Armsim Technical Report

Bob Jones University
CPS 310

David Polar

December 12, 2021

Contents

Introduction

This technical report aims to outline the various technical aspects of the ARM processor simulator. It will be begin with a **Features** section, presenting its features arranged according to C-level, B-level, and A-level implementations. The **Software Prerequisites** section will describe the system and software requirements necessary for running the program. The **Build and Test** section will provide instructions on how to compile and execute the project. The **Configuration** section will additionally contain instructions on how to modify the various configurations included in the program. The **User Guide** section is a comprehensive guide to the features of the simulator. The **Software Architecture** section will discuss the design and organization of code in the project, including a UML class diagram. The **Bug Report** will compare the expected results in the log files for the provided test executables to the actual results of the program. The **Appendices** will include the project journal, git log, and source code listings.

Features

Features Listed by A, B, & C Levels

C-Level

All C-level features have been implemented for each version of the simulator. These include, for each verison—

Loader

- A loader which correctly interprets and loads an ELF file into virtual memory and additionally computes the correct checksum based on that virtual memory
- Valid command line parsing using a third party library
- Suitable methods for interacting with virtual memory

GUI

- A functioning GUI will all the necessary display panels
- Loading and Single-Step capabilities via the GUI

Execution I

- An accurate disassembly panel
- Correct trace output for the C-level ARM subsection
- The optional `--exec` command line argument

Execution II

- A Keyboard Device that interacts with the user via dialog box
- Accurate disassembly for the complete ARM subset specified

B-Level

All B-level features have been implemented for each version of the simulator. These include, for each verison—

Loader

- A scrollable memory grid
- Unit tests

GUI

- Multithreadded Run and Stop capabilites via the GUI
- Reset capabaility
- Shortcut keys

Execution I

- Correct trace output for the B-level ARM subsection

Execution II

- Complete model-view separation
- A multi-threaded approach to the GUI operations

A-Level

All A-level features have been implemented for each version of the simulator except for Execution II. These include, for each verison—

Loader

- A logging framework
- Extensive error handling

GUI

- Breakpoints
- Panel resizing

Execution I

- Exhaustive unit tests for the decode and execute tasks

Supported ARM Instructions & Addressing Modes

Data Processing

Instructions

- MOV
- MVN
- ADD
- SUB
- RSB
- MUL
- AND
- ORR
- EOR
- BIC

Addressing Modes

- Barrel shifter with: lsl, lsr, asr, and ror
- Operand2 with: immediate, register with immediate shift, and register with register shift

Load/Store

Instructions

- LDR
- STR
- LDM
- STM

Addressing Modes

- LDR/STR: word and unsigned byte, preindexed, with and without writeback
- LDM/STM: FD variant, with and without writeback

Miscellaneous

- SWI
- CMP
- B
- BL
- BX

IO Capabilites

When a process encounters the SWI #0x00 instruction, it writes to the simulator's console display panel. When it encounters the SWI #0x6a instruction, it opens a dialog box and requests for input from the user. This input is written to the specified location in memory and reflected into the console display panel as well.

GUI Overview

The GUI contains the buttons, along with their keyboard shortcuts, Load, Reset, Breakpoints, Run, Stop, Step, and Trace a program. This encapsulates all the funcionality outlined by the requirements for the simluator. The Run and Stop functions are run on different threads to prevent the GUI from locking up. After the operations are completed, the display panels are updated. The display panels included are Memory, Stack, Disassembly, Console, Registers, and Flags.

Software Prerequisites

This program is intended to run on the Windows 10 operating system. The software packages necessary for running this program are thus—

- Visual Studio 2019
- MSTest.TestAdapter
- MSTest.TestFramework
- CommandLineParser distributed by gsscoder,nemec,ericnewton76,moh-hassan
- log4net distributed by The Apache Software Foundation

Build and Test

Via IDE

Verify that program will be building in Release mode. To add command line arguments, go to Project → armsim Properties. Click on the Debug pane and, under Start options, enter the command line arguments. To run the unit tests, go to Test → Run All Tests. This will open Test Explorer and display the test results upon completion.

Via Command Line

Open the command prompt using one of two options—

1. Searching for Developer Command Prompt for VS 2019 in the Windows Search Bar
2. Going to the list of applications and finding Visual Studio 2019

Navigate to the project solution file. In the project hierarchy, it is located in src → armsim. Enter `msbuild armsim.sln` to build the project. This will create the executable file in src → armsim → armsim → bin → Debug. To run the program, navigate to the Debug folder and enter `armsim.exe` with the appropriate command line arguments.

Configuration

Logging is, by default, turned on and being directed to the file armsim.log. To turn off logging, navigate to the file Loader.cs in src → armsim. Find the `loggingEnabled` variable at the top of the Loader class and change its value to `false`. The armsim.log file will still be created at build time but nothing will be written to it.

User Guide

Run the program following the instructions outlined in the section Build and Test. The usage for the command line is `armsim [ --mem memory-size ] [ --exec ] [ elf-file ]`, where `memory-size` must be between 0 and 1,000,000, and `elf-file` must be an ELF file. The application will start and display the GUI. If a file was provided in the command line, it will load and display the content of the file. If there was no file provided, the only enabled button will be Load. To load a file, either click Load or use the key shortcut Ctrl-O.

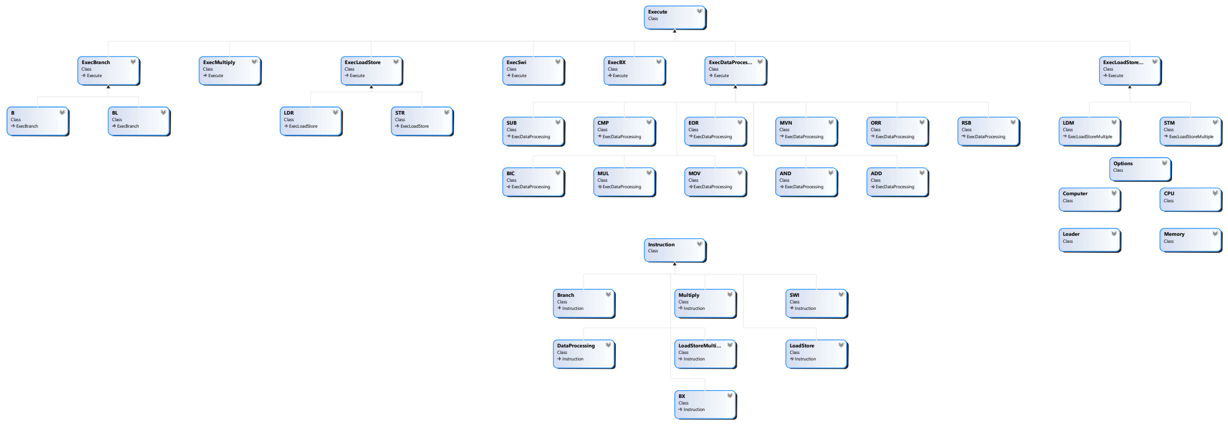
Once a file is loaded, enter breakpoints by clicking the Breakpoints button, or using the key shortcut Ctrl-B. This will open a dialog box which requests an address at which to insert the breakpoint. The address, machine language instruction, and disassembly of the entered address will appear as an entry in the breakpoints list. The breakpoint may also be removed by clicking on the breakpoint entry in the list and subsequently clicking Remove.

To execute the program that has been loaded into virtual memory, there are two options. Click Run, or use the keyboard shortcut F5 to execute the process until it encounters a breakpoint or the end of the program. During this process, the Stop button is enabled, along with the keyboard shortcut Ctrl-Q, which can also be used to stop the program's execution. The other option to execute the program is to click Step, or use the keyboard shortcut F10 to execute only one instruction in the program.

If at any point the program needs to be reset, merely press Reset, or use the keyboard shortcut Ctrl-R, to reset the program back to its original state.

Software Architecture

The design of this project maintains model-view separation using events. The third party libraries used can be found in the section Software Prerequisites. Additional threads are created when running or stepping through the program. IO has been discussed in the section IO Capabilites. The relationships between the model classes are shown in the UML diagram on the next page.



