**T**he **A**merican **U**niversity in **C**airo

The School of Sciences and Engineering

CSCE 231/2303 – Assembly Language Programming

Dr. Mohamed Shalan

Project 1: RISC-V Simulator and Disassembler

Due Date: 10/07/2017

Submitted by:

Ahmed Ghazy ….. 900153471

Ahmed Ibrahim … 900153478

Amr Gouhar ……. 900153482

George Youssif … 900150531

**Introduction:**

In this project, we simulated the RISC-V 32I ISA in C++. The main idea revolves around the instruction formats in this ISA. The simulator gets the instructions of some program in binary and it disassembles it by translating the zeroes and ones into RISC-V 32I Assembly language. Further, we implemented an interface for it so that the simulator could be invoked from the command line prompt. We divided the code into functions for each instruction format and we had a function for the I/O specified in the prompt. The details of the implementation are explained in the following section.

**Introducing RISC-V Instruction Set Architecture:**

RISC-V (pronounced “risk five”) is an open source implementation of RISC (reduced instruction set computing) based ISA (instruction set architecture.

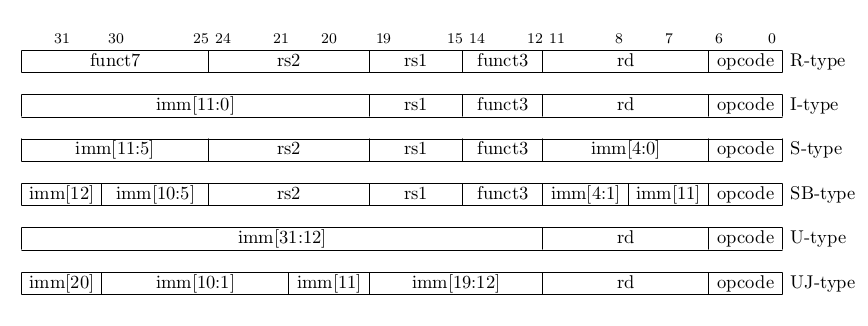
Most ISAs are commercially protected by patents, preventing practical efforts to reproduce the computer systems. In contrast, RISC-V is open, permitting any person or group to construct compatible computers and use associated software.

RISC-V has 32 integer registers– Register number 0 is a constant 0 and all arithmetic and logic operations occur between registers. The instructions must be aligned to 32-bit addresses and the memory is addressed by 8-bit bytes and

the only instructions that can access main memory are loads and stores

The instruction set includes other features to increase a computer's speed, while reducing its cost and power usage which includes placing most-significant bits at a fixed location to speed sign- extension, and a bit-arrangement designed to reduce the number of multiplexers in a CPU

RISC-V does not detect or flag most arithmetic errors, including overflow, underflow and divide by zero. RISC-V intentionally lacks condition codes, and even lacks a carry bit Instead RISC-V builds comparison operations into its conditional- jumps

Figure (1): RISC-V base instructions formats.

**Implementation:**

**Introduction:**

First of all, in order to simulate the byte-addressable memory of the RISC-V ISA, we used an array of type character of size 8 K-Bytes. Also, for the 32-bit registers, we used an array of type integer of size 32 because the ISA has 32 registers. Then, there is a variable for the program counter (pc) that points at the instruction location in memory. We have in our implementation one main function that gets the instruction word from the binary file, then decides on its format based on its opcode. This opcode is always located at the most right 7 bits of the instruction word, regardless of its format. Based on this opcode, this function sends the instruction word to the corresponding function that deals with its format. The prototypes of the functions are shown below followed by an explanation for each one.

**Function Prototypes and Explanation:**

1. **void instDecExec(unsigned int instWord);**

This function, as explained earlier, decides on the type of the instruction based on the opcode. It gets the opcode by ANDing the instruction word with a bit mask equal to 0x0000007F. This mask was chosen as it turns off all the bits after the 7th bit and sets the least significant 7 bits to their values in the instruction word. After getting the opcode, it sends the instruction word, along with the opcode, to the corresponding function that disassembles it.

1. **void R(unsigned int instWord, unsigned int opcode);**

This function deals with R-type (Register type) instructions. These include addition, subtraction and logical operations. It follows the same technique explained in the previous function by using the appropriate bit masks in getting the values of the destination register, source registers and function (Figure 1).

rd = (instWord >> 7) & 0x0000001F;

funct3 = (instWord >> 12) & 0x00000007;

rs1 = (instWord >> 15) & 0x0000001F;

rs2 = (instWord >> 20) & 0x0000001F;

funct7 = (instWord >> 25) & 0x00007F;

Figure 1: Bit Manipulation used to decode the R-type instructions

After determining the registers on which the operation specified by the function should be performed, it applies this operation.

1. **void U\_UJ(unsigned int instWord, unsigned int opcode);**

This function deals with U-type instructions (lui, load upper immediate, and auipc, load upper immediate to pc) and UJ instructions (jal, jump and link unconditional). It gets the bits corresponding to the destination register and the U/UJ immediate the same way it does with other instructions, using bit manipulation. It performs signed-extension on the immediate once it extracts it from the instruction word (Figure 2).

opcode = instWord & 0x7F;

rd = (instWord >> 7) & 0x1F

U\_imm = ((instWord >> 12) & 0xFFFFF)| (((instWord >> 31) ? 0xFFF00000 : 0x0)); // signed-extension

Figure 2: Bit Manipulation used to decode the U-type instructions

The UJ immediate differs from this because its bits are not stored in order in the instruction word (Figure 3).

