ALU Post-Synthesis Timing Simulation	5		
	ALU Post-Implementation Timing Simulation	5		
	Hardware Demonstration	8		
	Execute Stage Behavioral Simulation	2		
Part 2 – Multiplier	Multiplier Behavioral Simulation	8		
	Full ALU Behavioral Simulation	2		
	Post-Synthesis Timing Simulation	5		
	Post-Implementation Timing Simulation	5		
	Completed Execute Stage Behavioral Simulation	5		

To receive any grading credit students must earn points for both the demonstration and the report.

Exercise 5: Execute Stage

Report		Point Value	Points Earned	Comments
Abstract		5		
Design Methodology	ALU and Execute Block Diagrams	5		
	Discussion of Functionality	5		
Results & Analysis	ALU and Execute Behavioral Simulations	5		
	Post-Implementation Timing Simulations	3		
	Discussion of Results	2		
Conclusion		5		
Source Code (Style)		5		
Total for prelab, demo, and report		100		

Exercise 6 Memory Stage

Objective

In this exercise, the Memory stage of the MIPS Datapath will be designed and implemented. The purpose of this stage is to offer memory to the system which can be used for storing and loading - depending on the command.

Pre-lab Activities

- Make a block diagram of the complete Memory Stage - including all signals to and from other stages
- Review the use of assert statements in a Testbench

Hardware Specification

Data Memory

This memory should be very similar to the Instruction Memory used in the Instruction Fetch stage. The values in the memory should be initialized, but this memory should be writable.

- Generics:
  - `mem_size` The size of the memory being used := 1024
- Inputs:
  - `clk` Clock for the memory
  - `w_en` Write Enable for writing to Memory
  - `addr` (28 bits) Address to write data to and read data from
  - `d_in` (32 bits) Data to be written to Memory
- Outputs:
  - `d_out` (32 bits) Data being read from `addr`
- Initial Values - Initialize the memory values at the following addresses to their corresponding numbers. If an initial value isn't given it can be set to any valid value.

Table 6.1: Initial Values for Memory

Address	**Value**
0	`X"ABCDDCBA"`
137	`X"C0FFEE12"`
489	`X"1A2B3C4D"`
512	`X"8F8F8F8F"`
750	`X"FEDCBA98"`
1023	`X"FFFF0000"`

Memory Stage

The total Memory Stage for the MIPS Datapath. This should contain an instance of the Data Memory and use a number of signals from previous stages to act on the memory.

- Inputs:
  - `clk` System Clock
  - `ALUResult` (32 bits) The result of the ALU from the Execute Stage
  - `MemWrData` (32 bits) Data to be written to the Data Memory
  - `ExMemWr` Memory Write bit from Execute Stage. Passes through to Writeback Stage
  - `ExMemRd` Memory Read bit from Execute Stage. Passes through to Writeback Stage
  - `ExMemWbIdx` (5 bits) Writeback index from Execute Stage. Passes through to Writeback Stage
- Outputs:
  - `RegData` (32 bits) Register Data from Execute Stage.
  - `MemData` (32 bits) Data retrieved from Data Memory
  - `MemWr` Memory Write Bit
  - `MemRd` Memory Read Bit
  - `WbIdx` (5 bits) Writeback index

Testbench

A testbench much be written which tests the following things:

- Reading the addresses corresponding with the initial values given in Table 6.1.
- Writing the values `X"01010101"` and `X"55001122"` to the addresses 18 and 256 respectively

For this testbench, assert statements must be used to confirm that cases stated above work as expected.

Procedure

1. Create a new Project in Xilinx Vivado for the exercise
2. Write a properly commented and formatted VHDL program according to the preceding specifications.
3. Write a testbench which properly tests the stage
4. Test the design with a Behavioral Simulation
5. Synthesize the design and test it using a Post-Synthesis Timing Simulation
6. Implement the design and test it using a Post-Implementation Timing Simulation

7. Demonstrate the above steps to a TA or Lab Instructor
8. Screenshot the appropriate waveforms for the report, making sure they are readable
9. Submit the source code and report online to the appropriate MyCourses dropbox

Lab Report

Write a report that meets the rules of professional technical writing and follows the lab format on MyCourses. Additional items that must be included:

