University of Kansas

Department of Electrical Engineering & Computer Science

Technical Report

MIPS ALU Simulator

Diego Soliz Castro

Dilesh Fernando

Cheng-Yeh Lee

Vuong Nguyen

04th May 2017

Copyright ©2017 Diego Soliz Castro, Dilesh Fernando, Cheng-Yeh Lee, and Vuong Nguyen

Revision History

1.0 (May 2017): Initial Release

First release of this document as a part of EECS 645 class ALU Simulator project.

Abstract

This project describes design and implementation of an Arithmetic Logic Unit (ALU) Simulator to a gain better understanding of how MIPS instructions set works in a MIPS architecture computer. This program simulates subset of MIPS instructions, mostly consisting of R and T type arithmetic instructions. ALU Simulator is implemented by using C programming language in a Linux environment. ALU Simulator simulates instructions read from a data file and prints outs the context of the registers after execution.

ABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

In this project, basic ALU simulator that executes MIPS instructions is implemented using C programming language in a Linux environment. To simulate ALU in MIPS architecture, ALU Simulator program will take instructions in hexadecimal format and executes the instruction by parsing the instruction in to its individual components. As a result, respective register that are associated with instruction is access and updated.

Chapter 2

Overview of MIPS Architecture

MIPS is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Technologies. The early MIPS architectures were 32-bit, with 64-bit versions added later. There are multiple versions of MIPS: including MIPS I, II, III, IV, and V; as well as five releases of MIPS32/64 (for 32- and 64-bit implementations, respectively).

The MIPS architecture is defined by the registers that are available to the assembly language programmer, the instruction set, the memory addressing models, and the data types. The figure below demonstrates different components and features of the MIPS architecture.

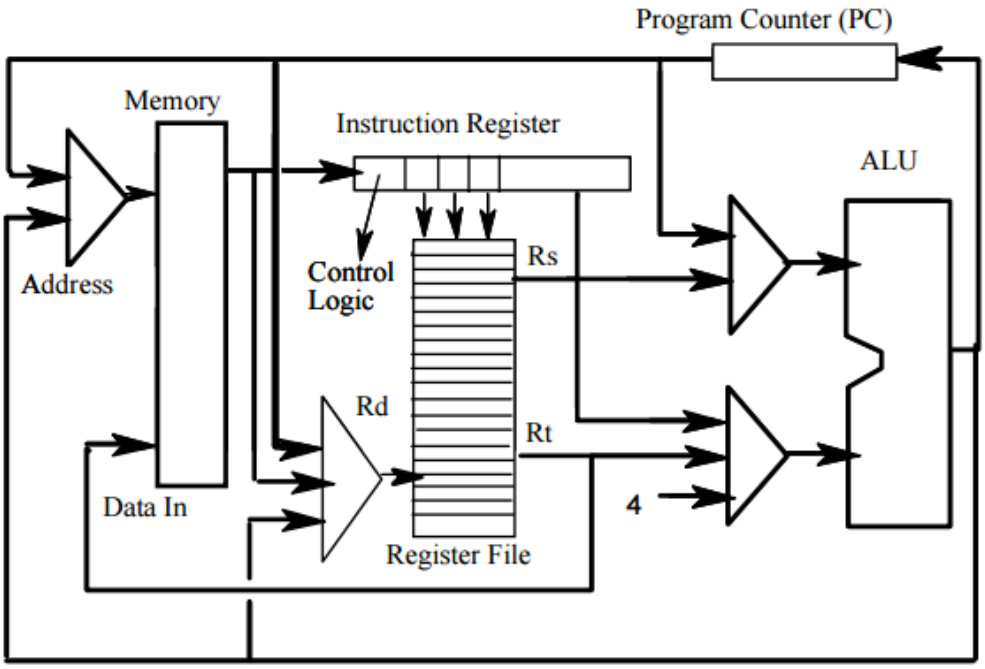
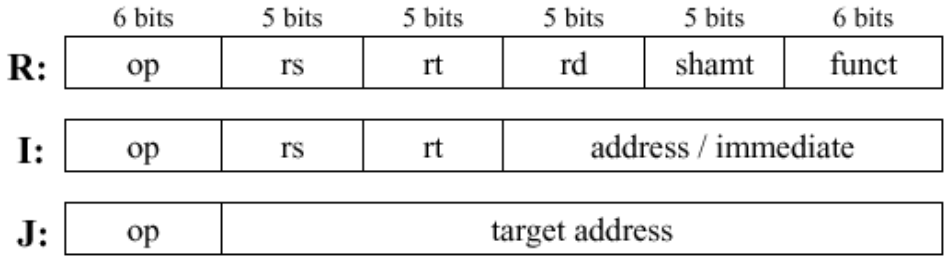


Figure 2.1: Overview of MIPS Architecture (Britton, 2002)

MIPS Instruction Formats

There are three main types of instructions that we will be implementing for this project: R and I instructions.



op: basic operation of the instruction (opcode)
rs: first source operand register
rt: second source operand register
rd: destination operand register
shamt: shift amount
funct: selects the specific variant of the opcode (function code)
address: offset for load/store instructions (+/-2^{15})
immediate: constants for immediate instructions

Figure 2.2: Overview of MIPS Instructions (Mafla, 2011)

R Instructions are used when all the data values used by the instructions are located in registers. All R-type instructions have the following format:

```
OP rd, rs, rt
```

Where “OP” is the mnemonic for the instructions (for example, add). rs and rt are the source registers, and rd is the destination register.

Converting an R mnemonic into binary code is performed with an opcode (6 bits) – the machine representation of the instruction, rs, rt, rd (each has 5 bits) – the numeric representation of the source registers and destination register, shift (5 bits) – only used with shift and rotate instructions, function parameter (6 bits) – contains the control code to differentiate the instructions.

I instructions are used when the instructions must operate on an immediate value (maximum of 16 bits long) and a register value. I instructions are called in the following way:

```
OP rt, rs, IMM
```

Where rt is the target register, rs is the source register, and IMM is the immediate value.

I instructions are converted into machine code with a 6-bit opcode – one-to-one correspondence with a specific mnemonic, rs and rt (each with 5 bits) and the 16-bit immediate value – the offset value in various instructions.

Overview of Register File

The term “register” refers to an electronic storage component. Every register in the MIPS architecture is a component with a capacity to hold a 32-bit binary number. Anyone who has ever used an electronic hand-held calculator has experienced the fact that there is some electronic component inside the calculator that holds the result of the latest computation.

The MIPS architecture has a register file containing 32 registers. Each register has a capacity to hold a 32-bit value. The range of values that can be represented with 32 bits is -2,147,483,648 to +2,147,483,647. When writing at the assembly language level almost every instruction requires that the programmers specify which registers in the register file are used in the execution of the instruction. A convention has been adopted that specifies which registers are appropriate to use in specific circumstances. The registers have been given names that help to remind us about this convention.

- Register $zero is special; it is the source of the constant value zero. Nothing can be stored in register $zero.

- Register number 1 has the name $at, which stands for assembler temporary. This register is reserved to implement “macro instructions” and should not be used by the assembly language programmer.
- Registers $k0 and $k1 are used by the kernel of the operating system and should not be changed by a user program.
- Register $31 is the link register.

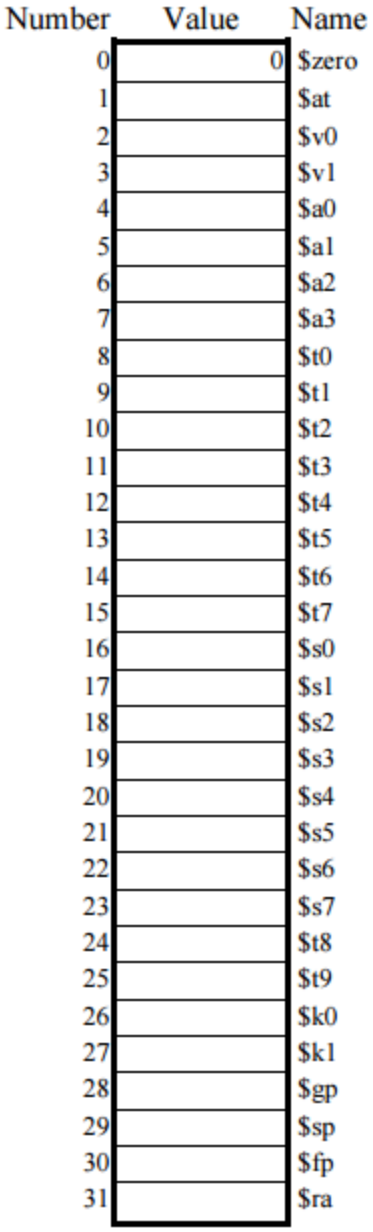


Figure 2.3: MIPS Register File (Britton, 2002)

Overview of Arithmetic Logic Unit (ALU)

The ALU, as its name implies, is a digital logic circuit designed to perform binary arithmetic operations, as well as binary logical operations such as “AND,” “OR,” and “Exclusive OR.” Which operation the ALU performs depends upon the operation code in the Instruction Register.

