ARM Processor Simulator

By: Melchisedek dulcio

Fall 2020 Microprocessor architecture

2020

TABLE OF CONTENTS

Table of Contents 1

Introduction 2

Features 2

Software prerequisites 3

Build and Test

Configuration

User Guide

Software Architecture

Bug Report

Appendices

Abstract

The ARM Simulator is a semester project for the course Microprocessor Architecture, that is designed for students to demonstrate their understanding of the microprocessor architectures by developing an application that simulates the work of an ARM microprocessor. Since the project covered an entire semester’s worth of work, this report summarizes what was accomplished the following sections:

1. Features: in addition to GUI overview, provides detail on which instructions, instruction addressing modes, and I/O capabilities are supported
2. Software Prerequisites: lists the OS and software needed for the application to run
3. Build & Test: shows how to compile the project along with compiling and running any unit tests
4. Configuration: shows how to configure files to turn logging on or off and redirecting the output to files of choice.
5. User Guide: demonstrates how to use the simulator, with details on how to create files to use with the simulator.
6. Software Architecture: provides a discussion on the design and layout of the program code.
7. Bug Report: discusses bugs that were present at the time of the assignment submission.
8. Appendices: provides insight on the development of the project and any lists significant source code.

Features

Instructions Supported:

1. Data Processing: MOV, MOVS, MVN, AND, ORR, EOR, BIC, MUL, ADD, SUB, RSB, CMP
2. Load / Store: LDR, STR, LDM, STM
3. Branch: B, BL, BX
4. Status Registers: MSR, MRS
5. Exception: SWI

Addressing Modes Supported:

1. Register shifted by Immediate
2. 12-bit Immediate

GUI Features:

1. ToolBar with the following buttons: load file, enable/disable trace, reset, add breakpoint, run, stop, step
2. Display panels: RAM with addressing functionality, disassembly view, console (fully functional I/O), stack view, processor mode view, NZCVI CPSR flags display, registers
3. Status bar: status of program, filename, checksum

Software Prerequisites

Platform:

1. Windows 10

Software Required:

1. Visual Studio 2017/2019
2. Developper Command Prompt for VS 2019
3. NUnit3 Console (for running unit tests)

Build & Test

How to Compile the solution from the command line:

**NOTE:** To use the Windows CMD, the user must have .NET frameworks added to the system environment variables.

1. Open the Windows command prompt (see *Note*) or the Developer Command Prompt for VS 2019 and navigate to the folder containing the solution for this project
2. Run the command *msbuild* to compile the solution, and the *executable* will be placed in the *bin\Debug*.

How to Compile and Run Unit Tests:

1. Install NUnit3-Console from *NUnit.org* and add to the system or environment path variable.
2. Open the Developer Command Prompt for VS 2019, also known as the `*x64 Native Tools Command Prompt for VS 2019*`, or any command prompt of choice.
3. Navigate to the location containing the project solution which will also contain the *armsim.csproj* file.
4. Run *nunit3-console armsim.csproj* and watch for the results.

Configuration

To configure the logging framework, locate the *armsim.exe.config* file that located in the folder as the executable file, and follow directions in the comment at the top that tells the user how to turn the logging on or off.

To redirect the logging output to a file, follow the comments in section in the config file labeled "This controls logging section output.”

User Guide

*The usage for ARMSim from the command line*

In the command line of your choice, preferably Developer Command Prompt for VS 2019, navigate to the location of the executable file, called *armsim.exe*.

The usage to the run the application is: *armsim.exe [ --mem memory-size ] [--traceall] [[--exec] elf-file]*, where *mem* is the optional memory size specification for program, *--exec* indicates whether to start executing the file, *--traceall* indicate whether to produce a trace of all CPSR modes and *elf-file* is the executable the user wants to load its contents.