- A block diagram of the design, which is not hand drawn and is not taken from the RTL schematic
- Area used, which is reported via number of occupied slices, FFs used, and LUTs used
- Marked simulation results

Exercise 6: Memory Stage

Student's Name:____________________ Section:____________________

PreLab		Point Value	Points Earned	Comments
PreLab	Memory Stage Block Diagram	10		

Demo		Point Value	Points Earned	Date
Demo	Behavioral Simulation	15		
	Post-Synthesis Timing Simulation	7		
	Schematic	6		
	Synthesis Report	5		
	Implementation Report	5		
	Post-Implementation Timing Simulation	8		

To receive any grading credit students must earn points for both the demonstration and the report.

Exercise 6: Memory Stage

Report		Point Value	Points Earned	Comments
Abstract		5		
Design Methodology	Memory Stage Block Diagram	5		
	Discussion of Functionality	8		
Results & Analysis	Behavioral Simulation	5		
	Post-Implementation Timing Simulation	5		
	Discussion of Results	6		
Conclusion		5		
Source Code (Style)		5		
Total for prelab, demo, and report		100		

Exercise 7 Writeback Stage

Objective

Pre-lab Activities

- Make a block diagram of the complete Writeback Stage - include where are signals are going back to

Hardware Specification

Writeback Stage

- Inputs:
  - `MemWbIdx` (5 bits) Writeback index
  - `MemWr` Bit that determines whether or not to write in the writeback
  - `MemRd` Bit that determines whether or not data was read from memory
  - `RegData` (32 bits) Data passed in from a register. Used when not reading from memory.
  - `MemData` (32 bits) Data passed in when read from memory. Gets written back if MemRd is '1'
- Outputs:
  - `WbData` (32 bits) Data to be written back, either from Memory or from a Register
  - `WbIdx` (5 bits) Writeback index
  - `WbEn` Bit which determines if anything is written back. Nothing is written back when writing to memory.

Procedure

1. Create a new Project in Xilinx Vivado for the exercise
2. Write a properly commented and formatted VHDL program according to the preceding specifications.
3. Write a testbench which properly tests the stage
4. Test the design with a Behavioral Simulation
5. Synthesize the design and test it using a Post-Synthesis Timing Simulation
6. Implement the design and test it using a Post-Implementation Timing Simulation
7. Demonstrate the above steps to a TA or Lab Instructor
8. Screenshot the appropriate waveforms for the report, making sure they are readable
9. Submit the source code and report online to the appropriate MyCourses dropbox

Lab Report

Write a report that meets the rules of professional technical writing and follows the lab format on MyCourses. Additional items that must be included:

- A block diagram of the design, which is not hand drawn and is not taken from the RTL schematic
- Area used, which is reported via number of occupied slices, FFs used, and LUTs used
- Marked simulation results

Exercise 7: Writeback Stage

Student's Name:____________________ Section:____________________

PreLab		Point Value	Points Earned	Comments
PreLab	Writeback Stage Block Diagram	10		

Demo		Point Value	Points Earned	Date
Demo	Behavioral Simulation	15		
	Post-Synthesis Timing Simulation	7		
	Schematic	6		
	Synthesis Report	5		
	Implementation Report	5		
	Post-Implementation Timing Simulation	8		

To receive any grading credit students must earn points for both the demonstration and the report.

Exercise 7: Writeback Stage

Report		Point Value	Points Earned	Comments
Abstract		5		
Design Methodology	Writeback Stage Block Diagram	5		
	Discussion of Functionality	8		
Results & Analysis	Behavioral Simulation	5		
	Post-Implementation Timing Simulation	5		
	Discussion of Results	6		
Conclusion		5		
Source Code (Style)		5		
Total for prelab, demo, and report		100		

Appendix A Xilinx Vivado Tutorial

Objective

The following is a tutorial on how to create and test a simple VHDL design in the Xilinx Vivado Design Suite. Vivado is a comprehensive software suite for FPGA development. It includes a source code editor, synthesis tools, implementation tools, a simulator, and tools for loading a design onto FPGAs.

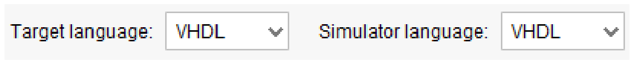
Vivado WebPack edition is available for students, free of charge, as explained here.