Some arithmetic and logical instructions operate on one operand from a register and the other from a 16-bit immediate value in the instruction word. The immediate operand is treated as signed for the arithmetic and compare instructions, and treated as logical (zero-extended to register length) for the logical instructions.

The ALU may include the following functions:

- Arithmetic functions (add, subtract, etc.)
- Logical functions (AND, OR, XOR, etc)
- Data transfer functions (Load and Store functions)
- Sequencing functions (Comparison and Shift functions)

Chapter 3

Principles of Operation

In this project, ALU Simulator is designed to simulate an Arithmetic Logic Unit (ALU). By project requirements, ALU Simulator is designed to simulate a subset of MIPS instructions. Mostly R and I type arithmetic instructions.

Following figure illustrates the design approach in building the ALU Simulator.

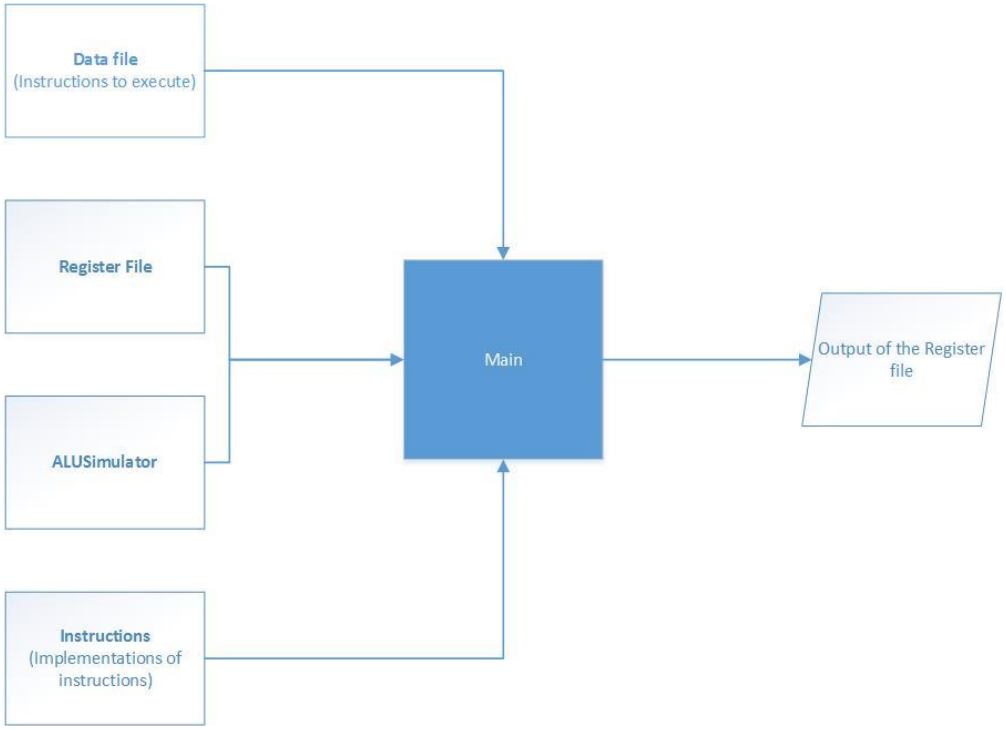


Figure 3.1: Block Diagram of ALU Simulator

To implement the ALU Simulator, main program utilizes three program files and a data file to simulate ALU process in MIPS. Following subsections contains detail descriptions of each of these components.

Main program

Main program consists of ALUSimulator_Main.c, which is the mainline to the ALU Simulator. It Initialize the contents of the Register File, which mimics hardware registers in MIPS CPU. After Initialization of the registers (Register File) and it loads initial starting values to registers.

Next, list of instructions are loaded from a file that contain MIPS instructions as input for the program.

Thereafter for each instruction read, mainline parsers the instruction to its individual components such as source/destination register information, shift amount, and immediate value and calls ALU Simulate program to perform the ALU operations.

After all instructions are simulated by the ALU Simulator, the contexts of the registers are display on to the console.

ALU Simulator

ALUSimulator.h and ALUSimulator.c files contains ALUSimulator program which is called by the main program when an instruction is parsed. The parsed instruction values are passed on to ALUSimulator subroutine as parameters.

The following describes the parameters to ALUSimulator subroutine:

- RegisterFile: Name of the register file used as registers.
- Instruction opcode: Instruction operation code.
- Rs, Rt, and Rd: Scours registers and destination register respectively.
- Shift amount: Bit shit amount for R-Type instructions.
- Function code: Function code of the instruction.
- Immediate value: Immediate value (only in I-Type instructions).
- Status: Status of the instruction operation (not implement).

Register File

Register file consists of RegisterFile_01.h and RegisterFile_01.c files, which implement a virtualized version of actual hardware register file. It contains following subroutines RegisterFile_Cycle, RegisterFile_Read, RegisterFile_Write, and RegisterFile_Dump to perform register file related function. (More detail explanation found in Functions section.)

Instruction File

Instruction file consist of MIPSInstructions.h and MIPSInstructions.c files, which includes implementations of all instruction that are simulated by the ALU Simulator. Following table illustrates the instructions that are implemented in ALU Simulator.

Table 3.1: List of MIPS Instructions Implemented in ALU Simulator

Mnemonic	Instruction Type	OpCode	Function Code
NOOP	R	000000	000000
SLL	R	000000	000000
SRL	R	000000	000010
SRA	R	000000	000011
SLLV	R	000000	000100
SRLV	R	000000	000110
ADD	R	000000	100000
ADDU	R	000000	100001
SUB	R	000000	100010
SUBU	R	000000	100011
AND	R	000000	100100
OR	R	000000	100101
XOR	R	000000	100110
SLT	R	000000	101010
SLTU	R	000000	101011
ADDI	I	001000	xxxxx
ADDIU	I	001001	xxxxx
SLTI	I	001010	xxxxx
SLTIU	I	001011	xxxxx

Data file

ALU Simulator accepts MIPS instruction in 32bit hexadecimal format number. Data File, MIPS_Instructions_01.txt provides ALU Simulator instruction to simulate (This is hard coded into the program). Assembly form of the same instruction can found under file name of MIPS_Instructions_01.asm.

Chapter 4

Data Structures

There are two main data structures used in the MIPS ALU simulator program, a struct and two one dimensional array.

```
typedef struct MIPS_Instruction {
     uint32_t OpCode;
     uint32_t Rs;
     uint32_t Rt;
     uint32_t Rd;
     uint32_t ShiftAmt;
     uint32_t FunctionCode;
     uint32_t ImmediateValue;
} MIPS_Instruction;
```

The struct data structure name MIPS_Instruction is implemented to store a single MIPS instruction. It utilizes seven variables to accommodate operation code number; the source registers indexes, destination register index, arithmetic shift amount value, function code, and the immediate value.

```
typedef int RegisterFile[Registers_Nbr]
```

A simple integer array is used to store the values of each register.

```
MIPS_Instruction MIPS_Instruction_Seq[Instructions_Nbr]
```

A simple MIPS_Instruction array that stores a sequence of MIPS instructions.

Parameters

`MIPS_Immediate_Exp 16`

Determines the amount of bits corresponding to the Immediate value field.

`Instructions_Nbr 5`

Determines the size of our instruction storage array (MIPS_Instruction_Seq).

`Register_Nbr 32`

This parameter determines the length of our register file, for this simulator 32 registers will be initialized since a 32 bit MIPS instruction set is being simulated.

`MIPS_Function_Exp 6`

This signifies the amount of bits dedicated to the function field, with a maximum value of 63.

`MIPS_Shift_Exp 5`

This signifies the amount of bits dedicated to the shift value field, with a maximum value of 31.

`MIPS_Rd_Exp 5`

This signifies the amount of bits dedicated to the destination register value field, with a maximum value of 31.