U\_imm = (((U\_imm >> 19) << 19) | ((U\_imm & 0xFF) << 11) | (((U\_imm >> 8) & 0x1) << 10) | ((U\_imm >> 9) & 0x3FF)); // bits from 1 - 31

U\_imm = U\_imm << 1; // adding bit 0 = 0

Figure 3: UJ Immediate

1. **void I(unsigned int instWord, unsigned int opcode);**

This function deals with I-type instructions which are mainly load instructions, addition, subtraction, logical operations that include immediate and the jalr instruction. The break down of the instruction format is done using bit manipulation as shown in Figure 4.

rd = (instWord >> 7) & 0x1F;//0x1F = 0b1\_1111

funct3 = (instWord >> 12) & 0x7; //0x7 = 0b111

rs1 = (instWord >> 15) & 0x1F;

rs2 = (instWord >> 20) & 0x1F; // — inst[31] — inst[30:25] inst[24:21] inst[20]

I\_imm = ((instWord >> 20) & 0x7FF) // read the first 11 bits

| (((instWord >> 31) ? 0xFFFFF800 : 0x0)); //perform sign extension

Figure 4: I-type format break down

1. **void S(unsigned int instWord, unsigned int opcode);**

This function deals with the store instructions. It extracts the index of memory in which the data byte, half word or word should be stored and stores it there.

rd = (instWord >> 7) & 0x0000001F;

funct3 = (instWord >> 12) & 0x00000007;

rs1 = (instWord >> 15) & 0x0000001F;

rs2 = (instWord >> 20) & 0x0000001F;

S\_imm = ((((instWord >> 7) & 0x1F) | ((instWord & 0xFE000000)) >> 20) | (((instWord >> 31) ? 0xFFFFF000 : 0x0))); // (index)

Figure 5: S-type break down

1. **void SB(unsigned int instWord, unsigned int opcode);**

This function deals with branching instructions. It extracts the destination register, 2 source registers and the immediate (offset) value from the instruction word. Then evaluates the condition based on the type of branching.

1. **void ecall(unsigned int instWord);**

This function deals with ecalls. It extracts the type of operation from regs[17], then evaluates the call type. Based on this it executes one of the following instructions:

10  exit the program

1  print the integer in regs[10]

5  read an integer into regs[10]

4  print the string with its base address in regs[10]

**Sample**

-Compiled using:

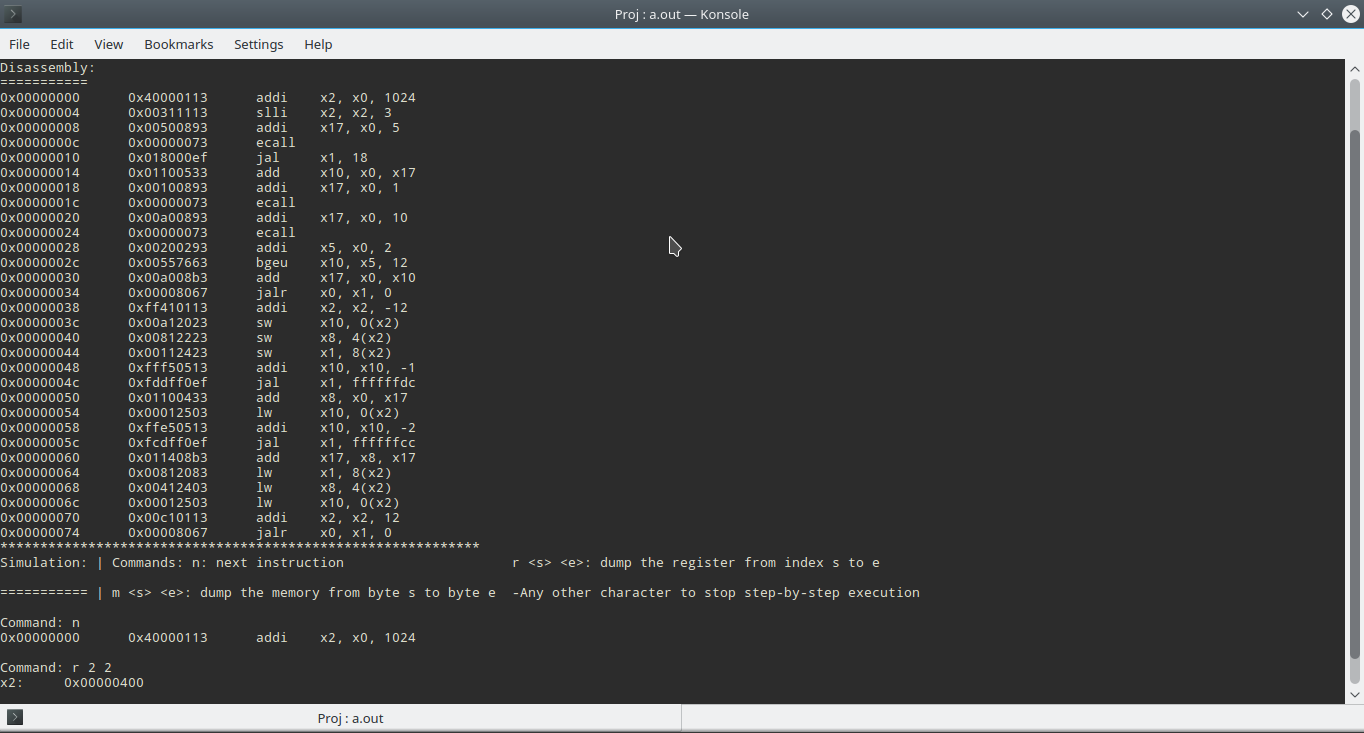
g++ disassembler.cpp -std=c++11

-Command line format is as follows:

./a.out <file\_name> [-no\_r\_dump] [-mem\_dump]

with the last 2 flags being optional, the first removing the register dump at the end of execution, and the other adding a memory dump at the end of it.

Sample Output:

Binaries used: fib.bin