- If you install Vivado to your personal computer, you will likely not be able to select the board that we will be using for these labs. Follow the instructions here to add them: `https://reference.digilentinc.com/reference/software/vivado/board-files`

Design

Here, we walk through the process of creating a Vivado project and design files. For this example, we will create a dataflow design for an 4-bit partial arithmetic logic unit (ALU) capable of NOT and Logical Left Shift (SLL) operations.

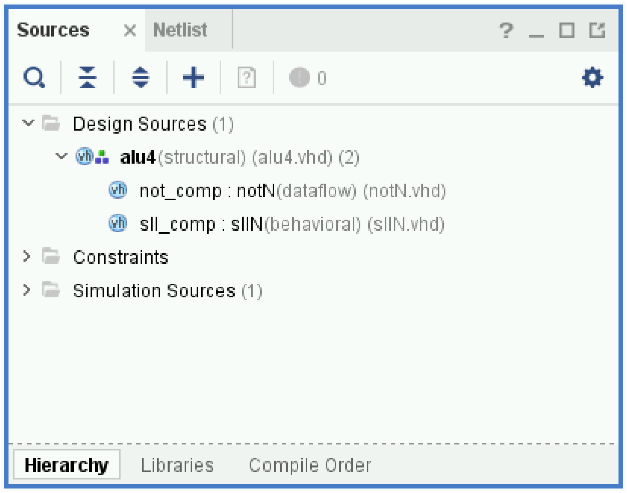
1. Project Creation Wizard
   (a) Create a new project from either the "Quick Start" or "File" menu. This will launch the project creation wizard. Click Next to get started.
   (b) **Project Name:** Give the project a name of your choice and select a location with fast storage (local disk is recommended). Click Next.
   (c) **Project Type:** Select "RTL Project." Click Next.
   (d) **Add Sources:**
      (i) At the bottom of the window, select VHDL as your target language and simulator language.



Target language: VHDL   Simulator language: VHDL

      (ii) For each of the following components, click **Create File**, specify the file name, and click OK.
         (1) notN.vhd
         (2) sllN.vhd
         (3) alu4.vhd
      (iii) Click Next to continue
   (e) **Add Constraints:** Skip this step for now. Click Next.
   (f) **Default Part:** Click Board, then select Basys3, then click Next.
   (g) Click Finish to complete the project creation wizard.
2. **Define Modules:** This will pop-up when your project opens. It provides a GUI to automatically generate entity declarations. We want to do this ourselves, so skip it by clicking OK. If you get a warning about unchanged files, click Yes to ignore it.

3. In the **Sources** panel, double-click on **notN.vhd** to open it. Copy the source code from Appendix B and save. Repeat this process for **sllN.vhd** and **alu4.vhd** with Appendix C and Appendix D respectively. At the end of this process, your **Sources** panel should look like this: *Note: Copying from the pdf file can cause spacing issues and hidden characters. If you run into problems that seemingly don't make sense, you may need to type out the file. One way to mitigate this is by downloading the manual and using something other than a web-browser to view it.*



```
Design Sources (1)
    alu4(structural) (alu4.vhd) (2)
        not_comp : notN(dataflow) (notN.vhd)
        sll_comp : sllN(behavioral) (sllN.vhd)
Constraints
Simulation Sources (1)
```

The tree now represents your hierarchical/structural design. This is helpful in understanding the architecture of a design at a glance. However, if you ever want to see a flat list of files in the project, click **Compile Order** along the bottom of the **Sources** panel.

Behavioral Simulation

A crucial step in the design process is testing whether or not your design provides the expected functionality. The simplest way to do this is by creating a VHDL test bench and running a behavioral simulation. This is an ideal simulation and does not account for any actual hardware. However, it is the fastest type of simulation to run and the results are easy to read and parse.

1. **Creating a Simulation Source**

   (a) From the **Sources** panel, click +. This will launch the **Add Sources** wizard once again.