`MIPS_Rs_Exp 5`

This signifies the amount of bits dedicated to the source register s value field, with a maximum value of 31.

`MIPS_Rt_Exp 5`

This signifies the amount of bits dedicated to the source register t value field, with a maximum value of 31.

`MIPS_OpCode_Exp 6`

This signifies the amount of bits dedicated to the operation code value field, with a maximum value of 63.

Function Descriptions

Following functions description describes all the instructions that ALU Simulator simulates.

```
extern void MIPS_Decode(uint32_t theMIPS_Instruction,
MIPS_Instruction* theMIPSInstruction_Struct)
```

This function decodes MIPS instruction stored on a 32 bit word/uint and stores the extracted data on a MIPS_Instruction structure being referenced by the pointer theMIPSInstruction_Struct.

```
extern void RegisterFile_Read(RegisterFile theRegisterFile,
uint32_t RdAddr_S, uint32_t* RdValue_S, uint32_t RdAddr_T,
uint32_t* RdValue_T)
```

This function reads two values from register indexes RdAddr_S and RdAddr_T and stores the values on the object being referenced by pointers RdAddr_S and RdAddr_T respectively.

```
extern void RegisterFile_Write(RegisterFile theRegisterFile,
bool WrtEnb, uint32_t WrtAddr, uint32_t WrtValue)
```

This function writes a value WrtValue to register index WrtAddr, given that the Boolean flag WrtEnb is true.

```
extern void RegisterFile_Dump(RegisterFile theRegisterFile)
```

This function dumps/prints the values in the RegisterFile array.

```
extern void RegisterFile_Cycle(RegisterFile theRegisterFile,
uint32_t RdAddr_S, uint32_t* RdValue_S, uint32_t RdAddr_T,
uint32_t* RdValue_T, bool WrtEnb, uint32_t WrtAddr, uint32_t
WrtValue)
```

This function reads two values from register indexes RdAddr_S and RdAddr_T and stores the values on the object being referenced by pointers RdAddr_S and RdAddr_T respectively additionally if the the boolean flag WrtEnb is true it also writes a value WrtValue to register index WrtAddr, given that, this function more closely resembles the behavior of a register access cycle.

```
extern void ALUSimulator(RegisterFile theRegisterFile, uint32_t
OpCode, uint32_t Rs, uint32_t Rt, uint32_t Rd, uint32_t
ShiftAmt, uint32_t FunctionCode, uint32_t ImmediateValue,
uint32_t* Status)
```

This function is called when an instruction needs to be executed, given an specific FunctionCode value an specific function is called to execute a MIPS instruction.

```
extern void NOOP()
```

This function does no operation.

```
extern void SLL(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t Rd, uint32_t ShiftAmt)
```

This function makes a logical left shift on a value stored in register index Rs by the value assigned to the ShiftAmt variable, the result is then stored at register index Rd.

```
extern void SRL(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t Rd, uint32_t ShiftAmt)
```

This function makes a logical right shift on a value stored in register index Rs by the value assigned to the ShiftAmt variable, the result is then stored at register index Rd.

```
extern void SRA(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t Rd, uint32_t ShiftAmt)
```

This function makes an arithmetic right shift on a value stored in register index Rs by the value assigned to the ShiftAmt variable, the result is then stored at register index Rd.

```
extern void SLLV(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t Rd)
```

This function makes a logical left shift on a value stored in register index Rs by a value stored in register index Rt, the result is then stored at register index Rd.

```
extern void SRLV(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t Rd)
```

This function makes a logical right shift on a value stored in register index Rs by a value stored in register index Rt, the result is then stored at register index Rd.

```
extern void ADD(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t Rd)
```

This function makes an addition operation to two values stored in register index Rs and other stored in register index Rt, the result is then stored at register index Rd.

```
extern void ADDU(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t Rd)
```

This function makes an unsigned addition operation to two values stored in register index Rs and other stored in register index Rt, the result is then stored at register index Rd.

```
extern void SUB(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t Rd)
```

This function subtracts a value stored in register index Rt from another stored in register index Rs, the result is then stored at register index Rd.

```
extern void SUBU(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t Rd)
```

This function subtracts an unsigned value stored in register index Rt from another stored in register index Rs, the result is then stored at register index Rd.

```
extern void AND(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t Rd)
```

This function does a bitwise AND comparison between two values stored on indexes Rs and Rt, the resulting value is then stored at register index Rd.

```
extern void OR(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t Rd)
```

This function does a bitwise OR comparison between two values stored on indexes Rs and Rt, the resulting value is then stored at register index Rd.

```
extern void XOR(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t Rd)
```

This function does a bitwise XOR comparison between two values stored on indexes Rs and Rt, the resulting value is then stored at register index Rd.

```
extern void SLT(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t Rd)
```

This function does a less than comparison operation between two values stored on indexes Rs and Rt, the resulting value is then stored at register index Rd.

```
extern void SLTU(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t Rd)
```

This function does a less than comparison operation between two unsigned values stored on indexes Rs and Rt, the resulting value is then stored at register index Rd.

```
extern void ADDI(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t ImmediateValue)
```

This function makes an addition operation to two values stored in register index Rs and other stored in the ImmediateValue variable, the result is then stored at register index Rd.

```
extern void ADDIU(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t ImmediateValue)
```

This function makes an addition operation to two unsigned values stored in register index Rs and other stored in the ImmediateValue variable, the result is then stored at register index Rd.

```
extern void OR(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t Rd)
```

This function does a bitwise OR comparison between two values stored on indexes Rs and Rt, the resulting value is then stored at register index Rd.

```
extern void SLTI(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t ImmediateValue)
```

This function does a less than comparison operation between two values stored on indexes Rs and ImmediateValue, the resulting value is then stored at register index Rd.

```
extern void SLTIU(RegisterFile theRegisterFile, uint32_t Rs,
uint32_t Rt, uint32_t ImmediateValue)
```

This function does a less than comparison operation between two unsigned values stored on indexes Rs and ImmediateValue, the resulting value is then stored at register index Rd.

Chapter 5

The following describes the file structure, compilation, and running of ALU Simulator Program.

Files

This project contains several program files, a MIPS instruction input data file, and one makefile. Following table lists the file names and a short description of their contents.

Following files were provided as project resources:

- ALUSimulator_Main.c,
- ALUSimulator.h, ALUSimulator.c
- RegisterFile_01.h, RegisterFile_01.c
- MIPS_Instructions_01.asm, MIPS_Instructions_01.txt
-

Table 5.1: List of Project Files and Descriptions

File Name	Discription
ALUSimulator_Main.c	Main ALU simulator program
ALUSimulator.h	Arithmetic Logic Unit (ALU) simulator program header file
ALUSimulator.c	Implementation of ALU program
RegisterFile_01.h	Register simulator program
RegisterFile_01.c	Implementation of register program
MIPSInstructions.h	MIPS instructions header file
MIPSInstructions.c	Implementation of MIPS instructions program
Makefile	Builds the ALU simulator
MIPS_Instructions_01.asm	MIPS assembly instructions
MIPS_Instructions_01.txt	MIPS assembly instructions in hexadecimal format

Compile and Running

ALU Simulator is built for Linux environment and complied with GNU GCC complier. Therefore future compilations and execution of the program should be performed in a Linux environment with GNU GCC complier. Follow the instruction below illustrate how to compile and run the Cache Simulator at command prompt.

Compiling:

To compile ALU Simulator run provided Makefile. Makefile contain commands to compile all parts of the ALU Simulator in one executable.

```
$ make
```

Running:

To run ALU Simulator, enter the following command in given format.

```
$ ./ALUSimulator
```

This will execute ALU Simulator with MIPS_Instructions_01.txt as the input source of MIPS instructions.

Note: MIPS_Instructions_01.txt must contain MIPS instructions in hexadecimal format.

Chapter 6

Testing

Testing was performed by generating test file that calls the instruction and determine if it functions perform correctly. During the testing process, several tools were utilized for validity of the testing process.

- The MIPS converter was used to covert instruction to Hexadecimal so it can be read by the program.
- Decimal to Hexadecimal converter is used to create the immediate value part of the instruction and check if the computation is correct.
- Bitwise calculator is used to check with the result of SLL, SRL, SRA, SLLV, SRLV, AND, OR and XOR instructions.

The following details the challenges faced and actions taken during testing individual instructions.