Execute
Class
ExecBranch
Class
Execute
ExecMultiply
Class
Execute
ExecLoadStore
Class
Execute
ExecSwi
Class
Execute
ExecBX
Class
Execute
ExecDataProces...
Class
Execute
ExecLoadStore...
Class
Execute
B
Class
ExecBranch
BL
Class
ExecBranch
LDR
Class
ExecLoadStore
STR
Class
ExecLoadStore
SUB
Class
ExecDataProcessing
CMP
Class
ExecDataProcessing
EOR
Class
ExecDataProcessing
MVN
Class
ExecDataProcessing
ORR
Class
ExecDataProcessing
RSB
Class
ExecDataProcessing
LDM
Class
ExecLoadStoreMultiple
STM
Class
ExecLoadStoreMultiple
BIC
Class
ExecDataProcessing
MUL
Class
ExecDataProcessing
MOV
Class
ExecDataProcessing
AND
Class
ExecDataProcessing
ADD
Class
ExecDataProcessing
Options
Class
Computer
Class
CPU
Class
Loader
Class
Memory
Class
Instruction
Class
Branch
Class
Instruction
Multiply
Class
Instruction
SWI
Class
Instruction
DataProcessing
Class
Instruction
LoadStoreMulti...
Class
Instruction
LoadStore
Class
Instruction
BX
Class
Instruction

Bug Report

The simluator produces the correct trace output for the test files ctest.exe, btest.exe, cmp.exe, branch.exe, and locals_no_io.exe. This is all that was necessary for the grade levels I attempted.

Appendices

Project Journal

The project journal may be found here

Git Log

```
commit ef23846f7e1f8143b9f55945558b11361f0f9cb0
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Sun Dec 12 23:48:20 2021 -0500

    correct exe in install folder

commit afcbe910f6cfc026419492f65b85360c4dbff4a7
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Sat Dec 11 00:11:25 2021 -0500

    io done

commit 558303a65edd55aae105851196d9e002fc6d6b5a
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Thu Dec 9 22:57:54 2021 -0500

    passes branch test

commit 0e741bd11a0433415325aeeabf42c722b084d8d4
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Wed Oct 27 23:46:26 2021 -0400

    branch trace passes

commit 80995b6fa38aedc23bf919322239cf8d402122aa
Author: Blakthorne <dlpolar38@gmail.com>
```

```
Date:   Wed Oct 27 10:29:50 2021 -0400

    cmp passes tracing tests

commit f7aaadd3f1c88b45b22da5c4c5a0615e1ac0022e
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Tue Oct 26 22:58:30 2021 -0400

    almost done with cmp

commit 6b55cbef19e1ecb467fd25bc0726f7460d2dcc00
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Fri Oct 22 15:38:02 2021 -0400

    fix read_me

commit d4ddd649f81c80b6421d0f4abc95109e0d0b6537
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Fri Oct 22 15:36:03 2021 -0400

    load and store work

commit 542a8d62175adbd0052f1a1da77baaa6d46f4747
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Thu Oct 21 23:51:23 2021 -0400

    load store executing

commit 224c06362d711ebf39be4a62907b26fbcac8e711
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Wed Oct 20 23:55:27 2021 -0400

    resubmission

commit 98f9e72b2d1ac0a468a623480e7899e5cae5b192
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Wed Oct 20 23:53:33 2021 -0400

    submission

commit 201e5874c0a9ecb11e2ebd135471199bbfc48582
Author: Blakthorne <dlpolar38@gmail.com>
```

```
Date:   Wed Oct 20 10:22:01 2021 -0400

    tracing works

commit 5892931a44b446a3f0aa4d3d80693d48fcff967c
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Mon Oct 18 12:31:58 2021 -0400

    barrel shifter done and some text reps

commit b95a3614b8e5773b5dc107086c39f1cfcdf24f4e
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Fri Oct 15 21:52:18 2021 -0400

    decode started working

commit 122a03935eb11b84b33adce4b4f1c771a926c3d4
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Thu Oct 14 22:44:35 2021 -0400

    refactored system

commit 558470f4fcc03483cd063200b70150503449ef59
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Sat Oct 9 22:15:19 2021 -0400

    working on fde

commit 6302fba6340e67c4b70ceb564a1a55cd70ec7e8d
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Fri Sep 24 22:44:23 2021 -0400

    exec switch working

commit 806fddd18983c8779ef70da17c2e41d2c8974c3a
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Wed Sep 22 23:43:15 2021 -0400

    trace button working with key shortcuts

commit 509e401870a83a0f701d0c857681e30d63ec0755
Author: Blakthorne <dlpolar38@gmail.com>
```

```
Date:   Tue Sep 21 21:36:26 2021 -0400

    updated comments

commit d3d79c13dd5ffe7b654672a1dee190994de47bfe
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Sat Sep 18 23:49:05 2021 -0400

    gui finished

commit 1bf6eeb043f88becb00706a8c4e247644c396887
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Sat Sep 18 23:26:55 2021 -0400

    breakpoints actually working now

commit 65f0161ccfd392702f6fe86ef85eec62ebd3bbd6
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Sat Sep 18 18:55:15 2021 -0400

    breakpoints working

commit 8fae8bc74ba64fb3bf0f9485d069fe83f41d6a96
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Fri Sep 17 23:20:11 2021 -0400

    key shortcuts and disassembly panel

commit a5f7d2760ea87847e88131ed1242148d9a8ec55d
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Tue Sep 14 23:16:13 2021 -0400

    dialog box

commit 4f295b69e407fec7a194a3914fe27bc2b205e2bc
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Tue Sep 14 21:22:56 2021 -0400

    good layout

commit 7b54d6dae0a5f5dafddea72b07a51a05bf6ee0c7
Author: Blakthorne <dlpolar38@gmail.com>
```

```
Date:   Fri Sep 10 14:49:23 2021 -0400

    Gutter look good

commit 5388799183328bce48a9708f37ca8b30927760c3
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Fri Sep 10 12:48:54 2021 -0400

    Tabs implemented

commit 6b8dfae65d287111b200b25632aed5dd5c9a7dd8
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Sun Sep 5 18:15:38 2021 -0400

    gui changes migrated

commit 0074735c8ecbb612d48e198281a9b79b04ef4680
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Sat Sep 4 23:53:43 2021 -0400

    fixed error in docs

commit 0377de7e9150620bce2feea2badd0b7e98050d7a
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Sat Sep 4 23:52:52 2021 -0400

    finished loader

commit 19890074ccf0d7c88a5bdddcc95ca2137909bfd3
Author: Blakthorne <dlpolar38@gmail.com>
Date:   Wed Sep 1 23:17:46 2021 -0400

    it works

commit 2eef418dd3280b777d97653dfe6f5f07b97509d4
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
Date:   Wed Sep 1 16:27:45 2021 +0000

    Setting up GitHub Classroom Feedback

commit cd7f734aca90d98fcc16f2457b85cedec769fdc2
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
```

```
Date:   Wed Sep 1 16:27:45 2021 +0000

    GitHub Classroom Feedback

commit 6f74f5fd06e74cb12f8f39a2e38510f639d16937
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
Date:   Wed Sep 1 16:27:44 2021 +0000

    GitHub Classroom Autograding Workflow

commit 99c056d3b3eec04f11d2eb58ad12bf884fa376cc
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
Date:   Wed Sep 1 16:27:44 2021 +0000

    GitHub Classroom Autograding

commit 2480ef4e021559971643408d6e4b5cf93a8acf88
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
Date:   Wed Sep 1 16:27:42 2021 +0000

    Initial commit
```

Model Source Code Listings

The source code listings begin on the next page.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace armsim
{
    class Branch : Instruction
    {
        /// <summary>
        /// Represents the information found in data processing instruction
        /// </summary>
        public Branch(uint instr) : base(instr)
        {
            l_b = FindL();
            offset_b = FindOffset();
        }

        /// <summary>
        /// Extracts the L bit and
        /// stores in the instance variable l
        /// </summary>
        private uint FindL()
        {
            return Memory.ShiftToEnd(instr, 24, 24);
        }

        /// <summary>
        /// Extracts the first 24 bits of the encoding
        /// </summary>
        public int FindOffset()
        {
            int offset_b = checked((int)Memory.ShiftToEnd(instr, 0, 23));
            int isNeg = offset_b & 0x800000;
            if (isNeg != 0)
            {
                offset_b |= 0x3F000000;
            }
            int shifted = offset_b << 2;
            return shifted;
        }

        /// <summary>
        /// Computes the final target address for the branch
        /// </summary>
        public int FindTarget()
        {
            return offset_b + (int)pc_b;
        }

        /// <summary>
        /// Produce the textual representation
        /// </summary>
        public void Disassemble()
        {
            target_b = FindTarget();

            text = "b";

            if (l_b == 1)
            {
                text += "l";
            }

            text += FindCondString() + " #" + string.Format("0x{0:X}", target_b);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace armsim
{
    class BX : Instruction
    {
        /// <summary>
        /// Represents the information found in bx instruction
        /// </summary>
        public BX(uint instr) : base(instr)
        {
            rm_bx = FindRm();
            bit0_bx = FindBit0();

            Disassemble();
        }

        /// <summary>
        /// Extracts the Rm nibble and
        /// stores in the instance variable rm
        /// </summary>
        private uint FindRm()
        {
            return Memory.ShiftToEnd(instr, 0, 3);
        }

        /// <summary>
        /// Extracts the bit 0
        /// </summary>
        private uint FindBit0()
        {
            return Memory.ShiftToEnd(instr, 0, 0);
        }