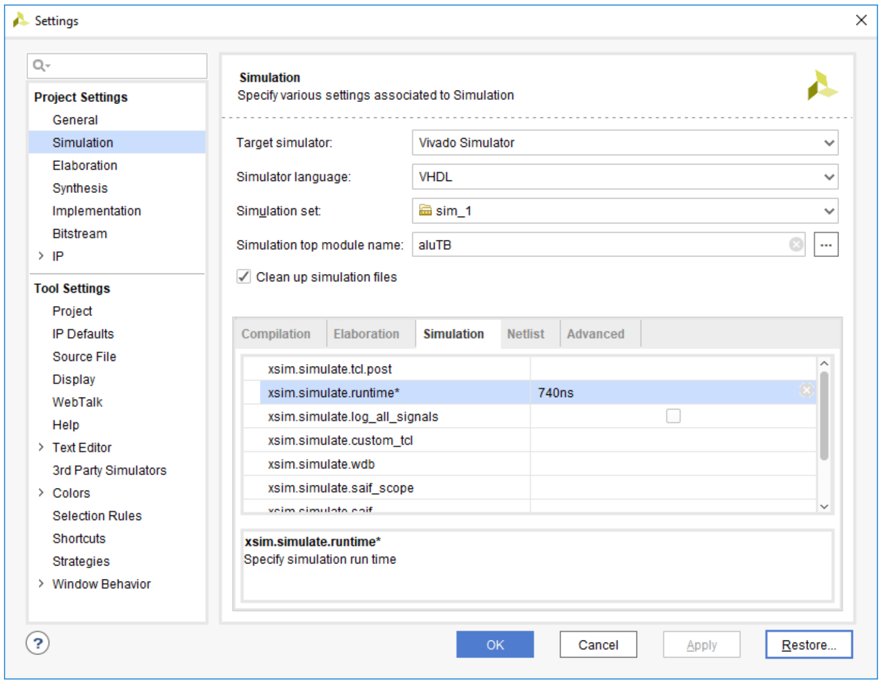
   (b) Choose **Add or create simulation sources**. Click Next.

   (c) Click **Create File**. Name it **aluTB** and click**OK**. Then, from the wizard, click Finish.

2. **Define Module:** Once again, this wizard will pop up. Again, we can skip it by clicking OK and then Yes.

3. From the sources panel, under **Simulation Sources** > **sim_1**, double-click on **aluTB.vhd** to open it. Copy the source code from Appendix E and save your file.

4. To specify the length of the simulation, go to the Flow Navigator and right click on SIMULATION. Click "Simulation Settings...". Under the **Simulation** tab, set the value of **xsim.simulate.runtime** to **740ns**, like so:



Simulation
Specify various settings associated to Simulation

Target simulator: Vivado Simulator
Simulator language: VHDL
Simulation set: sim_1
Simulation top module name: aluTB
☑ Clean up simulation files

xsim.simulate.tcl.post	
xsim.simulate.runtime*	740ns
xsim.simulate.log_all_signals	☐
xsim.simulate.custom_tcl	
xsim.simulate.wdb	
xsim.simulate.saif_scope	

xsim.simulate.runtime*
Specify simulation run time

Click **Apply**, then click **OK**.

5. From the Flow Navigator, under **SIMULATION**, click **Run Simulation**. Then click **Run Behavioral Simulation**.

6. Vivado should show a waveform of the simulation to to 200ns. Behavioral simulations are logical only. As such, no design is synthesized and no hardware delays are incurred. Check to make sure that all combinations of operations and operands produce correct results.

Synthesis & Post-Synthesis Simulation

Synthesis "compiles" your design into a gate-level netlist, represented by the UNISIM library, a Xilinx library containing basic primitives. In this step, we will synthesize ALU design, check the synthesis reports, view the register transfer level (RTL) diagram, and run a Post-Synthesis simulation.

1. From the **Flow Navigator**, under **SYNTHESIS**, click **Run Synthesis**, then click **OK**. The status indicator in the top-right of your screen should indicate that this process is running. This could take a few minutes. Wait until the status indicator displays **Synthesis Complete**.

2. To simulate this model, go to **Flow Navigator** > **SIMULATION** > **Run Simulation** > **Run Post-Synthesis Timing Simulation**. If given a warning, click **Yes**. The waveforms in this simulation should include delays incurred by the gates in the model. As such, there should be setup times for some signals and signals may change value multiple times before becoming stable.