SLL instruction is tested by shifting a small value and a large value registers 1 and 31 bits. A bug was discovered during this process, which the opcode and the function code of SLL are identical to NOOP; therefore we need to differentiate two instructions. The bug was fixed by checking the other parts of the instruction if they are also 0.

The same testing procedure is done for SRL instruction.

SRA instruction was tested by shifting a positive value and a negative value registers. It will shift in 1 if the value is negative and 0 if the value is positive.

The testing procedures of SLLV and SRLV instructions are similar to SLL since we don't have to check for overflow.

ADD and ADDU instructions are tested by doing addition of two positive value registers, one positive and one negative register, and two negative value registers.

SUB and SUBU instructions are tested by doing subtraction of two positive value registers, one positive and one negative register, and two negative value registers.

SLT and SLTU instructions are tested by comparing two positive value registers, and one positive and one negative value register. If the first register given is smaller than the second register than it will return 1, else return 0.

ADDI and ADDIU instructions are tested by doing addition of one positive value register and one positive immediate value, one positive value register and one negative immediate value, one negative value register and one positive immediate value, and one negative value register and one negative immediate value.

SLTI and SLTIU instructions are tested by comparing one positive value register and one positive immediate value, one positive value register and one negative immediate value, one negative value register and one positive immediate value, and one negative value register and one negative immediate value.

By testing all the instructions, we are able to determine the output generated by the ALU simulator is correct.

Results

MIPS_Instructions_01.txt is the test file provided as part of the project. The initial register file and the final register file after running MIPS_Instructions_01.txt is shown below.

```
Initial RegisterFile: ======================================

00000000:  00000000  00000005  00000010  00000017

00000004:  00000390  00001010  00000000  00000000

00000008:  00000000  00000000  00000000  00000000

0000000C:  00000000  00000000  00000000  00000000

00000010:  00000000  00000000  00000000  00000000

00000014:  00000000  00000000  00000000  00000000

00000018:  00000000  00000000  00000000  00000000

0000001C:  00000000  00000000  00000000  00000000
```

```
Final RegisterFile: ======================================

00000000:  00000000  00000005  00000010  00000017

00000004:  00000390  00001010  00000000  00000000

00000008:  00000000  00000074  00000F90  00000395

0000000C:  00010100  00001390  00000007  00000007

00000010:  00000000  00000000  00000000  00000000

00000014:  00000000  00000000  00000000  00000000

00000018:  00000000  00000000  00000000  00000000

0000001C:  00000000  00000000  00000000  00000000
```

The following table displays the instructions that were executed to obtain the result above. By examining the execution of each instruction and comparison of the above output the provided output, it was determined that the ALU Simulator is function as intended.

Table 6.1: List of Instructions Tested.

Instruction	Format	Result (Hex)
20490064	addi $09,S02,100	$09 ← 16 + 100 = 116 (74)
20aaff80	addi $10,$05,-128	$10 ← 4112 + (-128) = 3984 (F90)
00245820	add $11,$01,$04	$11 ← 5 + 912 = 917 (395)
00056100	sll $12,$05,4	$12 ← 4112 << 4 = 65792 (10100)
00856825	or $13,$04,$05	$13 ← 912 | 4112 = 5008 (1390)
00437026	xor $14,$02,$03	$14 ← 16 ⊕ 23 = 7 (7)
00627822	sub $15,$03,$02	$15 ← 23 − 16 = 7 (7)

Chapter 7

Member Contribution

Diego Soliz Castro:

- Function prototyping and coding.
- Report writing: data struct, function, and parameter descriptions.

Dilesh Fernando:

- Program design and coding.
- Creating figures, tables, and report writing.

Cheng-Yeh Lee:

- Testing
- Report writing

Vuong Nguyen:

- Writing and editing the final report.

Conclusion

When we write the ALUSimulator in C, we are implementing a list of instructions for the arithmetic operations, logic operations, comparison and shifting functions. These instructions direct the processor to perform to accomplish some task (an algorithm). The ALUSimulator takes in: the RegisterFile, the instruction opcode and the register indices, the shift amount, the function code, the immediate value and the status. The provided structure includes a main program, a simulation of a register file and the list of instructions to be implemented.

With such setup for this project, we are able to understand the basic overview of MIPS architecture and how each component operates in the system. Once we have acquired a general understanding of the functional model of the processor and MIPS instructions, we can implement each given instructions in the ALUSimulator.

Things we have learned

- The overview of MIPS architecture and the operation of the processor.
- The basic functional components of the MIPS processor.
- The difference between a register file and the ALU and how they function.
- Details about the ALU and its different operational instructions.

- The binary number system, the rules of performing arithmetic and detecting overflow. Although, overflow-detection was too complicated in implement in this project, we were able to ask questions and research about best industry practices to handle overflow.
- The utility of logical operators and shift operations.
- Implementing ALU instructions in C programming language.
- Testing the arithmetic and logic operations in 32-bit binary and hexadecimal representation.

Things we would do differently next time

- Use test driven development approach to writing instruction functions. For example, write a test first and write the instruction function to pass the test.

References

Britton, Robert. *MIPS Assembly Language Programming*. Chico, California: Robert Britton, 2002. http://www.ittc.ku.edu/~gminden/Computer_Architecture/PDFs/britton-mips-tutorial.pdf Accessed May 2017.

Mafla, E. (2011, April 11). MIPS instruction format [Course Notes, CDA3101]. Retrieved from http://www.cise.ufl.edu/~emafla/

Minden, Gary J. Initial project structure was downloaded from following URL */~gminden/Computer_Architecture/Software*

www.ittc.ku.edu/~gminden/Computer_Architecture/Software/. Accessed May 1, 2017.

MIPS Assembly/Instruction Formats. Wikibooks, open books for an open world. N.p., 3 Oct. 2016. https://en.wikibooks.org/wiki/MIPS_Assembly/Instruction_Formats . Accessed May 2017.

Price, Charles. *MIPS IV Instruction Set*. Mountain View, CA: MIPS Technologies, Inc. , 1995. Revision 3.2. http://www.ittc.ku.edu/~gminden/Computer_Architecture/PDFs/mips-isa.pdf. Accessed May 2017