        /// <summary>
        /// Produce the textual representation
        /// </summary>
        public void Disassemble()
        {
            text = "bx" + FindCondString() + " r" + rm_bx;
        }
    }
}
```

```
// Computer.cs
// Holds instances of the Memory class for
// RAM and registers.
// Contains methods for run and step.

using System;
using System.Collections.Generic;
using System.Data;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace armsim
{
    public class Computer
    {
        const int NUM_REGISTERS = 17 * 4;   // number of registers; r0-r15 and C
PSR
        const int PC_REG = 15 * 4;          // start of program counter word
        const int SP_REG = 13 * 4;          // start of stack pointer word
        private int memSize;                // size of memory
        private string fileName;            // name of ELF file
        public Memory ram;                  // instance of Memory class to simul
ate the virtual RAM
        public Memory registers;            // instance of Memory class to simul
ate the registers
        public CPU processor;               // instance of CPU class
        public Loader load;                         // instance of the Loader cla
ss
        private static int retCode;         // code that returns to View
        public uint textStart;       // start address of the .text section
        public uint textSize;        // size in bytes of the .text section
        public static bool finished;    // indicates whether the cpu is finished
 executing

        public event EventHandler RunCompleted; // delegate for handling the OnR
unCompleted event

        /// <summary>
        /// Constructor for the Computer class
        /// </summary>
        /// <param name="_memSize">size in bytes of virtual memory</param>
        /// <param name="_fileName">name of ELF file to load</param>
        public Computer(int _memSize, string _fileName)
        {
            memSize = _memSize;
            fileName = _fileName;
            LoadFile();
            finished = false;
        }

        /// <summary>
        /// Runs through the program beginning at the address
        /// specified in the program counter and performs the
        /// fetch, decode, execute cycle until fetch return 0
        /// </summary>
        public void run()
        {
            while (!finished)
            {
                foreach (MainWindow.breakRow row in MainWindow.breakList)
                {
                    if (string.Format("{0:X8}:", registers.ReadWord(PC_REG)) == r
ow.breakAddr)
                    {
                        processor.running = false;
                        OnRunCompleted(EventArgs.Empty);
                        return;
                    }
```

```
                }

                uint instrWord = processor.fetch();
                registers.WriteWord(PC_REG, registers.ReadWord(PC_REG) + 4);
 // update PC

                Instruction instr = processor.decode(instrWord);
                processor.execute(instr);
            }

            processor.running = false;
            OnRunCompleted(EventArgs.Empty);
        }

        /// <summary>
        /// Performs one fetch, decode, execute cycle
        /// at the address specified in the program counter
        /// </summary>
        public void step()
        {
            if (!finished)
            {
                uint instruction = processor.fetch();
                registers.WriteWord(PC_REG, registers.ReadWord(PC_REG) + 4);
 // update PC
                Instruction instr = processor.decode(instruction);
                processor.execute(instr);

                processor.running = false;
                OnRunCompleted(EventArgs.Empty);
            }
        }

        /// <summary>
        /// Creates:
        ///     an instance of Memory for virtual RAM,
        ///     an instance of Memory for registers,
        ///     an instance of CPU for processing the information,
        ///     an instance of Loader for loading the ELF file into virtual memo
ry
        /// </summary>
        public void LoadFile()
        {
            ram = new Memory(memSize);
            registers = new Memory(NUM_REGISTERS);
            processor = new CPU(ram, registers);
            load = new Loader(memSize, fileName, ram, this);
            registers.WriteWord(PC_REG, GetPC());
            registers.WriteWord(SP_REG, 0x7000);
            retCode = load.GetRetCode();
        }

        /// <summary>
        /// Getter method to retrieve the initial program counter
        /// value from the load instance variable
        /// </summary>
        public uint GetPC()
        {
            return load.GetElfEntry();
        }

        /// <summary>
        /// Getter method to retrieve the stack pointer
        /// </summary>
        public uint GetSP()
        {
            return registers.ReadWord(SP_REG);
        }

        /// <summary>
        /// Getter method to retrieve the retCode instance variable
        /// </summary>
        public int GetRetCode()
        {
            return retCode;
        }

        /// <summary>
        /// Getter method to retrieve the memSize instance variable
        /// </summary>
        public int GetMemSize()
        {
            return memSize;
        }

        /// <summary>
        /// Makes CalculateChecksum() in the Memory class accessible to MainWind
ow.xaml.cs
        /// </summary>
        /// <returns>Checksum as an int</returns>
        public int GetChecksum()
        {
            return ram.CalculateChecksum();
        }

        /// <summary>
        /// Event handler method
        /// </summary>
        /// <param name="e"></param>
        public virtual void OnRunCompleted(EventArgs e)
        {
            RunCompleted?.Invoke(this, e);
        }
    }
}
```

```
// CPU.cs
// Perform the fetch-decode-execute cycle
// using the RAM and registers instances
// of the Memory class.

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;
using System.Runtime.InteropServices.WindowsRuntime;
using System.Windows.Media.TextFormatting;

namespace armsim
{
    public class CPU
    {
        public Memory ram;              // instance variable referencing the vir
tual RAM in the Computer class
        public Memory registers;        // instance variable referencing the reg
isters in the Computer class
        const int PC_REG = 15 * 4;      // start of program counter word
        public bool running = false;    // flag to detect if the CPU is currentl
y running
        public string traceLine;        // line that gets output to trace.log
        public const int REGS_MULTIPLIER = 4;
        public bool isFinished;
        public int consolePtr; // points to the most recent position in the cons
ole
        public string console;  // keeps up to date with GUI

        public event EventHandler ExecuteCompleted; // delegate for handling the
 OnRunCompleted event
        public event EventHandler PutChar; // delegate for handling the OnPutCha
r event
        public event EventHandler ReadLine; // delegate for handling the ReadLin
e event

        /// <summary>
        /// Constructor for the CPU class
        /// </summary>
        /// <param name="_ram">Reference to ram variable in the Computer class</
param>
        /// <param name="_registers">Reference to the registers variable in the
Computer class</param>
        public CPU(Memory _ram, Memory _registers)
        {
            ram = _ram;
            registers = _registers;
            consolePtr = 0;
        }

        /// <summary>
        /// Fetches the word at the address
        /// specified by the program counter
        /// </summary>
        /// <returns>A uint resembling the word fetched from memory</returns>
        public uint fetch()
        {
            if (!running) { return 0; }
            uint instrAddr = registers.ReadWord(PC_REG);    // program counter
            traceLine = string.Format("{0:000000} ", MainWindow.traceCount);
            traceLine += string.Format("{0:X8} ", instrAddr);
            uint test = ram.ReadWord(instrAddr);            // instruction at pr
ogram counter value
            return test;
        }

        /// <summary>
        /// Decodes the word that was fetched from memory
        /// </summary>
        /// <param name="instr">The word that needs to be decoded</param>
        /// <returns>An instance of the Instruction class containing information
        ///          for that specific instruction</returns>
        //public Instruction decode(uint instr)
        public Instruction decode(uint instr)
        {
            uint type = Instruction.FindType(instr);
            uint bit4 = Instruction.FindBit4(instr);
            uint bit7 = Instruction.FindBit7(instr);
            uint isBx = Instruction.FindBxEncoding(instr);

            if (isBx == 0x12FFF1)
            {
                BX bx = new BX(instr);
                bx.typeInstr = Instruction.TypeInstr.BX;
                return bx;
            }
            else if ((type == 0b001) ||
                ((type == 0b000) &&
                (!((bit4 == 0b1) &&
                  (bit7 == 0b1)))))          // type = 000/001; data processing
instruction
            {
                DataProcessing dp = new DataProcessing(instr);
                dp.typeInstr = Instruction.TypeInstr.DataProcessing;
                return dp;
            }
            else if ((type == 0b000) &&   // type = 000; multiply instruction
                     (bit4 == 0b1) &&
                     (bit7 == 0b1))
            {
                Multiply mul = new Multiply(instr);
                mul.typeInstr = Instruction.TypeInstr.Multiply;
                return mul;
            }
            else if ((type == 0b010) ||
                     (type == 0b011))     // type = 010/011; load/store instruct
ion
            {
                LoadStore ls = new LoadStore(instr);
                ls.typeInstr = Instruction.TypeInstr.LoadStore;
                return ls;
            }
            else if (type == 0b100)       // type = 100; load/store multiple ins
truction
            {
                LoadStoreMultiple lsm = new LoadStoreMultiple(instr);
                lsm.typeInstr = Instruction.TypeInstr.LoadStoreMultiple;
                return lsm;
            }
            else if (type == 0b111)
            {
                SWI swi = new SWI(instr);
                swi.typeInstr = Instruction.TypeInstr.Swi;
                return swi;
            }
            else if (type == 0b101)
            {
                Branch branch = new Branch(instr);
                branch.typeInstr = Instruction.TypeInstr.Branch;
                branch.pc_b = registers.ReadWord(PC_REG) + 4;
                branch.Disassemble();
                return branch;
            }
            else { return new Instruction(instr); }
        }