Features Detailed:  
Toolbar:

|  |
| --- |
|  |

* Buttons:
  + Load File: Opens dialog to select file from a location.
  + Trace: Enables or disables trace
  + Reset: Zeroes out all registers and resets PC, CPSR to original state
  + Break Point: Opens a window that prompts for an address
  + Run: Steps through the program until halt instruction encountered
  + Stop: Stops execution of a running programming
  + Step: Fetches, decodes, and executes the single instruction pointed to by the PC.

Displays:

1. Memory: Display addressable RAM grid representation

|  |
| --- |
|  |

1. Disassembly: Displays Disassembly of instructions before and after PC

|  |
| --- |
|  |

1. Stack: Displays the couple top words of the stack

|  |
| --- |
|  |

1. Registers: Displays registers’ status

|  |
| --- |
|  |

1. Processor modes: Show the current mode of the program running

|  |
| --- |
|  |

1. CPSR: Displays the state of the NZCV and I bit of the CPSR register

|  |
| --- |
|  |

Terminal: Allows for user input I/O into program

|  |
| --- |
|  |

Software Architecture

Design:

|  |
| --- |
| Graphical user interface, diagram, application  Description automatically generated |

Classes and Design Decisions:

*Memory*

The Memory class is used in the Computer, CPU, and Instruction classes to simulate memory for RAM and our registers.

*Computer*

The Computer class holds instances of CPU, RAM (Memory), and registers (Memory) and defines the generic run, step, and reset that should be available with any computer simulator.

*CPU*

The CPU class is responsible for defining the standard fetch-decode-execute functions for a CPU to process an instruction and defines exception handling for the exception throwing instructions like SWI.

*Instruction*

Instruction is an abstraction of the different types of instructions. The CPU directly accesses this class to not have to mind the details of the different subclasses of Instructions and their subclasses. The Instruction class defines the following 3 instructions that each class must implement: DecodeInst(), decodes instruction when called; Execute(), executes instruction; ToString(), for representing the disassembly of the instruction.

*Derived Classes*: DataProcess, LoadStore, Branch, SWI, MSR, MRS

*Operand2*

The Operand2 class is used by various types of instructions that use and operand2 field in their instruction encoding.

*Offset*

The Offset class defines the different addressing modes for the instructions that access memory.

*BarrelShift*

The Operand2 and Offset classes use the BarrelShift class to perform bit shifting.

One of the bigger struggles of this project was know what to put where. Through asking questions like, “Is \_\_\_\_\_\_ a type of \_\_\_\_\_\_?” or “Does \_\_\_\_\_\_\_ use \_\_\_\_\_\_\_?” The first question would answer whether or not inheritance would be used as in the case of the Instruction classes, and the second would answer whether it should be utility function or class as in the case of creating separate classes for Operand2, Offset, and BarrelShift. The rest of the design was simply then looking at the different components of the ARM microprocessor architecture and implementing each component.

To maintain model-view separation, I decided to use Databinding for the various components of the GUI to different component in Computer. The tricky part was careful coding to make sure that no race conditions were occurring, especially when using multithreading to allow for adding a breakpoint and running the program while having the main window remain responsive.

For the terminal I/O, I used an event handler to add set the IRQ pin variable in the CPU to high to notify the CPU that a character has been typed, that later sets the PC to the proper address in the vector table and lets the OS take it from there. The console device itself is bound to a property that is updated when input handling is done by the loaded OS.

Bug Report

Files Analysis:

1. test1.exe – no difference
2. test2.exe – no difference
3. test3.exe – no difference
4. btest.exe – no difference
5. ctest.exe – no difference
6. test1.exe – no difference
7. branch.exe – no difference
8. cmp.exe – no difference
9. locals\_no\_io.exe – no difference
10. quicksort\_no\_io.exe – no difference
11. mersenne\_no\_io.exe – no difference

Known issues:

1. There will be times when a run has completed that the displays will not update until the user presses the step button.
2. The terminal cursor moves to the beginning of the text upon each update.

Appendix