Appendix

ALUSimulator_Main.c

```
//***************************************************************************
//--ALUSimulator_Main
//
//              Author:         Gary J. Minden
//              Organization:   KU/EECS/EECS 645
//              Date:                   2017-04-22 (B70422)
//              Version:        1.0
//              Description:    This is the main program to manage and control
//                                              a simple ALU simulator
//              Notes:
//
//***************************************************************************
//

#include <stddef.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdarg.h>

#include <stdio.h>

#include "RegisterFile_01.h"
#include "ALUSimulator.h"
#include "MIPSInstructions.h"

//
//      Define a structure to hold a decoded MIPS instruction.
//      Not all fields are valid for each instruction.
//      For example, the instruction is a R-type, the immediate
//              field is not valid.
//      The OpCode field determines the instruction type.
//
typedef struct MIPS_Instruction {
        uint32_t OpCode;
        uint32_t Rs;
        uint32_t Rt;
        uint32_t Rd;
        uint32_t ShiftAmt;
        uint32_t FunctionCode;
        uint32_t ImmediateValue;
} MIPS_Instruction;

#define         Instructions_Nbr 5

MIPS_Instruction MIPS_Instruction_Seq[Instructions_Nbr];

RegisterFile Primary_RegisterFile;

//
//***************************************************************************
//
//      Defines for decoding a MIPS instruction. Only R-format and I-format
//      are handled for now.
//
#define         MIPS_Immediate_Exp              16
#define         MIPS_Immediate_Nbr              ( 1 << MIPS_Immediate_Exp )
#define         MIPS_Immediate_Mask             ( MIPS_Immediate_Nbr - 1 )
#define         MIPS_Immediate_Offset   0

#define         MIPS_Function_Exp               6
#define         MIPS_Function_Nbr               ( 1 << MIPS_Function_Exp )
#define         MIPS_Function_Mask              ( MIPS_Function_Nbr - 1 )
```

```
#define         MIPS_Function_Offset    0

#define         MIPS_Shift_Exp                  5
#define         MIPS_Shift_Nbr                  ( 1 << MIPS_Shift_Exp )
#define         MIPS_Shift_Mask                 ( MIPS_Shift_Nbr - 1)
#define         MIPS_Shift_Offset               ( MIPS_Function_Exp )

#define         MIPS_Rd_Exp                             5
#define         MIPS_Rd_Nbr                             ( 1 << MIPS_Rd_Exp )
#define         MIPS_Rd_Mask                    ( MIPS_Rd_Nbr - 1)
#define         MIPS_Rd_Offset                  ( MIPS_Shift_Offset + MIPS_Shift_Exp )

#define         MIPS_Rt_Exp                             5
#define         MIPS_Rt_Nbr                             ( 1 << MIPS_Rt_Exp )
#define         MIPS_Rt_Mask                    ( MIPS_Rt_Nbr - 1)
#define         MIPS_Rt_Offset                  ( MIPS_Rd_Offset + MIPS_Rd_Exp )

#define         MIPS_Rs_Exp                             5
#define         MIPS_Rs_Nbr                             ( 1 << MIPS_Rs_Exp )
#define         MIPS_Rs_Mask                    ( MIPS_Rs_Nbr - 1)
#define         MIPS_Rs_Offset                  ( MIPS_Rt_Offset + MIPS_Rt_Exp )

#define         MIPS_OpCode_Exp                 6
#define         MIPS_OpCode_Nbr                 ( 1 << MIPS_OpCode_Exp )
#define         MIPS_OpCode_Mask        ( MIPS_OpCode_Nbr - 1)
#define         MIPS_OpCode_Offset              ( MIPS_Rs_Offset + MIPS_Rs_Exp )

extern void MIPS_Instruction_Dump(MIPS_Instruction theMIPSInstruction) {
        printf(">>Opcode: %02X; Rs: %02X; Rt: %02X; Rd: %02X;\n",
                        theMIPSInstruction.OpCode, theMIPSInstruction.Rs,
                        theMIPSInstruction.Rt, theMIPSInstruction.Rd);

        printf(">>>>ShiftAmt: %02X; FunctionCode: %02X; ImmediateValue: %04X;\n",
                        theMIPSInstruction.ShiftAmt, theMIPSInstruction.FunctionCode,
                        theMIPSInstruction.ImmediateValue);

}

void MIPS_Offset_Report() {
        printf(">>>>Immediate Offset: %d\n", MIPS_Function_Offset);
        printf(">>>>Function Offset: %d\n", MIPS_Function_Offset);
        printf(">>>>Shift Offset: %d\n", MIPS_Shift_Offset);
        printf(">>>>Rd Offset: %d\n", MIPS_Rd_Offset);
        printf(">>>>Rt Offset: %d\n", MIPS_Rt_Offset);
        printf(">>>>Rs Offset: %d\n", MIPS_Rs_Offset);
        printf(">>>>OpCode Offset: %d\n", MIPS_OpCode_Offset);
}

//
//      The variable "theMIPS_Instruction" is a 32-bit MIPS instruction.
//      The variable "theMIPSInstruction_Struct" is a structure
//              representing a MIPS instruction for simulation.
//

extern void MIPS_Decode(uint32_t theMIPS_Instruction,
                MIPS_Instruction* theMIPSInstruction_Struct) {

        //
        //      This subroutine does not distinguish between
        //              a R-format or an I-format instruction.
        //
        //      First extract the Immediate portion of the instruction.
        //
        theMIPSInstruction_Struct->ImmediateValue = (theMIPS_Instruction
                        & MIPS_Immediate_Mask);

        //
        //      Extract the remaining R-format fields
        //
```

```
        theMIPSInstruction_Struct->FunctionCode = (theMIPS_Instruction
                        & MIPS_Function_Mask);

        theMIPSInstruction_Struct->ShiftAmt = ((theMIPS_Instruction
                        >> MIPS_Shift_Offset) &
        MIPS_Shift_Mask);

        theMIPSInstruction_Struct->Rd = ((theMIPS_Instruction >> MIPS_Rd_Offset) &
        MIPS_Rd_Mask);

        theMIPSInstruction_Struct->Rt = ((theMIPS_Instruction >> MIPS_Rt_Offset) &
        MIPS_Rt_Mask);

        theMIPSInstruction_Struct->Rs = ((theMIPS_Instruction >> MIPS_Rs_Offset) &
        MIPS_Rs_Mask);

        theMIPSInstruction_Struct->OpCode = ((theMIPS_Instruction
                        >> MIPS_OpCode_Offset) &
        MIPS_OpCode_Mask);

//      MIPS_Instruction_Dump( *theMIPSInstruction_Struct );

}

//
//*****************************************************************************
//
int32_t main() {

        uint32_t Files_Nbr = 3;
        uint32_t Files_Idx;
        char* Filenames[] = { "MIPS_Instructions_01.txt" };
        FILE* MIPS_Iinstruction_File;
        uint32_t FReadStatus;

        char Test_String[128];
        uint32_t aMIPS_Instruction;

        uint32_t ALUStatus = 0;

//      MIPS_Offset_Report();

        //
        //      Load Primary_RegisterFile
        //
        RegisterFile_Write(Primary_RegisterFile, true, 0X01, 0X05);
        RegisterFile_Write(Primary_RegisterFile, true, 0X02, 0X10);
        RegisterFile_Write(Primary_RegisterFile, true, 0X03, 0X17);
        RegisterFile_Write(Primary_RegisterFile, true, 0X04, 0X390);
        RegisterFile_Write(Primary_RegisterFile, true, 0X05, 0X1010);

        printf("Initial RegisterFile: ======================================\n");
        RegisterFile_Dump(Primary_RegisterFile);

        //
        //      Open MIPS instruction file
        //
        Files_Idx = 0;
        MIPS_Iinstruction_File = fopen(Filenames[Files_Idx], "r");
        if (MIPS_Iinstruction_File == NULL) {
                printf(">>>>File open error.\n");
                return (0);
        }

//      printf( ">>>>File opened.\n" );

        while (1) {

                FReadStatus = fscanf(MIPS_Iinstruction_File, "%x", &aMIPS_Instruction);
```

```
                //
                //      Check for end of file
                //
                if (FReadStatus == EOF) {
                        break;
                }

                printf("Instruction: %08X\n", aMIPS_Instruction);
                MIPS_Decode(aMIPS_Instruction, &MIPS_Instruction_Seq[0]);
                MIPS_Instruction_Dump(MIPS_Instruction_Seq[0]);

                ALUSimulator(Primary_RegisterFile, MIPS_Instruction_Seq[0].OpCode,
                                MIPS_Instruction_Seq[0].Rs, MIPS_Instruction_Seq[0].Rt,
                                MIPS_Instruction_Seq[0].Rd, MIPS_Instruction_Seq[0].ShiftAmt,
                                MIPS_Instruction_Seq[0].FunctionCode,
                                MIPS_Instruction_Seq[0].ImmediateValue, &ALUStatus);

        }

        fclose(MIPS_Iinstruction_File);

        printf("Final RegisterFile: ======================================\n");

        RegisterFile_Dump(Primary_RegisterFile);
}
```

ALUSimulator.h

```
//************************************************************************
//--ALUSimulator.h
//
//              Author:                 Gary J. Minden
//              Organization:   KU/EECS/EECS 645
//              Date:                   2017-04-22 (B70422)
//              Version:                1.0
//              Description:    This is the prototype for a simple ALU simulator
//              Notes:
//
//************************************************************************
//

extern void ALUSimulator( RegisterFile theRegisterFile,
                                uint32_t OpCode,
                                uint32_t Rs, uint32_t Rt, uint32_t Rd,
                                uint32_t ShiftAmt,
                                uint32_t FunctionCode,
                                uint32_t ImmediateValue,
                                uint32_t* Status );
```

