Carnegie Mellon University
18-347 Introduction to Computer Architecture

Lab 2: Computer Arithmetic and ALUs

Due: The week of September 29, 2003 prior to the start of Lab
(100 points, may be done in groups of two)

Objective

In this lab you will implement the internals of an Arithmetic Logic Unit (ALU). You will see the trade-offs in speed and area between two types of adders and understand how arithmetic and logic circuits, along with their associated condition codes, are built.

Introduction

One of the main tasks for a microprocessor is to steer instructions into Arithmetic Logic Units (ALUs). The ALUs (and other important parts of the processor, such as the next PC logic) need to have fast and efficient adders, because the results of addition operations are used in virtually every cycle. In aggressive processors, such as the Pentium 4, there are even adders which produce certain results every half clock cycle!

For this lab, you will first implement a simple behavioral Verilog model for the ALU, then you will structurally implement logical operators, a barrel shifter, and two versions of the adder. The specifics of the MIPS ALU are described in Chapter 4 of Patterson and Hennessy's Computer Organization and Design and in your lecture notes.

Part 1: The Behavioral ALU

In the first part of this lab, you will build a simple behavioral version of the MIPS ALU. Your ALU should be able to handle the MIPS instructions listed in Table 1.

Table 1: Instructions supported by your ALU. Instructions ignore fields marked N/A.

Instruction	Operation	OP Field	Funct Field	Shamt Field?
ADD	Addition with overflow	0	0x20	No
ADDU	Addition without overflow	0	0x21	No
ADDI	Addition immediate with overflow	0x08	N/A	N/A
ADDIU	Addition immediate with-out overflow	0x09	N/A	N/A
AND	Logical bitwise AND	0	0x24	No

Table 1: Instructions supported by your ALU. Instructions ignore fields marked N/A.

Instruction	Operation	OP Field	Funct Field	Shamt Field?
ANDI	Bitwise AND immediate	0x0c	N/A	N/A
NOR	Logical bitwise NOR	0	0x27	No
OR	Logical bitwise OR	0	0x25	No
ORI	Bitwise OR immediate	0x0d	N/A	N/A
SLL	Shift left logical	0	0x00	Yes
SRA	Shift right arithmetic	0	0x03	Yes
SRL	Shift right logical	0	0x02	Yes
SUB	Subtract with overflow	0	0x22	No
SUBU	Subtract without overflow	0	0x23	No
XOR	Logical bitwise XOR	0	0x26	No
XORI	Bitwise XOR immediate	0x0e	N/A	N/A
LUI	Load upper immediate	0x0f	N/A	N/A

HINT: If you are careful with your instruction decode, you only need to implement one decoding block to handle both R and I-format instructions. Because of this, the actual number of unique operations you have to implement is *much smaller* than the daunting table might suggest.

To support your ALU, we are supplying a simple instruction feeding mechanism which includes a simple 3-ported register file (a behavioral big brother of beloved Lab 1) and a method for reading the above ALU instructions from disk. This will interface with your ALU as shown in Figure 1. You will implement the logic in the grey areas. Your tasks include writing logic to decode the ALU operations and functions, routing and sign extension of immediate values, choosing applicable source and destination registers, and performing the correct ALU operation.

Two exceptions should be reported:

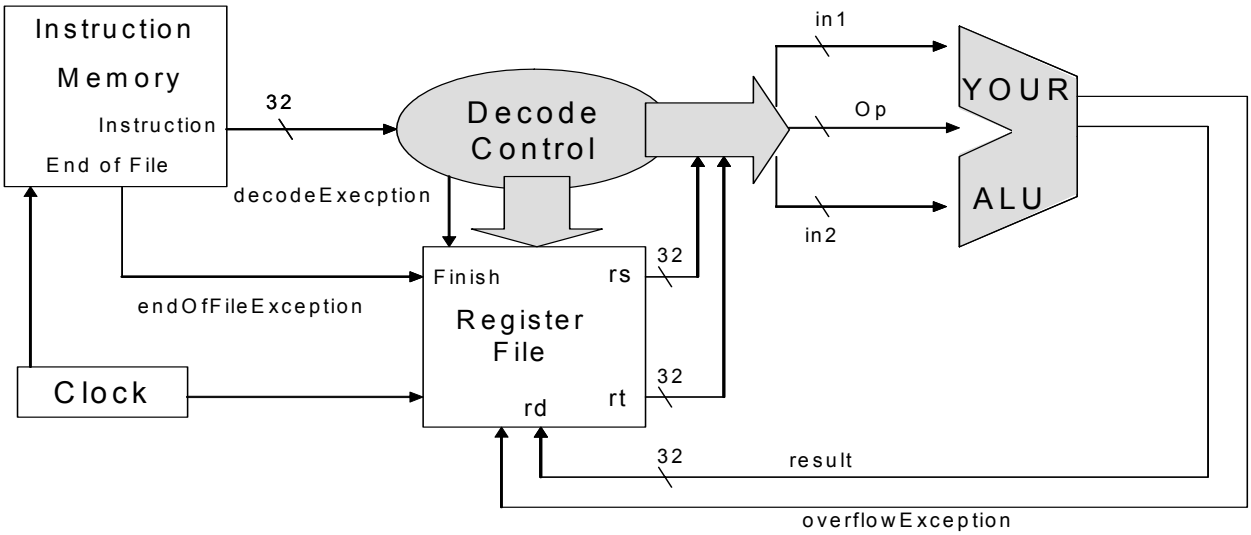
- The ALU should assert the “overflowException” line in the case of an integer overflow (only for instructions which require an overflow check)
- The decoder should raise the “decodeException” line if it sees instructions not listed in Figure 1