        /// <summary>
```

```
        /// Executes the instruction provided
        /// by the Instruction instance
        /// </summary>
        /// <param name="instr"></param>
        public void execute(Instruction instr)
        {
            uint cpsrReg = 16 * 4;

            registers.WriteWord(PC_REG, registers.ReadWord(PC_REG) + 4);
            Execute.Execute.ExecuteInstruction(instr, this);
            if (instr.typeInstr != Instruction.TypeInstr.Branch ||
               ((instr.typeInstr == Instruction.TypeInstr.Branch) &&
                !instr.executed))
            {
                registers.WriteWord(PC_REG, registers.ReadWord(PC_REG) - 4);
            }

            if (instr.typeInstr == Instruction.TypeInstr.Swi)
            {
                switch (instr.offset_swi)
                {
                    case 0x0:
                        OnPutChar(EventArgs.Empty);
                        break;
                    case 0x11:
                        Computer.finished = true;
                        break;
                    case 0x06a:
                        OnReadLine(EventArgs.Empty);
                        break;
                    default:    // all other swi numbers are treated as no-ops
                        break;
                }
            }

            traceLine += string.Format("{0:X8} ", ram.CalculateChecksum());

            if (registers.TestFlag(cpsrReg, 31))
            {
                traceLine += "1";
            }
            else { traceLine += "0"; }
            if (registers.TestFlag(cpsrReg, 30))
            {
                traceLine += "1";
            }
            else { traceLine += "0"; }
            if (registers.TestFlag(cpsrReg, 29))
            {
                traceLine += "1";
            }
            else { traceLine += "0"; }
            if (registers.TestFlag(cpsrReg, 28))
            {
                traceLine += "1 ";
            }
            else { traceLine += "0 "; }

            traceLine += "SYS ";

            for (int i = 0; i < 15; ++i)
            {
                traceLine += i + "=" + string.Format("{0:X8} ", registers.ReadWor
d((uint)(i * REGS_MULTIPLIER)));
            }

            OnExecuteCompleted(EventArgs.Empty);
            traceLine = "";
        }
        /// <summary>
        /// Event handler method
        /// </summary>
        /// <param name="e"></param>
        protected virtual void OnExecuteCompleted(EventArgs e)
        {
            ExecuteCompleted?.Invoke(this, e);
        }

        /// <summary>
        /// Event handler method
        /// </summary>
        /// <param name="e"></param>
        protected virtual void OnPutChar(EventArgs e)
        {
            PutChar?.Invoke(this, e);
        }

        /// <summary>
        /// Event handler method
        /// </summary>
        /// <param name="e"></param>
        protected virtual void OnReadLine(EventArgs e)
        {
            ReadLine?.Invoke(this, e);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace armsim
{
    class DataProcessing : Instruction
    {
        public static uint[] opcodes = {
            0x0, // AND
            0x1, // EOR
            0x2, // SUB
            0x3, // RSB
            0x4, // ADD
            0x5, // ADC
            0x6, // SBC
            0x7, // RSC
            0x8, // TST
            0x9, // TEQ
            0xA, // CMP
            0xB, // CMN
            0xC, // ORR
            0xD, // MOV
            0xE, // BIC
            0xF  // MVN
        };

        public static string[] opcodeNames = {
            "and",
            "eor",
            "sub",
            "rsb",
            "add",
            "adc",
            "sbc",
            "rsc",
            "tst",
            "teq",
            "cmp",
            "cmn",
            "orr",
            "mov",
            "bic",
            "mvn"
        };

        /// <summary>
        /// Represents the information found in data processing instruction
        /// </summary>
        public DataProcessing(uint instr) : base(instr)
        {
            opcode_DP = FindOpcode();
            s_DP = FindS();
            rn_DP = FindRn();
            rd_DP = FindRd();

            if (type == 0b000)      // requires a shift either by register or im
mediate
            {
                shiftCode_DP = FindShiftCode();
                rm_DP = FindRm();

                if (bit4 == 0b0) // requires a shift by immediate
                {
                    typeDP = TypeDP.ShiftByImm;
                    shiftAmount_DP = FindShiftAmount();
                }
                else if (bit4 == 0b1)    // requires a shift by register
                {
                    typeDP = TypeDP.ShiftByReg;
                    rs_DP = FindRs();
                }
                else { }
            }
            else if (type == 0b001) // does not require a shift
            {
                typeDP = TypeDP.Imm;
                rotate_DP = FindRotate();
                imm_DP = FindImm();
            }
            else { }

            Disassemble();
        }

        /// <summary>
        /// Extracts the opcode nibble and
        /// stores in the instance variable opcode
        /// </summary>
        private uint FindOpcode()
        {
            return Memory.ShiftToEnd(instr, 21, 24);
        }

        /// <summary>
        /// Extracts the S bit and
        /// stores in the instance variable s
        /// </summary>
        private uint FindS()
        {
            return Memory.ShiftToEnd(instr, 20, 20);
        }

        /// <summary>
        /// Extracts the Rn nibble and
        /// stores in the instance variable rn
        /// </summary>
        private uint FindRn()
        {
            return Memory.ShiftToEnd(instr, 16, 19);
        }

        /// <summary>
        /// Extracts the Rd nibble and
        /// stores in the instance variable rd
        /// </summary>
        private uint FindRd()
        {
            return Memory.ShiftToEnd(instr, 12, 15);
        }

        /// <summary>
        /// Extracts the rotate nibble and
        /// stores in the instance variable rotate
        /// </summary>
        private uint FindRotate()
        {
            return Memory.ShiftToEnd(instr, 8, 11);
        }

        /// <summary>
        /// Extracts the rotate nibble and
        /// stores in the instance variable rotate
        /// </summary>
        private uint FindImm()
        {
            return Memory.ShiftToEnd(instr, 0, 7);
        }
```

```

        /// <summary>
        /// Extracts the 5 bit shift number and
        /// stores in the instance variable shiftAmount
        /// </summary>
        private uint FindShiftAmount()
        {
            return Memory.ShiftToEnd(instr, 7, 11);
        }

        /// <summary>
        /// Extracts the Rm nibble and
        /// stores in the instance variable rm
        /// </summary>
        private uint FindRm()
        {
            return Memory.ShiftToEnd(instr, 0, 3);
        }

        /// <summary>
        /// Extracts the 2 bit Sh code and
        /// stores in the instance variable shiftCode
        /// </summary>
        private uint FindShiftCode()
        {
            return Memory.ShiftToEnd(instr, 5, 6);
        }

        /// <summary>
        /// Extracts the Rs nibble and
        /// stores in the instance variable rs
        /// </summary>
        private uint FindRs()
        {
            return Memory.ShiftToEnd(instr, 8, 11);
        }

        /// <summary>
        /// Finds the correct opcode string
        /// for the textual representation
        /// </summary>
        public string FindOpcodeString()
        {
            string name = "";
            for (int i = 0; i < opcodes.Length; ++i)
            {
                if (opcode_DP == i)
                {
                    name = opcodeNames[i];
                }
            }

            return name;
        }

        /// <summary>
        /// Finds the correct immediate string after rotation
        /// for the textual representation
        /// </summary>
        public string FindRotatedImm()
        {
            return Convert.ToString((imm_DP >> ((int)rotate_DP) * 2) | (imm_DP <
< ((32 - (int)rotate_DP) * 2)));
        }

        /// <summary>
        /// Produce the textual representation
        /// </summary>
        public void Disassemble()
        {
            if (type == 0b000)      // requires a shift either by register or im
mediate
            {
                if (bit4 == 0b0) // requires a shift by immediate
                {
                    if (opcode_DP == 0b1010)    // CMP
                    {
                        text = FindOpcodeString() + FindCondString() + "r" + rn_
DP + ",r" + rm_DP + "," + FindShiftString() + "#" + shiftAmount_DP;
                    }
                    else
                    {
                        text = FindOpcodeString() + FindCondString() + "r" + rd_
DP + ",r" + rm_DP + "," + FindShiftString() + "#" + shiftAmount_DP;
                    }
                }
                else if (bit4 == 0b1)    // requires a shift by register
                {
                    if (opcode_DP == 0b1010)    // CMP
                    {
                        text = FindOpcodeString() + FindCondString() + "r" + rn_
DP + ",r" + rm_DP + "," + FindShiftString() + "r" + rs_DP;
                    }
                    else
                    {
                        text = FindOpcodeString() + FindCondString() + "r" + rd_
DP + ",r" + rm_DP + "," + FindShiftString() + "r" + rs_DP;
                    }
                }
                else { }
            }
            else if (type == 0b001) // does not require a shift
            {
                if ((opcode_DP == 0b1101) ||    // MOV or MVN that only have one
 register
                    (opcode_DP == 0b1111))
                {
                    text = FindOpcodeString() + FindCondString() + "r" + rd_DP +
 ",#" + FindRotatedImm();
                }
                else if (opcode_DP == 0b1010)
                {
                    text = FindOpcodeString() + FindCondString() + "r" + rn_DP +
 ",#" + FindRotatedImm();
                }
                else
                {
                    text = FindOpcodeString() + FindCondString() + "r" + rd_DP +
 ",r" + rn_DP + ",#"+ FindRotatedImm();
                }
            }
            else { }
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace armsim.Execute
{
    class ExecBranch : Execute
    {
        public static void ExecuteBranch()
        {
            if (instr.l_b == 0)
            {
                B b = new B();
            }
            else
            {
                BL bl = new BL();
            }
        }