ALUSimulator.c

```
//************************************************************************
//--ALUSimulator.c
//
//              Author:                 Gary J. Minden
//              Organization:   KU/EECS/EECS 645
//              Date:                   2017-04-22 (B70422)
//              Version:                1.0
//              Description:    This is the test standin for a simple ALU simulator
```

```
//             Notes:
//
//*************************************************************************
//

#include <stddef.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdarg.h>

#include <stdio.h>

#include "RegisterFile_01.h"
#include "ALUSimulator.h"
#include "MIPSInstructions.h"

extern void ALUSimulator(RegisterFile theRegisterFile, uint32_t OpCode,
               uint32_t Rs, uint32_t Rt, uint32_t Rd, uint32_t ShiftAmt,
               uint32_t FunctionCode, uint32_t ImmediateValue, uint32_t* Status) {
        printf(">>ALU: Opcode: %02X; Rs: %02X; Rt: %02X; Rd: %02X;\n", OpCode, Rs,
                       Rt, Rd);

        printf(
                       ">>>>ALU: ShiftAmt: %02X; FunctionCode: %02X; ImmediateValue: %04X;\n",
                       ShiftAmt, FunctionCode, ImmediateValue);

        // Determine type instruction
        // If OpCode is 0, its R type instruction else I type instruction
        if (OpCode == 0) {
               //R-Type instruction
               switch (FunctionCode) {
               case 0:
                       if ((Rs == 0) && (Rt == 0) && (Rd == 0) && (ShiftAmt == 0)
                                       && (ImmediateValue == 0)) {
                               // Bit pattern for NOOP
                               NOOP();
                       } else {
                               SLL(theRegisterFile, Rs, Rt, Rd, ShiftAmt);
                       }
                       break;
               case 2:
                       SRL(theRegisterFile, Rs, Rt, Rd, ShiftAmt);
                       break;
               case 3:
                       SRA(theRegisterFile, Rs, Rt, Rd, ShiftAmt);
                       break;
               case 4:
                       SLLV(theRegisterFile, Rs, Rt, Rd);
                       break;
               case 6:
                       SRLV(theRegisterFile, Rs, Rt, Rd);
                       break;
               case 32:
                       ADD(theRegisterFile, Rs, Rt, Rd);
                       break;
               case 33:
                       ADDU(theRegisterFile, Rs, Rt, Rd);
                       break;
               case 34:
                       SUB(theRegisterFile, Rs, Rt, Rd);
                       break;
               case 35:
```

```
                        SUBU(theRegisterFile, Rs, Rt, Rd);
                        break;
                case 36:
                        AND(theRegisterFile, Rs, Rt, Rd);
                        break;
                case 37:
                        OR(theRegisterFile, Rs, Rt, Rd);
                        break;
                case 38:
                        XOR(theRegisterFile, Rs, Rt, Rd);
                        break;
                case 42:
                        SLT(theRegisterFile, Rs, Rt, Rd);
                        break;
                case 43:
                        SLTU(theRegisterFile, Rs, Rt, Rd);
                        break;
                default:
                        //Wrong instruction.
                        printf(
                                        ANSI_COLOR_RED_BACK "Error: Wrong instruction."
ANSI_COLOR_RESET "\n");
                        //return (1);
                }
        } else {
                //I-Type instruction
                switch (OpCode) {
                case 8:
                        ADDI(theRegisterFile, Rs, Rt, ImmediateValue);
                        break;
                case 9:
                        ADDIU(theRegisterFile, Rs, Rt, ImmediateValue);
                        break;
                case 10:
                        SLTI(theRegisterFile, Rs, Rt, ImmediateValue);
                        break;
                case 11:
                        SLTIU(theRegisterFile, Rs, Rt, ImmediateValue);
                        break;
                default:
                        //Wrong instruction.
                        printf(
                                        ANSI_COLOR_RED_BACK "Error: Wrong instruction."
ANSI_COLOR_RESET "\n");
                        //return (1);
                }
        }

} //End of ALUSimulator function
```

RegisterFile_01.h

```
//  Created by Gary J. Minden on 10/20/2015.
//  Copyright 2015 Gary J. Minden. All rights reserved.
//
//  Updates:
//
//      B51020 -- initial version
//
//
```

```
//     Description:
//                     This program simulates a three port register file.
//

#ifndef __RegisterFile_H_
#define __RegisterFile_H_

#include <stddef.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdarg.h>

//
//     Define RegisterFile structures
//
#define Registers_Nbr 32

typedef int RegisterFile[Registers_Nbr];

//****************************************************************************
//
// Prototypes for the APIs.
//
//****************************************************************************

extern void RegisterFile_Cycle( RegisterFile theRegisterFile,
                                                                uint32_t RdAddr_S, uint32_t*
RdValue_S,
                                                                uint32_t RdAddr_T, uint32_t*
RdValue_T,
                                                                bool WrtEnb, uint32_t WrtAddr,
uint32_t WrtValue );

extern void RegisterFile_Read( RegisterFile theRegisterFile,
                                                                uint32_t RdAddr_S, uint32_t*
RdValue_S,
                                                                uint32_t RdAddr_T, uint32_t*
RdValue_T );

extern void RegisterFile_Write( RegisterFile theRegisterFile,
                                                                bool WrtEnb, uint32_t WrtAddr,
uint32_t WrtValue );

extern void RegisterFile_Dump( RegisterFile theRegisterFile );

#endif         // __RegisterFile_H_
```

RegisterFile_01.c

```
//  Created by Gary J. Minden on 10/20/2015.
//  Copyright 2015 Gary J. Minden. All rights reserved.
//
//  Updates:
//
//      B51020 -- initial version
//
//
//     Description:
//                     This program simulates a CPU RegisterFile.
//
```

```
#include <stdio.h>
#include <time.h>
#include <strings.h>
#include <stdlib.h>
#include <unistd.h>

#include <stddef.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdarg.h>

#include "RegisterFile_01.h"

//
//      RegisterFile_Cycle performs one cycle of the register file. This involves
//              two read operations and one write operation. The write operation is
//              gated by a WrtEnb signal. The write operation is performed after
//              the read operations which better reflects the behavior of a
//              hardware register file.
//
extern void RegisterFile_Cycle(RegisterFile theRegisterFile, uint32_t RdAddr_S,
                uint32_t* RdValue_S, uint32_t RdAddr_T, uint32_t* RdValue_T,
                bool WrtEnb, uint32_t WrtAddr, uint32_t WrtValue) {

        printf(
                        "RegisterFile_Cycle: RdAddr_S: %02d; RdAddr_T: %02d; WrtEnb: %01d;
WrtAddr: %02d\n",
                        RdAddr_S, RdAddr_T, WrtEnb, WrtAddr);

        //
        //      Read the operands
        //
        *RdValue_S = theRegisterFile[RdAddr_S];
        *RdValue_T = theRegisterFile[RdAddr_T];

        //
        //      If enabled, write data to the register file.
        //
        if (WrtEnb == true) {
                theRegisterFile[WrtAddr] = WrtValue;
        }

}

//
//      In many cases it is only necessary to read value from a register file.
//              RegisterFile_Read just does the value read operation.
//
extern void RegisterFile_Read(RegisterFile theRegisterFile, uint32_t RdAddr_S,
                uint32_t* RdValue_S, uint32_t RdAddr_T, uint32_t* RdValue_T) {

        printf("RegisterFile_Read: RdAddr_S: %02d; RdAddr_T: %02d;\n", RdAddr_S,
                        RdAddr_T);

        //
        //      Read the operands
        //
        *RdValue_S = theRegisterFile[RdAddr_S];
        *RdValue_T = theRegisterFile[RdAddr_T];

}
```

```
//
//      In many cases it is only necessary to write a value to the register file.
//              RegisterFile_Write just does the write operation. The WrtEnb argument
//              is included for consistency.
//
extern void RegisterFile_Write(RegisterFile theRegisterFile, bool WrtEnb,
                uint32_t WrtAddr, uint32_t WrtValue) {

        printf("RegisterFile_Write:  WrtEnb: %01d; WrtAddr: %02d; WrtValue: %08X\n", WrtEnb,
                        WrtAddr,WrtValue);

        //
        //      If enabled, write data to the register file.
        //
        if (WrtEnb == true) {
                theRegisterFile[WrtAddr] = WrtValue;
        }

}

//
//      RegisterFile_Dump prints the contents of a register file to standard output
//              (stdout). RegisterFile_Dump is useful for debugging purposes. In the future,
//              this subroutine should include a file descriptor indicating the output.
//
extern void RegisterFile_Dump(RegisterFile theRegisterFile) {

        //
        //      This routine prints four values per line in hexadecimal format.
        //              The number of lines is determined by the number of registers
        //              in the register file. We assume the number of registers is
        //              a multiple of 4. If not, some registers will not be printed.
        //
        uint32_t Lines_Nbr = ( Registers_Nbr / 4);
        uint32_t Line_Idx;

        for (Line_Idx = 0; Line_Idx < Lines_Nbr; Line_Idx++) {
                printf("%08X:  %08X  %08X  %08X  %08X\n", (Line_Idx * 4),
                                theRegisterFile[(Line_Idx * 4) + 0],
                                theRegisterFile[(Line_Idx * 4) + 1],
                                theRegisterFile[(Line_Idx * 4) + 2],
                                theRegisterFile[(Line_Idx * 4) + 3]);
        }
}
```