3. One debugging technique is to examine the RTL diagram of a design. To do this, under **Flow Navigator** > **SYNTHESIS** > **Open Synthesized Design**, click **Schematic**. The RTL schematic can be a useful debugging tool, as it shows the connections between internal components of the synthesized design.

4. One can determine the resource utilization under **SYNTHESIS** > **Report Utilization**. Determine how many Slice LUTs are used in this design.

Errors in this type of simulation stage are often (but not always) the result of a latch or some other design flaw within a process.

NOTE: A functional simulation also tests the gate-level model, but does not account for delays. The upside is that, due to predictable timing, it is easier to automate testing.

Implementation & Post-Implementation Simulation

Implementation places and routes your synthesized design into a model of the FPGA, allowing you to simulate how your design will behave on the actual hardware.

1. From the **Flow Navigator**, under **IMPLEMENTATION**, click **Run Implementation**, then click **OK**. The status indicator in the top-right of your screen should indicate that this process is running. This could take a few minutes. Wait until the status indicator displays **Implementation Complete**.

2. As with Synthesis, Implementation also generates a schematic. View it under **Flow Navigator** > **IMPLEMENTATION** > **Open Implemented Design** > **Schematic.**

3. To view the resource utilization of the implemented model, go to **Flow Navigator** > **IMPLEMENTATION** > **Open Implemented Design** > **Report Utilization**. Note the number of slice LUTS being used. Also note the number and types of slices being used.

4. To simulate this model, go to **Flow Navigator** > **SIMULATION** > **Run Simulation** > **Run Post-Implementation Timing Simulation**. If given a warning, click **Yes**. The waveforms in this simulation should include delays incurred by the gates in the model. As such, there should be setup times for some signals and signals may change value multiple times before becoming stable.

Again, errors in this type of simulation stage are often (but not always) the result of a latch or some other design flaw within a process.

NOTE: A functional simulation also tests the implemented model, but does not account for delays. The upside is that, due to predictable timing, it is easier to automate testing.

Hardware

In this step, we will load the partial ALU design onto a Basys3 FPGA development kit. To do this, we will generate a "bitstream" file, which contains the necessary information for programming the target device with the design.

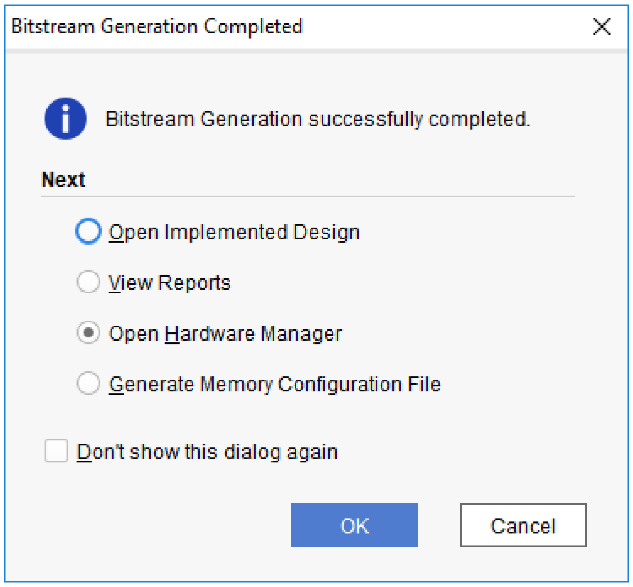
1. Create Constraints

   (a) In order to map ports on the top-level design to pins on the FPGA, a Xilinx Design Constraints (XDC) file is required. To create one, click + in the **Sources** panel.

   (b) Click **Add or create constraints**, then click **Next**.
   (c) Click **Create File**, enter "adc4" for the **File name**, click **OK**, then click **Finish**.
   (d) Copy the contents of Appendix F to adc4.xdc. This will map SW3 through SW0 to Port B, SW7 through SW4 to Port A, and SW15 to Port OP.
   (e) Save the file

2. To generate the bitstream file, go to the **Flow Navigator** > **PROGRAM AND DEBUG**, and then click **Generate Bitstream** and wait for that process to complete. You may be asked to if you'd like to re-run the Synthesis and Implementation steps. If that dialog appears, hit **Yes**.

3. After some time, the following message box should pop up:



Bitstream Generation Completed

Bitstream Generation successfully completed.