        //public static uint FindTargetAddr()
        //{
        //    int isNeg = instr.offset_b & 800000;
        //    if (isNeg != 0)
        //    {
        //        instr.offset_b |= 0x3f000000;
        //    }
        //    int left_shifted = instr.offset_b << 2;
        //    int pc_read = (int)cpu.registers.ReadWord(15 * 4);
        //    return (uint)(left_shifted + pc_read);
        //}
    }

    class B : ExecBranch
    {
        public B()
        {
            cpu.registers.WriteWord(15 * 4, (uint)instr.target_b);
        }
    }

    class BL : ExecBranch
    {
        public BL()
        {
            cpu.registers.WriteWord(14 * 4, cpu.registers.ReadWord(15 * 4) - 4);
            cpu.registers.WriteWord(15 * 4, (uint)instr.target_b);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace armsim.Execute
{
    class ExecBX : Execute
    {
        public static void ExecuteBX()
        {
            if (instr.bit0_bx == 1)
            {
                cpu.registers.SetFlag(16 * 4, 5, true);
            }
            else
            {
                cpu.registers.SetFlag(16 * 4, 5, false);
            }

            cpu.registers.WriteWord(15 * REGS_MULTIPLIER, (cpu.registers.ReadWor
d(instr.rm_bx * REGS_MULTIPLIER) & 0xFFFFFFFE) + 4);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace armsim.Execute
{
    class ExecDataProcessing : Execute
    {
        public static void ExecuteDataProcessing()
        {
            switch (instr.opcode_DP)
            {
                case 0b1101:    // MOV
                    MOV mov = new MOV();
                    break;
                case 0b1111:    // MVN
                    MVN mvn = new MVN();
                    break;
                case 0b0100:    // ADD
                    ADD add = new ADD();
                    break;
                case 0b0010:    // SUB
                    SUB sub = new SUB();
                    break;
                case 0b0011:    // RSB
                    RSB rsb = new RSB();
                    break;
                case 0b0000:    // AND
                    AND and = new AND();
                    break;
                case 0b1100:    // ORR
                    ORR orr = new ORR();
                    break;
                case 0b0001:    // EOR
                    EOR eor = new EOR();
                    break;
                case 0b1110:    // BIC
                    BIC bic = new BIC();
                    break;
                case 0b1010:    // CMP
                    CMP cmp = new CMP();
                    break;
                default:
                    break;
            }
        }

        public static uint FindOperand2()
        {
            if (instr.typeDP == Instruction.TypeDP.Imm)
            {
                // rotate right
                return (instr.imm_DP >> ((int)instr.rotate_DP) * 2) | (instr.imm
_DP << ((32 - (int)instr.rotate_DP) * 2));
            }
            else if (instr.typeDP == Instruction.TypeDP.ShiftByImm)
            {
                switch (instr.shiftCode_DP)
                {
                    case 0b00:    // logical shift left
                        return cpu.registers.ReadWord(instr.rm_DP * REGS_MULTIPL
IER) << (int)instr.shiftAmount_DP;
                    case 0b01:    // logical shift right
                        return cpu.registers.ReadWord(instr.rm_DP * REGS_MULTIPL
IER) >> (int)instr.shiftAmount_DP;
                    case 0b10:    // arithmetic shift right
                        return (uint)(unchecked((int)cpu.registers.ReadWord(inst
r.rm_DP * REGS_MULTIPLIER)) >> (int)instr.shiftAmount_DP);
                    case 0b11:    // rotate right
                        return (cpu.registers.ReadWord(instr.rm_DP * REGS_MULTIP
LIER) >> (int)instr.shiftAmount_DP) | (cpu.registers.ReadWord(instr.rm_DP * REGS
_MULTIPLIER) << (32 - (int)instr.shiftAmount_DP));
                    default:
                        return 0;
                }
            }
            else if (instr.typeDP == Instruction.TypeDP.ShiftByReg)
            {
                switch (instr.shiftCode_DP)
                {
                    case 0b00:    // logical shift left
                        return cpu.registers.ReadWord(instr.rm_DP * REGS_MULTIPL
IER) << (int)cpu.registers.ReadWord(instr.rs_DP * REGS_MULTIPLIER);
                    case 0b01:    // logical shift right
                        return cpu.registers.ReadWord(instr.rm_DP * REGS_MULTIPL
IER) >> (int)cpu.registers.ReadWord(instr.rs_DP * REGS_MULTIPLIER);
                    case 0b10:    // arithmetic shift right
                        return (uint)(unchecked((int)cpu.registers.ReadWord(inst
r.rm_DP * REGS_MULTIPLIER)) >> (int)cpu.registers.ReadWord(instr.rs_DP * REGS_MU
LTIPLIER));
                    case 0b11:    // rotate right
                        return (cpu.registers.ReadWord(instr.rm_DP * REGS_MULTIP
LIER) >> (int)cpu.registers.ReadWord(instr.rs_DP * REGS_MULTIPLIER)) | (cpu.regi
sters.ReadWord(instr.rm_DP * REGS_MULTIPLIER) << (32 - (int)cpu.registers.ReadWo
rd(instr.rs_DP * REGS_MULTIPLIER)));
                    default:
                        return 0;
                }
            }
            else { return cpu.registers.ReadWord(instr.rm_DP * REGS_MULTIPLIER);
 }
        }
    }

    class AND : ExecDataProcessing
    {
        public AND()
        {
            uint first = cpu.registers.ReadWord(instr.rn_DP * REGS_MULTIPLIER);
            uint second = FindOperand2();
            cpu.registers.WriteWord(instr.rd_DP * REGS_MULTIPLIER, first & secon
d);
        }
    }

    class ADD : ExecDataProcessing
    {
        public ADD()
        {
            uint first = cpu.registers.ReadWord(instr.rn_DP * REGS_MULTIPLIER);
            uint second = FindOperand2();
            cpu.registers.WriteWord(instr.rd_DP * REGS_MULTIPLIER, first + secon
d);
        }
    }

    class BIC : ExecDataProcessing
    {
        public BIC()
        {
            uint first = cpu.registers.ReadWord(instr.rn_DP * REGS_MULTIPLIER);
            uint second = FindOperand2();
            cpu.registers.WriteWord(instr.rd_DP * REGS_MULTIPLIER, first & ~seco
nd);
        }
    }

    class CMP : ExecDataProcessing
    {
        const uint CPSR = 16 * 4;
```

```
        public CMP()
        {
            uint eval = cpu.registers.ReadWord(instr.rn_DP * REGS_MULTIPLIER) -
FindOperand2();
            uint eval_bit31 = eval >> 31;

            // set N flag
            if (eval_bit31 == 1)
            {
                cpu.registers.SetFlag(CPSR, 31, true);
            }
            else
            {
                cpu.registers.SetFlag(CPSR, 31, false);
            }

            // set Z flag
            if (eval == 0)
            {
                cpu.registers.SetFlag(CPSR, 30, true);
            }
            else
            {
                cpu.registers.SetFlag(CPSR, 30, false);
            }

            // set C flag
            if (FindOperand2() <= cpu.registers.ReadWord(instr.rn_DP * REGS_MULT
IPLIER))
            {
                cpu.registers.SetFlag(CPSR, 29, true);
            }
            else
            {
                cpu.registers.SetFlag(CPSR, 29, false);
            }

            // set V flag
            if ((((cpu.registers.ReadWord(instr.rn_DP * REGS_MULTIPLIER) >> 31)
^ (FindOperand2() >> 31)) == 1) &&
                 (((cpu.registers.ReadWord(instr.rn_DP * REGS_MULTIPLIER) >> 31)
 ^ eval_bit31) == 1))
            {
                cpu.registers.SetFlag(CPSR, 28, true);
            }
            else
            {
                cpu.registers.SetFlag(CPSR, 28, false);
            }
        }
    }

    class EOR : ExecDataProcessing
    {
        public EOR()
        {
            uint first = cpu.registers.ReadWord(instr.rn_DP * REGS_MULTIPLIER);
            uint second = FindOperand2();
            cpu.registers.WriteWord(instr.rd_DP * REGS_MULTIPLIER, first ^ secon
d);
        }
    }

    class MOV : ExecDataProcessing
    {
        public MOV()
        {
            cpu.registers.WriteWord(instr.rd_DP * REGS_MULTIPLIER, FindOperand2(
));
```