MIPSInstructions.h

```
//*****************************************************************************
//
//      Program:                MIPSInstructions.h
//      Author:                 Dilesh Fernando <fernando.dilesh@gmail.com>
//                                      Cheng-Yeh Lee <chengyeh90@gmail.com>
//                                      Vuong Nguyen <nptvuong2912@gmail.com>
//                                      Diego Soliz Castro <zilosgodi@gmail.com>
//      Date:                   2017-04-21
//      Description:    A program to simulate MIPS Instructions.
//
//*****************************************************************************
```

```
#include <stdint.h>
#include "RegisterFile_01.h"

#define ANSI_COLOR_RED_BACK "\x1b[41m"
#define ANSI_COLOR_RED      "\x1b[31m"
#define ANSI_COLOR_GREEN    "\x1b[32m"
#define ANSI_COLOR_YELLOW   "\x1b[33m"
#define ANSI_COLOR_BLUE     "\x1b[34m"
#define ANSI_COLOR_MAGENTA  "\x1b[35m"
#define ANSI_COLOR_CYAN     "\x1b[36m"
#define ANSI_COLOR_RESET    "\x1b[0m"

#ifndef __MIPSInstructions_H_
#define __MIPSInstructions_H_

extern void NOOP();

extern void SLL(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd, uint32_t ShiftAmt);

extern void SRL(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd, uint32_t ShiftAmt);

extern void SRA(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd, uint32_t ShiftAmt);

extern void SLLV(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd);

extern void SRLV(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd);

extern void ADD(RegisterFile, uint32_t, uint32_t, uint32_t);

extern void ADDU(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd);

extern void SUB(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd);

extern void SUBU(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd);

extern void AND(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd);

extern void OR(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd);

extern void XOR(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd);

extern void SLT(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd);

extern void SLTU(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd);

extern void ADDI(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t ImmediateValue);

extern void ADDIU(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
```

```
               uint32_t ImmediateValue);

extern void SLTI(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t ImmediateValue);

extern void SLTIU(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t ImmediateValue);

#endif  // __MIPSInstructions_H_
```