Next

- ○ Open Implemented Design
- ○ View Reports
- ● Open Hardware Manager
- ○ Generate Memory Configuration File

☐ Don't show this dialog again

   Click **OK** to open the **Hardware Manager**.

4. Now connect your Basys3 to your workstation using a microUSB cable.

5. Flip the power switch SW16 up to turn on the device. Windows may take a few minutes to install drivers for the device.

6. Near the top of the screen, you should see a green bar with the link **Open target**. Click that link, then click **Auto Connect**.

7. The green bar should then show a link to **Program device**. Click it, then click **Program**.

8. At this point, the ALU design should be loaded onto the device. Test it by changing inputs using the aforementioned switches. The output of the ALU is displayed via the four rightmost LEDs.

9. Close **Hardware Manager**

10. Power off the Basys3 using SW16 and disconnect it from your workstation.

Appendix B notN Source Code

```
---------------------------------------------------------------------------------
-- Company:  Rochester Institute of Technology (RIT)
-- Engineer: <YOUR_NAME_HERE> (<YOUR_EMAIL_HERE>)
--
-- Create Date:    <CREATION_TIME_HERE>
-- Design Name:    notN
-- Module Name:    notN - dataflow
-- Project Name:   <PROJECT_NAME_HERE>
-- Target Devices: Basys3
--
-- Description: N-bit bitwise NOT unit
---------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity notN is
    GENERIC (N : INTEGER := 4); --bit width
    PORT (
            A : IN std_logic_vector(N-1 downto 0);
            Y : OUT std_logic_vector(N-1 downto 0)
        );
end notN;

architecture dataflow of notN is
begin
    Y <= not A;
end dataflow;
```

Appendix C sllN Source Code

```
--------------------------------------------------------------------------------
-- Company:  Rochester Institute of Technology (RIT)
-- Engineer: <YOUR_NAME_HERE> (<YOUR_EMAIL_HERE>)
--
-- Create Date:    <CREATION_TIME_HERE>
-- Design Name:    sllN
-- Module Name:    sllN - behavioral
-- Project Name:   <PROJECT_NAME_HERE>
-- Target Devices: Basys3
--
-- Description: N-bit logical left shift (SLL) unit
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity sllN is
    GENERIC (N : INTEGER := 4); --bit width
    PORT (
            A         : IN std_logic_vector(N-1 downto 0);
            SHIFT_AMT : IN std_logic_vector(N-1 downto 0);
            Y         : OUT std_logic_vector(N-1 downto 0)
        );
end sllN;

architecture behavioral of sllN is
begin
    process(A, SHIFT_AMT) is
        variable int_shamt : integer;
    begin
        int_shamt := to_integer(unsigned(SHIFT_AMT));

        for i in N-1 downto 0 loop
            if (i - int_shamt >= 0) then
                Y(i) <= A(i - int_shamt);
            else
                Y(i) <= '0';
            end if;
        end loop;
    end process;
end behavioral;
```

Appendix D alu4 Source Code

```
----------------------------------------------------------------------------------
-- Company:  Rochester Institute of Technology (RIT)
-- Engineer: <YOUR_NAME_HERE> (<YOUR_EMAIL_HERE>)
--
-- Create Date:    <CREATION_TIME_HERE>
-- Design Name:    alu4
-- Module Name:    alu4 - structural
-- Project Name:   <PROJECT_NAME_HERE>
-- Target Devices: Basys3
--
-- Description: Partial 4-bit Arithmetic Logic Unit
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity alu4 is
    GENERIC (N : INTEGER := 4); --bit width
    PORT (
            A   : IN std_logic_vector(N-1 downto 0);
            B   : IN std_logic_vector(N-1 downto 0);
            OP  : IN std_logic;
            Y   : OUT std_logic_vector(N-1 downto 0)
        );
end alu4;

architecture structural of alu4 is
-- Declare teh inverter component
    Component notN is
        GENERIC ( N : INTEGER := 4); -- bit width
        PORT (
            A : IN std_logic_vector(N-1 downto 0);
            Y : OUT std_logic_vector(N-1 downto 0)
        );
    end Component;

-- Declare the shift left component
    Component sllN is
        GENERIC (N : INTEGER := 4); --bit width
        PORT (
            A         : IN std_logic_vector(N-1 downto 0);
            SHIFT_AMT : IN std_logic_vector(N-1 downto 0);
            Y         : OUT std_logic_vector(N-1 downto 0)
        );
    end Component;

    signal not_result : std_logic_vector(3 downto 0);
    signal sll_result : std_logic_vector(3 downto 0);

begin
```

```
    -- Instantiate the inverter
    not_comp: notN
        generic map ( N => 4)
        port map ( A => A, Y => not_result );

    -- Instantiate the SLL unit
    sll_comp: sllN
        generic map ( N => 4 )
        port map ( A=> A, SHIFT_AMT => B, Y => sll_result );

    -- Use OP to control which operation to show/perform
    Y <= not_result when OP = '0' else  -- NOT
         sll_result;                    -- SLL
end structural;
```