```
        }
    }

    class MVN : ExecDataProcessing
    {
        public MVN()
        {
            cpu.registers.WriteWord(instr.rd_DP * REGS_MULTIPLIER, ~FindOperand2
());
        }
    }

    class ORR : ExecDataProcessing
    {
        public ORR()
        {
            uint first = cpu.registers.ReadWord(instr.rn_DP * REGS_MULTIPLIER);
            uint second = FindOperand2();
            cpu.registers.WriteWord(instr.rd_DP * REGS_MULTIPLIER, first | secon
d);
        }
    }

    class RSB : ExecDataProcessing
    {
        public RSB()
        {
            uint first = cpu.registers.ReadWord(instr.rn_DP * REGS_MULTIPLIER);
            uint second = FindOperand2();
            cpu.registers.WriteWord(instr.rd_DP * REGS_MULTIPLIER, second - firs
t);
        }
    }

    class SUB : ExecDataProcessing
    {
        public SUB()
        {
            uint second = cpu.registers.ReadWord(instr.rn_DP * REGS_MULTIPLIER);
            uint first = FindOperand2();
            cpu.registers.WriteWord(instr.rd_DP * REGS_MULTIPLIER, second - firs
t);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace armsim.Execute
{
    class ExecLoadStore : Execute
    {
        public static void ExecuteLoadStore()
        {
            if (instr.l_LS == 1)
            {
                LDR ldr = new LDR();
            }
            else
            {
                STR str = new STR();
            }
        }

        public static uint FindEA()
        {
            uint ea;
            if (instr.u_LS == 1)
            {
                ea = FindPosEA();
            }
            else
            {
                ea = FindNegEA();
            }
            return ea;
        }

        public static uint FindPosEA()
        {
            uint ea = 0;
            uint baseReg = cpu.registers.ReadWord(instr.rn_LS * REGS_MULTIPLIER)
;

            if (instr.type == 0b010)
            {
                ea = baseReg + instr.imm_LS;
            }
            else
            {
                if ((instr.shiftAmount_LS == 0) &&    // register offset
                    (instr.shiftCode_LS == 0))
                {
                    ea = baseReg + cpu.registers.ReadWord(instr.rm_LS * REGS_MUL
TIPLIER);
                }
                else    // scaled register offset
                {
                    switch (instr.shiftCode_LS)
                    {
                        case 0b00:    // logical shift left
                            ea = baseReg + (cpu.registers.ReadWord(instr.rm_LS *
 REGS_MULTIPLIER) << (int)instr.shiftAmount_LS);
                            break;
                        case 0b01:    // logical shift right
                            ea = baseReg + (cpu.registers.ReadWord(instr.rm_LS *
 REGS_MULTIPLIER) >> (int)instr.shiftAmount_LS);
                            break;
                        case 0b10:    // arithmetic shift right
                            ea = baseReg + ((uint)(unchecked((int)cpu.registers.
ReadWord(instr.rm_LS * REGS_MULTIPLIER)) >> (int)instr.shiftAmount_LS));
                            break;
                        case 0b11:    // rotate right
                            ea = baseReg + ((instr.imm_LS >> (int)instr.shiftAmo
unt_LS) | (cpu.registers.ReadWord(instr.rm_LS * REGS_MULTIPLIER) << (32 - (int)i
nstr.shiftAmount_LS)));
                            break;
                        default:
                            return 0;
                    }
                }
            }
            return ea;
        }

        public static uint FindNegEA()
        {
            uint ea = 0;
            uint baseReg = cpu.registers.ReadWord(instr.rn_LS * REGS_MULTIPLIER)
;

            if (instr.type == 0b010)
            {
                ea = baseReg - instr.imm_LS;
            }
            else
            {
                if ((instr.shiftAmount_LS == 0) &&    // register offset
                    (instr.shiftCode_LS == 0))
                {
                    ea = baseReg - cpu.registers.ReadWord(instr.rm_LS * REGS_MUL
TIPLIER);
                }
                else    // scaled register offset
                {
                    switch (instr.shiftCode_LS)
                    {
                        case 0b00:    // logical shift left
                            ea = baseReg - (cpu.registers.ReadWord(instr.rm_LS *
 REGS_MULTIPLIER) << (int)instr.shiftAmount_LS);
                            break;
                        case 0b01:    // logical shift right
                            ea = baseReg - (cpu.registers.ReadWord(instr.rm_LS *
 REGS_MULTIPLIER) >> (int)instr.shiftAmount_LS);
                            break;
                        case 0b10:    // arithmetic shift right
                            ea = baseReg - ((uint)(unchecked((int)cpu.registers.
ReadWord(instr.rm_LS * REGS_MULTIPLIER)) >> (int)instr.shiftAmount_LS));
                            break;
                        case 0b11:    // rotate right
                            ea = baseReg - ((instr.imm_LS >> ((int)instr.shiftAm
ount_LS) * 2) | (cpu.registers.ReadWord(instr.rm_LS * REGS_MULTIPLIER) << ((32 -
 (int)instr.shiftAmount_LS) * 2)));
                            break;
                        default:
                            return 0;
                    }
                }
            }
            return ea;
        }
    }

    class LDR : ExecLoadStore
    {
        public LDR()
        {
            uint ea = FindEA();

            if (instr.b_LS == 1)
            {
                cpu.registers.WriteWord(instr.rd_LS * REGS_MULTIPLIER, cpu.ram.R
eadByte(ea));
            }
```

```
            else
            {
                cpu.registers.WriteWord(instr.rd_LS * REGS_MULTIPLIER, cpu.ram.R
eadWord(ea));
            }

            if (instr.w_LS == 1)
            {
                cpu.registers.WriteWord(instr.rn_LS * REGS_MULTIPLIER, ea);
            }
        }
    }

    class STR : ExecLoadStore
    {
        public STR()
        {
            uint ea = FindEA();

            if (instr.b_LS == 1)
            {
                cpu.ram.WriteByte(ea, cpu.registers.ReadByte(instr.rd_LS * REGS_
MULTIPLIER));
            }
            else
            {
                cpu.ram.WriteWord(ea, cpu.registers.ReadWord(instr.rd_LS * REGS_
MULTIPLIER));
            }

            if (instr.w_LS == 1)
            {
                cpu.registers.WriteWord(instr.rn_LS * REGS_MULTIPLIER, ea);
            }
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace armsim.Execute
{
    class ExecLoadStoreMultiple : Execute
    {
        public static void ExecuteLoadStoreMultiple()
        {
            if (instr.l_LSM == 1)
            {
                LDM ldm = new LDM();
            }
            else
            {
                STM stm = new STM();
            }
        }

        public static uint FindEA()
        {
            return 0;
        }
    }

    class LDM : ExecLoadStoreMultiple
    {
        public LDM()
        {
            uint ea = 0;

            // full descending (increment after)
            if ((instr.l_LSM == 1) &&
                (instr.p_LSM == 0) &&
                (instr.u_LSM == 1))
            {
                ea = cpu.registers.ReadWord(instr.rn_LSM * REGS_MULTIPLIER);

                foreach (int reg in instr.regsList_LSM)
                {
                    cpu.registers.WriteWord((uint)reg * REGS_MULTIPLIER, cpu.ram
.ReadWord(ea));
                    ea += 4;
                }
            }


            if (instr.w_LSM == 1)
            {
                cpu.registers.WriteWord(instr.rn_LSM * REGS_MULTIPLIER, cpu.regi
sters.ReadWord(instr.rn_LSM * REGS_MULTIPLIER) + ((uint)instr.regsList_LSM.Count
 * 4));
            }
        }

    }

    class STM : ExecLoadStoreMultiple
    {
        public STM()
        {
            uint ea = 0;
            // full descending (decrement before)
            if ((instr.l_LSM == 0) &&
                (instr.p_LSM == 1) &&
                (instr.u_LSM == 0))
            {
                ea = cpu.registers.ReadWord(instr.rn_LSM * REGS_MULTIPLIER) - (u
int)(instr.regsList_LSM.Count * 4);
```

```
                foreach (int reg in instr.regsList_LSM)
                {
                    cpu.ram.WriteWord(ea, cpu.registers.ReadWord((uint)reg * REG
S_MULTIPLIER));
                    ea += 4;
                }
            }


            if (instr.w_LSM == 1)
            {
                cpu.registers.WriteWord(instr.rn_LSM * REGS_MULTIPLIER, cpu.regi
sters.ReadWord(instr.rn_LSM * REGS_MULTIPLIER) - ((uint)instr.regsList_LSM.Count
 * 4));
            }
        }