Both lines should be fed into the register file, which will halt the simulation and output the register file state. For part 1, use **behavioral Verilog** (e.g., using keywords and operators such as if, assign, +, -, <<, >>, etc.).

The template for our instruction feeder is available from the following file:

`/afs/ece/class/ece347/public_html/LABS/lab2_template.v`

Figure 1: The ALU instruction feeder and register file. You will implement logic in the grey areas



The instruction feeder reads input from a file named 'memory.dat'. The format of this file is the binary representation of each MIPS instruction. We have supplied a number of sample test input files in the lab2_tests/ directory, however your are *strongly advised to write more complete test cases on your own*. Our omniscient grading scripts will test operations not covered in the sample tests.

For hand in, submit your behavioral Verilog files before your lab period.

At the beginning of your lab period, submit a lab report including more diagrams of your ALU, starting with a diagram similar to Figure 1. Please use a computer to draw your diagram.

Part 2: Structural ALU

Now, armed with the behavioral ALU, you should implement a bitsliced ripple carry adder, logical operations, and a barrel shifter using **structural Verilog** (yes, that's gates, none of the behavioral constructs from before). The ripple carry adder should be based off full adders, such as the one in Slide 12 of Lecture 5. The structural barrel shifter can be built out of muxes, and is described in Lecture 6.

Gates in Verilog default to a zero delay. This is not realistic assumption, particularly since it is interesting to see exactly how slow the ripple carry adder can be. For this part and part 3, please use the gate module library in:

`/afs/ece/class/ece347/public_html/LABS/lab2_gates.v`

for your implementation. This file contains declarations for structural Verilog gates of various delays and input counts (Yes, more inputs will yield slower gates, and different logic operations have different delays). The module names are the capitalized version of the gate primitive, appended with the number of inputs, for example a 3-input AND gate is 'AND3'. All gates have parameterizable widths.

For hand in, you should submit your structural Verilog files for part 2 before your lab period. At the beginning of your lab period, you should submit a diagram showing the bitslice unit, how they connect to form a 32-bit MIPS ALU, logical operation circuits, and a block diagram of your barrel shifter as part of your report for this lab.

Part 3: Structural ALU

The ripple carry adder has a simple structure, but it does not deliver high performance. In this part, you will upgrade your ALU with a **structural 32-bit carry-lookahead adder**, similar to the one described in Lecture 5. You will have to build upon the 16-bit adder presented there to generate this adder. Maintain the logical functions (AND, OR, XOR, NOR, shifts) from part 2. No changes need to be made to your barrel shifter.

For hand in, you should submit your structural Verilog files before your lab period. At the beginning of your lab period, you should submit a block diagram of your carry lookahead unit as part of your report for this lab.

Questions

Quantify the size difference between your two adders (ripple carry and carry lookahead). Assume inverters take one area unit, XOR gates take three area units, and all other gates take one area unit per input. How long (in time units) are their worst-case critical paths? How do they compare? Show the critical paths on each adder's diagram.

Suppose in the future, 32 and 64-bit machines are too small for some blindly optimistic DotCom companies. Extrapolate the critical path lengths for a 96-bit machine for both types of adders. How would your barrel shifter's critical path change? Show your work.

Grading

- **Part 1**
  - Verilog ALU and decoder 10 points
  - Block diagram 10 points
- **Part 2**
  - Verilog decode logic 5 points
  - Verilog ripple carry adder and logical operations 15 points
  - Verilog barrel shifter 10 points
  - Block diagram 10 points
- **Part 3**
  - Verilog carry lookahead adder 20 points
  - Block diagram 5 points
- **Questions** **15 points**
- **Total** **100 points**

Late Lab Policy

Late labs and projects will lose 10 points for each day following your assigned lab due date and time. The clock stops when all lab materials have been turned in (including Verilog code, diagrams, answers to questions, etc.) and all demos have been completed.