MIPSInstructions.c

```
//*****************************************************************************
//
//      Program:              MIPSInstructions.c
//      Author:               Dilesh Fernando <fernando.dilesh@gmail.com>
//                                    Cheng-Yeh Lee <chengyeh90@gmail.com>
//                                    Vuong Nguyen <nptvuong2912@gmail.com>
//                                    Diego Soliz Castro <zilosgodi@gmail.com>
//      Date:                 2017-04-21
//      Description:   Implementation of MIPS Instructions.
//
//*****************************************************************************

#include <stdio.h>
#include <stdlib.h>

#include "MIPSInstructions.h"

/*
 * NOTE: delete all print statements from each function afters testing.
 */

/*
 NOOP -- no operation
 Description:  Performs no operation.
 Operation:    advance_pc (4);
 Syntax:       noop
 Encoding:     0000 0000 0000 0000 0000 0000 0000 0000
 */
extern void NOOP() {
       // Do nothing
}

/*
 SLL -- Shift left logical
 Description:  Shifts a register value left by the shift amount listed in the instruction and
places the result in a third register. Zeroes are shifted in.
 Operation:    $d = $t << h; advance_pc (4);
 Syntax:       sll $d, $t, h
 Encoding:     0000 00ss ssst tttt dddd dhhh hh00 0000
 */
extern void SLL(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd, uint32_t ShiftAmt) {
       uint32_t s, t, d;
       RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);
       RegisterFile_Write(theRegisterFile, true, Rd, t << ShiftAmt);

}
```

```
/*
 SRL -- Shift right logical
 Description:  Shifts a register value right by the shift amount (shamt) and places the value in
the destination register. Zeroes are shifted in.
 Operation:    $d = $t >> h; advance_pc (4);
 Syntax:       srl $d, $t, h
 Encoding:     0000 00-- ---t tttt dddd dhhh hh00 0010
 */
extern void SRL(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd, uint32_t ShiftAmt) {
       uint32_t s, t, d;
       RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);
       RegisterFile_Write(theRegisterFile, true, Rd, t >> ShiftAmt);
}

/*
 SRA -- Shift right arithmetic
 Description:  Shifts a register value right by the shift amount (shamt) and places the value in
the destination register. The sign bit is shifted in.
 Operation:    $d = $t >> h; advance_pc (4);
 Syntax:       sra $d, $t, h
 Encoding:     0000 00-- ---t tttt dddd dhhh hh00 0011
 */
extern void SRA(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd, uint32_t ShiftAmt) {
       uint32_t s, t, d;
       RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);

       //Check the signed bit
       uint32_t signBit = t >> 4;

       //If sign bit is 1, Rt is signed else not signed
       if (signBit == 1) {
              // Rt is signed number

              //Generate a mask
              //Shift right by n bits.
              uint32_t mask = 31 << (5 - ShiftAmt); // 31(decimal) = 11111(binary)

              //AND the mask.
              RegisterFile_Write(theRegisterFile, true, Rd, (t >> ShiftAmt) & mask);

       } else {
              // Rt is unsigned number
              RegisterFile_Write(theRegisterFile, true, Rd, t >> ShiftAmt);
       }
}

/*
 SLLV -- Shift left logical variable
 Description:  Shifts a register value left by the value in a second register and places the
result in a third register. Zeroes are shifted in.
 Operation:    $d = $t << $s; advance_pc (4);
 Syntax:       sllv $d, $t, $s
 Encoding:     0000 00ss ssst tttt dddd d--- --00 0100
 */
extern void SLLV(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd) {
       uint32_t s, t, d;
       RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);
       RegisterFile_Write(theRegisterFile, true, Rd, t << s);
}
```

```
/*
 SRLV -- Shift right logical variable
 Description:  Shifts a register value right by the amount specified in $s and places the value
in the destination register. Zeroes are shifted in.
 Operation:    $d = $t >> $s; advance_pc (4);
 Syntax:       srlv $d, $t, $s
 Encoding:     0000 00ss ssst tttt dddd d000 0000 0110
 */
extern void SRLV(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
                uint32_t Rd) {
        uint32_t s, t, d;
        RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);
        RegisterFile_Write(theRegisterFile, true, Rd, t >> s);
}

/*
 ADD -- Add (with overflow)
 Description:  Adds two registers and stores the result in a register
 Operation:    $d = $s + $t; advance_pc (4);
 Syntax:       add $d, $s, $t
 Encoding:     0000 00ss ssst tttt dddd d000 0010 0000
 */
extern void ADD(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
                uint32_t Rd) {
        uint32_t s, t;
        RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);
        int32_t s_int = s;
        int32_t t_int = t;
        int32_t d = s + t;

        RegisterFile_Write(theRegisterFile, true, Rd, d);
}

/*
 ADDU -- Add unsigned (no overflow)
 Description:  Adds two registers and stores the result in a register
 Operation:    $d = $s + $t; advance_pc (4);
 Syntax:       addu $d, $s, $t
 Encoding:     0000 00ss ssst tttt dddd d000 0010 0001
 */
extern void ADDU(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
                uint32_t Rd) {
        uint32_t s, t;
        RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);
        RegisterFile_Write(theRegisterFile, true, Rd, t + s);
}

/*
 SUB -- Subtract
 Description:  Subtracts two registers and stores the result in a register
 Operation:    $d = $s - $t; advance_pc (4);
 Syntax:       sub $d, $s, $t
 Encoding:     0000 00ss ssst tttt dddd d000 0010 0010
 */
extern void SUB(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
                uint32_t Rd) {
        uint32_t s, t;
        RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);
        int32_t s_int = s;
        int32_t t_int = t;
        int32_t d = s - t;
```

```
        RegisterFile_Write(theRegisterFile, true, Rd, d);
}

/*
 SUBU -- Subtract unsigned
 Description:  Subtracts two registers and stores the result in a register
 Operation:    $d = $s - $t; advance_pc (4);
 Syntax:       subu $d, $s, $t
 Encoding:     0000 00ss ssst tttt dddd d000 0010 0011
 */
extern void SUBU(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
                 uint32_t Rd) {
        uint32_t s, t;
        RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);
        RegisterFile_Write(theRegisterFile, true, Rd, s - t);
}

/*
 AND -- Bitwise and
 Description:  Bitwise ands two registers and stores the result in a register
 Operation:    $d = $s & $t; advance_pc (4);
 Syntax:       and $d, $s, $t
 Encoding:     0000 00ss ssst tttt dddd d000 0010 0100
 */
extern void AND(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
                uint32_t Rd) {
        uint32_t s, t, d;
        RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);
        RegisterFile_Write(theRegisterFile, true, Rd, s & t);

}

/*
 OR -- Bitwise or
 Description:  Bitwise logical ors two registers and stores the result in a register
 Operation:    $d = $s | $t; advance_pc (4);
 Syntax:       or $d, $s, $t
 Encoding:     0000 00ss ssst tttt dddd d000 0010 0101
 */
extern void OR(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
                uint32_t Rd) {
        uint32_t s, t, d;
        RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);
        RegisterFile_Write(theRegisterFile, true, Rd, s | t);
}

/*
 XOR -- Bitwise exclusive or
 Description:  Exclusive ors two registers and stores the result in a register
 Operation:    $d = $s ^ $t; advance_pc (4);
 Syntax:       xor $d, $s, $t
 Encoding:     0000 00ss ssst tttt dddd d--- --10 0110
 */
extern void XOR(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
                uint32_t Rd) {
        uint32_t s, t, d;
        RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);
        RegisterFile_Write(theRegisterFile, true, Rd, s ^ t);
}

/*
```

```
 SLT -- Set on less than (signed)
 Description:  If $s is less than $t, $d is set to one. It gets zero otherwise.
 Operation:    if $s < $t $d = 1; advance_pc (4); else $d = 0; advance_pc (4);
 Syntax:       slt $d, $s, $t
 Encoding:     0000 00ss ssst tttt dddd d000 0010 1010
 */
extern void SLT(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd) {
       uint32_t s, t;
       RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);
       int32_t s_int = s;
       int32_t t_int = t;
       RegisterFile_Write(theRegisterFile, true, Rd, s_int < t_int);
}

/*
 SLTU -- Set on less than unsigned
 Description:  If $s is less than $t, $d is set to one. It gets zero otherwise.
 Operation:    if $s < $t $d = 1; advance_pc (4); else $d = 0; advance_pc (4);
 Syntax:       sltu $d, $s, $t
 Encoding:     0000 00ss ssst tttt dddd d000 0010 1011
 */
extern void SLTU(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t Rd) {
       uint32_t s, t;
       RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);
       RegisterFile_Write(theRegisterFile, true, Rd, s < t);
}

/*
 ADDI -- Add immediate (with overflow)
 Description:  Adds a register and a sign-extended immediate value and stores the result in a
register
 Operation:    $t = $s + imm; advance_pc (4);
 Syntax:       addi $t, $s, imm
 Encoding:     0010 00ss ssst tttt iiii iiii iiii iiii
 */
extern void ADDI(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
               uint32_t ImmediateValue) {
       // Initialized and assign local variables
       uint32_t s, t, imm;

       // Determine ImmediateValue is negative number
       const int isNegative = (ImmediateValue & (1 << 15)) != 0;

       // Assign imm value
       if (isNegative) {
              imm = ImmediateValue | ~((1 << 16) - 1);
       } else {
              imm = ImmediateValue;
       }

       RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);
       RegisterFile_Write(theRegisterFile, true, Rt, s + imm);
}

/*
 ADDIU -- Add immediate unsigned (no overflow)
 Description:  Adds a register and a sign-extended immediate value and stores the result in a
register
 Operation:    $t = $s + imm; advance_pc (4);
 Syntax:       addiu $t, $s, imm
```

```
 Encoding:     0010 01ss ssst tttt iiii iiii iiii iiii
 */
extern void ADDIU(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
                uint32_t ImmediateValue) {
        // Initialized and assign local variables
        uint32_t s, t, imm;

        // Determine ImmediateValue is negative number
        const int isNegative = (ImmediateValue & (1 << 15)) != 0;

        // Assign imm value
        if (isNegative) {
                imm = ImmediateValue | ~((1 << 16) - 1);
        } else {
                imm = ImmediateValue;
        }

        RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);

        // TODO Implement ignore overflow

        RegisterFile_Write(theRegisterFile, true, Rt, s + imm);
}

/*
 SLTI -- Set on less than immediate (signed)
 Description:  If $s is less than immediate, $t is set to one. It gets zero otherwise.
 Operation:    if $s < imm $t = 1; advance_pc (4); else $t = 0; advance_pc (4);
 Syntax:       slti $t, $s, imm
 Encoding:     0010 10ss ssst tttt iiii iiii iiii iiii
 */
extern void SLTI(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
                uint32_t ImmediateValue) {
        // Initialized and assign local variables
        uint32_t s, t, imm;

        RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);

        // Determine Rs register is negative number
        const int isNegativeRegister = (s & (1 << 31)) != 0;

        // Assign Rs value
        if (isNegativeRegister) {
                s = s | ~((1 << 32) - 1);
        } else {
                s = s;
        }

        // Determine ImmediateValue is negative number
        const int isNegative = (ImmediateValue & (1 << 15)) != 0;

        // Assign imm value
        if (isNegative) {
                imm = ImmediateValue | ~((1 << 16) - 1);
        } else {
                imm = ImmediateValue;
        }

        //RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);

        // Implement the instruction
        if ((int32_t) s < (int32_t) imm) {
```

```
                RegisterFile_Write(theRegisterFile, true, Rt, 1);
        } else {
                RegisterFile_Write(theRegisterFile, true, Rt, 0);
        }
}

/*
 SLTIU -- Set on less than immediate unsigned
 Description:  If $s is less than the unsigned immediate, $t is set to one. It gets zero
otherwise.
 Operation:    if $s < imm $t = 1; advance_pc (4); else $t = 0; advance_pc (4);
 Syntax:       sltiu $t, $s, imm
 Encoding:     0010 11ss ssst tttt iiii iiii iiii iiii
 */
extern void SLTIU(RegisterFile theRegisterFile, uint32_t Rs, uint32_t Rt,
                uint32_t ImmediateValue) {
        // Initialized and assign local variables
        uint32_t s, t, imm;

        // Determine Rs register is negative number
        const int isNegativeRegister = (s & (1 << 31)) != 0;

        // Assign Rs value
        if (isNegativeRegister) {
                s = s | ~((1 << 32) - 1);
        } else {
                s = s;
        }

        // Determine ImmediateValue is negative number
        const int isNegative = (ImmediateValue & (1 << 15)) != 0;

        // Assign imm value
        if (isNegative) {
                imm = ImmediateValue | ~((1 << 16) - 1);
        } else {
                imm = ImmediateValue;
        }

        RegisterFile_Read(theRegisterFile, Rs, &s, Rt, &t);

        // Implement the instruction
        if ((int32_t) s < (int32_t) imm) {
                RegisterFile_Write(theRegisterFile, true, Rt, 1);
        } else {
                RegisterFile_Write(theRegisterFile, true, Rt, 0);
        }
}
```

MIPS_Instructions_01.asm

```
#
#       This is a small test program for the EECS_645
#       ALUSimulator. All instructions just use the
#       register file.
#
#-----------------------------------------------------------------------
#
#
                        .text
```

```
Start:          addi                    $09,$02,100
                        addi                    $10,$05,-128
                        add                             $11,$01,$04
                        sll                             $12,$05,4
                        or                              $13,$04,$05
                        xor                             $14,$02,$03
                        sub                             $15,$03,$02
```

MIPS_Instructions_01.txt

```
20490064
20aaff80
00245820
00056100
00856825
00437026
00627822
```

Makefile

```
ALUSimulator: ALUSimulator_Main.o RegisterFile_01.o ALUSimulator.o MIPSInstructions.o
        cc -o ALUSimulator ALUSimulator_Main.o RegisterFile_01.o ALUSimulator.o
MIPSInstructions.o

ALUSimulator_Main.o: ALUSimulator_Main.c RegisterFile_01.h ALUSimulator.h
        cc -c ALUSimulator_Main.c

RegisterFile_01.o: RegisterFile_01.c RegisterFile_01.h
        cc -c RegisterFile_01.c

ALUSimulator.o: ALUSimulator.c ALUSimulator.h MIPSInstructions.h
        cc -c ALUSimulator.c

MIPSInstructions.o: MIPSInstructions.c MIPSInstructions.h
        cc -c MIPSInstructions.c

clean:
        rm -f *.o *~ ALUSimulator

clean-object-files:
        rm *.o

clear:
        cls

run:
        ./ALUSimulator
```