    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace armsim.Execute
{
    class ExecMultiply : Execute
    {
        public static void ExecuteMultiply()
        {
            MUL mul = new MUL();
        }
    }

    class MUL : ExecDataProcessing
    {
        public MUL()
        {
            uint first = cpu.registers.ReadWord(instr.rs_M * REGS_MULTIPLIER);
            uint second = cpu.registers.ReadWord(instr.rm_M * REGS_MULTIPLIER);
            cpu.registers.WriteWord(instr.rd_M * REGS_MULTIPLIER, first * second
);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace armsim.Execute
{
    class ExecSwi : Execute
    {
        public event EventHandler PutChar; // delegate for handling the OnPutCha
r event

        public static void ExecuteSwi()
        {
            switch(instr.offset_swi)
            {
                case 0x0:
                    break;
                case 0x11:
                    Computer.finished = true;
                    break;
                case 0x06a:
                    break;
                default:    // all other swi numbers are treated as no-ops
                    break;
            }
        }

        public void Put()
        {
            OnPutChar(EventArgs.Empty);
        }

        /// <summary>
        /// Event handler method
        /// </summary>
        /// <param name="e"></param>
        protected virtual void OnPutChar(EventArgs e)
        {
            PutChar?.Invoke(this, e);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace armsim.Execute
{
    class Execute
    {
        protected static Instruction instr;
        protected static CPU cpu;
        public const int REGS_MULTIPLIER = 4;
        public static bool N;
        public static bool Z;
        public static bool C;
        public static bool V;

        public static void ExecuteInstruction(Instruction _instr, CPU _cpu)
        {
            instr = _instr;
            cpu = _cpu;

            N = cpu.registers.TestFlag(16 * 4, 31);
            Z = cpu.registers.TestFlag(16 * 4, 30);
            C = cpu.registers.TestFlag(16 * 4, 29);
            V = cpu.registers.TestFlag(16 * 4, 28);

            if (FindIfContinue())
            {
                instr.executed = true;
                switch (instr.typeInstr)
                {
                    case Instruction.TypeInstr.DataProcessing:
                        ExecDataProcessing.ExecuteDataProcessing();
                        break;
                    case Instruction.TypeInstr.LoadStore:
                        ExecLoadStore.ExecuteLoadStore();
                        break;
                    case Instruction.TypeInstr.LoadStoreMultiple:
                        ExecLoadStoreMultiple.ExecuteLoadStoreMultiple();
                        break;
                    case Instruction.TypeInstr.Multiply:
                        ExecMultiply.ExecuteMultiply();
                        break;
                    case Instruction.TypeInstr.Branch:
                        ExecBranch.ExecuteBranch();
                        break;
                    case Instruction.TypeInstr.BX:
                        ExecBX.ExecuteBX();
                        break;
                    default:
                        break;
                }
            }
        }

        public static bool FindIfContinue()
        {
            switch (instr.conditionFlags)
            {
                case 0b0000:  // Equal
                    return Z;
                case 0b0001:  // Not equal
                    return !Z;
                case 0b0010:  // Carry set/unsigned higher or same
                    return C;
                case 0b0011:  // Carry clear/unsigned lower
                    return !C;
                case 0b0100:  // Minus/negative
                    return N;
                case 0b0101:  // Plus/positive or zero
```

```
                    return !N;
                case 0b0110:  // Overflow
                    return V;
                case 0b0111:  // No overflow
                    return !V;
                case 0b1000:  // Unsigned higher
                    return C && !Z;
                case 0b1001:  // Unsigned lower or same
                    return !C || Z;
                case 0b1010:  // Signed greater than or equal
                    return N == V;
                case 0b1011:  // Signed less than
                    return N != V;
                case 0b1100:  // Signed greater than
                    return !Z && (N == V);
                case 0b1101:  // Signed less than or equal
                    return Z || (N != V);
                case 0b1110:  // always
                    return true;
                default:
                    instr.executed = false;
                    return false;
            }
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace armsim
{
    public class Instruction
    {
        protected static uint[] conds = {
            0x0, // Equal
            0x1, // Not equal
            0x2, // Carry set/unsigned higher or same
            0x3, // Carry clear/unsigned lower
            0x4, // Minus/negative
            0x5, // Plus/positive or zero
            0x6, // Overflow
            0x7, // No overflow
            0x8, // Unigned higher
            0x9, // Unsigned lower or same
            0xA, // Signed greater than or equal
            0xB, // Signed less than
            0xC, // Signed greater than
            0xD, // Signed less than or equal
            0xE  // Always (unconditional)
        };

        protected static string[] condNames = {
            "eq",
            "ne",
            "cs/hs",
            "cs/lo",
            "mi",
            "pl",
            "vs",
            "vc",
            "hi",
            "ls",
            "ge",
            "lt",
            "gt",
            "le",
            ""
        };

        public static uint[] shifts = {
            0b00,   // LSL
            0b01,   // LSR
            0b10,   // ASR
            0b11    // ROR
        };

        public static string[] shiftNames = {
            "lsl",
            "lsr",
            "asr",
            "ror"
        };

        public enum TypeInstr
        {
            DataProcessing,
            LoadStore,
            LoadStoreMultiple,
            Multiply,
            Swi,
            Branch,
            BX
        }
```

```
        public enum TypeDP
        {
            Imm,
            ShiftByImm,
            ShiftByReg
        }

        public enum TypeLS
        {
            Imm,
            Register
        }

        public uint instr;          // holds the instruction for this class and 
all derived classes
        public string text;         // textual ARM representation

        public TypeInstr typeInstr;
        public TypeDP typeDP;
        public TypeLS typeLS;

        /* ---------------------------------------------------------------------
----------------------------------------*/
        // ALL INSTRUCTIONS

        // bits pertaining to all instructions
        public string instrString;  // contains the disassembled instruction str
ing
        public uint conditionFlags; // bits 28-31; code for the condition flags
        public uint type;           // bits 25-27; determine types of operation 
- dataprocessing, load/store, or branch
        public uint bit4;           // bit 4; if data processing, set to 1, and 
bit7 set to 1 - then multiply
        public uint bit7;           // bit 7; if data processing, set to 1, and 
bit4 set to 1 - then multiply
        public bool executed;       // if the conditional allowed the instructio
n to execute

        /* ---------------------------------------------------------------------
----------------------------------------*/
        // DATA PROCESSING

        // bits pertaining to all data processing instructions
        public uint opcode_DP; // bits 21-24; determines the specific instructio
n
        public uint s_DP;      // bit 20; determines whether or not to write con
dition codes
        public uint rn_DP;     // bits 16-19; base register
        public uint rd_DP;     // bits 12-15; destination register

        // bits pertaining to immediate data processing instructions
        public uint rotate_DP;      // bits 8-11; immediate value is rotated by 
(2 * rot)
        public uint imm_DP;         // bits 0-7; the immediate value

        // bits pertaining to all shifted data processing instructions
        public uint shiftCode_DP;   // bits 5-6; code determines the shift opera
tion
        public uint rm_DP;          // bits 0-3; the register in operand2 whose 
value can be shifted

        // bits pertaining to immediate shifted data processing instructions
        public uint shiftAmount_DP;        // bits 7-11; imm value to shift by

        // bits pertaining to register shifted data processing instructions
        public uint rs_DP;              // bits 8-11; the register in operand2 w
hose value is the shift amount
```

```
        /* ---------------------------------------------------------------------
----------------------------------------*/
        // LOAD/STORE

        // bits pertaining to all load/store commands
        public uint p_LS;  // bit 24; 0=post-indexed, 1=pre-indexed
        public uint u_LS;  // bit 23; 1=positive offset, 0=negative offset
        public uint b_LS;  // bit 22; 1=unsigned byte, 0=word
        public uint w_LS;  // bit 21; when p=1, 0=no writeback, 1=writeback
        public uint l_LS;  // bit 20; 1=load, 0=store
        public uint rn_LS; // bits 16-19; the base register
        public uint rd_LS; // bits 12-15; the destination register

        // bits pertaining to immediate load/store commands
        public uint imm_LS;         // bits 0-11; the immediate value offset

        // bits pertaining to shifted load/store commands
        public uint shiftAmount_LS;    // bits 7-11; the immediate value if shif
t is by immediate
        public uint shiftCode_LS;      // bits 5-6; code that determines the shi
ft operation
        public uint rm_LS;             // bits 0-3; the register whose value is
being shifted