Appendix E aluTB Source Code

```
----------------------------------------------------------------------------------
-- Company:  Rochester Institute of Technology (RIT)
-- Engineer: <YOUR_NAME_HERE> (<YOUR_EMAIL_HERE>)
--
-- Create Date:     <CREATION_TIME_HERE>
-- Design Name:     aluTB
-- Module Name:     aluTB - behavioral
-- Project Name:    <PROJECT_NAME_HERE>
--
-- Description: Testbec\nch for Partial 32-bit ALU
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity aluTB is
end aluTB;

architecture Behavioral of aluTB is
--Declare the ALU component
    Component alu4 is
        PORT (
            A    : IN std_logic_vector(3 downto 0);
            B    : IN std_logic_vector(3 downto 0);
            OP   : IN std_logic;
            Y    : OUT std_logic_vector(3 downto 0)
        );
    end Component;

    constant delay : time := 20 ns;
    signal A, B, Y : std_logic_vector(3 downto 0) := (others => '0');
    signal OP      : std_logic := '0';

begin
    -- Instantiate an instance of the ALU
    alu_inst: alu4 PORT MAP (
        A => A,
        B => B,
        OP => OP,
        Y => Y
    );

    data_proc: process is
    begin
        for i in 0 to 7 loop
            wait for delay;
            A <= std_logic_vector(unsigned(A) + 1);
        end loop;

        wait for delay;
        OP <= '1';

        for i in 1 to 7 loop
            A <= std_logic_vector(unsigned(A) + 1);
            for j in 0 to 3 loop
                wait for delay;
                B <= std_logic_vector((unsigned(B) + 1) mod 4);
            end loop;
        end loop;

        wait;

    end process;
end Behavioral;
```

Appendix F Xilinx Design Constraints File

```
## Switches
set_property PACKAGE_PIN V17 [get_ports {B[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {B[0]}]
set_property PACKAGE_PIN V16 [get_ports {B[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {B[1]}]
set_property PACKAGE_PIN W16 [get_ports {B[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {B[2]}]
set_property PACKAGE_PIN W17 [get_ports {B[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {B[3]}]
set_property PACKAGE_PIN W15 [get_ports {A[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {A[0]}]
set_property PACKAGE_PIN V15 [get_ports {A[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {A[1]}]
set_property PACKAGE_PIN W14 [get_ports {A[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {A[2]}]
set_property PACKAGE_PIN W13 [get_ports {A[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {A[3]}]

set_property PACKAGE_PIN R2 [get_ports {OP}]
    set_property IOSTANDARD LVCMOS33 [get_ports {OP}]

## LEDs
set_property PACKAGE_PIN U16 [get_ports {Y[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Y[0]}]
set_property PACKAGE_PIN E19 [get_ports {Y[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Y[1]}]
set_property PACKAGE_PIN U19 [get_ports {Y[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Y[2]}]
set_property PACKAGE_PIN V19 [get_ports {Y[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Y[3]}]
set_property PACKAGE_PIN W18 [get_ports {Y[4]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Y[4]}]
set_property PACKAGE_PIN U15 [get_ports {Y[5]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Y[5]}]
set_property PACKAGE_PIN U14 [get_ports {Y[6]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Y[6]}]
set_property PACKAGE_PIN V14 [get_ports {Y[7]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Y[7]}]

## Configuration options, can be used for all designs
set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]
```

Notes