        /* ---------------------------------------------------------------------
----------------------------------------*/
        // LOAD/STORE MULTIPLE

        public List<int> regsList_LSM;  // bits 0-15; whatever number bits are 1
, that register is loaded/stored
        public uint p_LSM;  // bit 24;
        public uint u_LSM;  // bit 23; 1=positive offset, 0=negative offset
        public uint s_LSM;  // bit 22; 1=unsigned byte, 0=word
        public uint w_LSM;  // bit 21; when p=1, 0=no writeback, 1=writeback
        public uint l_LSM;  // bit 20; 1=load, 0=store
        public uint rn_LSM; // bits 16-19; the base register


        /* ---------------------------------------------------------------------
----------------------------------------*/
        // MULTIPLY

        public uint s_M;           // bit 20; determines whether or not to write
 condition codes
        public uint rd_M;          // bits 16-19; destination register
        public uint rs_M;          // bits 8-11; value to be multiplied with the
 value in rm
        public uint rm_M;          // bits 0-3; first value to be multiplied

        /* ---------------------------------------------------------------------
----------------------------------------*/
        // SWI

        public uint offset_swi;     // bits 0-23; the offset in a swi instructio
n

        /* ---------------------------------------------------------------------
----------------------------------------*/
        // Branch

        public uint l_b;            // bit 24; whether or not to update link reg
ister (r14)
        public int offset_b;        // bits 0-23; the offset in a branch instruc
tion
        public uint pc_b;           // value of pc for branch
        public int target_b;       // value of target address for branch

        /* ---------------------------------------------------------------------
----------------------------------------*/
```

```
        // BX

        public uint rm_bx;          // bits 0-3; rm register
        public uint encoding_bx;    // bits 4-27; the encoding specific to bx
        public uint bit0_bx;        // bit 0; for determining thumb mode


        /// <summary>
        /// Creates a narrower instance of Instruction by
        /// using the type instance variable
        /// </summary>
        public Instruction(uint _instr)
        {
            instr = _instr;
            conditionFlags = FindConditionFlags();
            type = FindType(instr);
            bit4 = FindBit4(instr);
            bit7 = FindBit7(instr);
        }

        /// <summary>
        /// Extracts the cond nybble and
        /// stores in the instance variable conditionFlags
        /// </summary>
        private uint FindConditionFlags()
        {
            return Memory.ShiftToEnd(instr, 28, 31);
        }

        /// <summary>
        /// Extracts the 3 bit type from the encoding and
        /// stores in the instance variable type
        /// </summary>
        public static uint FindType(uint instr)
        {
            return Memory.ShiftToEnd(instr, 25, 27);
        }

        /// <summary>
        /// Extracts the 7th bit of the encoding and
        /// stores in the instance variable bit7
        /// </summary>
        public static uint FindBit7(uint instr)
        {
            return Memory.ShiftToEnd(instr, 7, 7);
        }

        /// <summary>
        /// Extracts the 4th bit of the encoding and
        /// stores in the instance variable bit4
        /// </summary>
        public static uint FindBit4(uint instr)
        {
            return Memory.ShiftToEnd(instr, 4, 4);
        }

        /// <summary>
        /// Extracts bits 4 - 27 that are
        /// specific to the bx instruction
        /// </summary>
        public static uint FindBxEncoding(uint instr)
        {
            return Memory.ShiftToEnd(instr, 4, 27);
        }

        /// <summary>
        /// Finds the correct conditional string
        /// ending for the textual representation
        /// </summary>
        public string FindCondString()
        {
            string name = "";
            for (int i = 0; i < conds.Length; ++i)
            {
                if (conditionFlags == i)
                {
                    name = condNames[i];
                }
            }

            return name;
        }

        /// <summary>
        /// Finds the correct shift string
        /// for the textual representation
        /// </summary>
        public string FindShiftString()
        {
            string name = "";

            if ((type == 0b000) ||
                (type == 0b001))
            {
                for (int i = 0; i < shifts.Length; ++i)
                {
                    if (shiftCode_DP == i)
                    {
                        name = shiftNames[i];
                    }
                }
            }
            else
            {
                for (int i = 0; i < shifts.Length; ++i)
                {
                    if (shiftCode_LS == i)
                    {
                        name = shiftNames[i];
                    }
                }
            }

            return name;
        }
    }
}
```

```
// Loader.cs
// Contains methods to implement loading
// of ELF executables into virtual memory.

using System;
using System.Collections.Generic;
using System.Data.Common;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;
using System.Threading.Tasks;
using System.Diagnostics;
using System.IO.Packaging;
using log4net;
using log4net.Config;

namespace armsim
{
    // Simulates loading contents of ELF into virtual memory
    public class Loader
    {
        // To log, set the loggingEnabled variable to true
        static bool loggingEnabled = true;

        private static int retCode = 0;      // the code to return with default o
f 0
        private static Memory memory;        // an instance of the Memory class
        private static Computer sim;         // the Computer instance that calls
this class
        private static int memSize;          // size in bytes of virtual memory
        private static string fileName;      // name of ELF file to load
        private ELF elfHeader;               // struct to hold the information in
 the ELF header

        private static readonly ILog log = LogManager.GetLogger(typeof(Loader));

        /// <summary>
        /// Constructor for the Loader class
        /// </summary>
        /// <param name="_memSize">size in bytes of virtual memory</param>
        /// <param name="_fileName">name of ELF file to load</param>
        /// <param name="_memory">Reference to ram variable in the Computer clas
s</param>
        public Loader(int _memSize, string _fileName, Memory _memory, Computer _
sim)
        {
            memSize = _memSize;
            fileName = _fileName;
            memory = _memory;
            sim = _sim;
            Continue();
        }

        /// <summary>
        /// Struct to hold data read from ELF header
        /// From Dr. Schaub
        /// </summary>
    [StructLayout(LayoutKind.Sequential, Pack = 1)]
        public struct ELF
        {
            public byte EI_MAG0, EI_MAG1, EI_MAG2, EI_MAG3, EI_CLASS, EI_DATA, E
I_VERSION;
            byte unused1, unused2, unused3, unused4, unused5, unused6, unused7,
unused8, unused9;
            public ushort e_type;
            public ushort e_machine;
            public uint e_version;
            public uint e_entry;
            public uint e_phoff;
            public uint e_shoff;
            public uint e_flags;
            public ushort e_ehsize;
            public ushort e_phentsize;
            public ushort e_phnum;
            public ushort e_shentsize;
            public ushort e_shnum;
            public ushort e_shstrndx;
        }

        /// <summary>
        /// Struct to hold data read from a progMemory header
        /// </summary>
        public struct Program
        {
            public uint p_type;
            public uint p_offset;
            public uint p_vaddr;
            public uint p_paddr;
            public uint p_filesz;
            public uint p_memsz;
            public uint p_flags;
            public uint p_align;
        }

        /// <summary>
        /// Struct to hold data read from a section header
        /// </summary>
        public struct Section
        {
            public uint s_name;
            public uint s_type;
            public uint s_addr;
            public uint s_off;
            public uint s_size;
            public uint s_es;
            public uint s_flg;
            public uint s_lk;
            public uint s_inf;
            public uint s_al;
        }

        /// <summary>
        /// Getter method to retrieve program entry address from
        /// the ELF Header - the initial PC value
        /// </summary>
        /// <returns></returns>
        public uint GetElfEntry()
        {
            return elfHeader.e_entry;
        }

        /// <summary>
        /// Getter method to retrieve the retCode instance variable
        /// </summary>
        /// <returns></returns>
        public int GetRetCode()
        {
            return retCode;
        }

        /// <summary>
        /// Loads contents of ELF executable into virtual memory
        /// </summary>
        /// <param name="stream">The FileStream object of the executable created
 in Continue()</param>
        /// <param name="elfHeader">ELF struct instance</param>
        public void LoadFile(FileStream stream, ELF elfHeader)
        {
            // Read program header;
```

```
            // From Dr. Schaub
            stream.Seek(elfHeader.e_phoff, SeekOrigin.Begin);

            Program programHeader;  // struct to hold individual program header
info
            byte[] headerData;
            byte[] data;
            int numBytesRead = 0;   // for record of offset in elfHeader

            XmlConfigurator.Configure();
            if (loggingEnabled == true)
            {
                log.Info("Simulator.LoadFile: Number of segments: " + elfHeader.e_phnum);
                log.Info("Simulator.LoadFile: Program header offset: " + elfHeader.e_phoff);
                log.Info("Simulator.LoadFile: Size of program headers: " + elfHeader.e_phents
ize);
            }

            // Iterate over program headers
            for (int i = 0; i < elfHeader.e_phnum; ++i)
            {
                headerData = new byte[elfHeader.e_phentsize];   // create array
of size specified in elfHeader
                numBytesRead += stream.Read(headerData, 0, (int)elfHeader.e_phen
tsize);
                programHeader = ByteArrayToStructure<Program>(headerData);

                if (loggingEnabled == true)
                {
                    log.Info("Simulator.LoadFile: Segment " + (i + 1) +
                                " - Address = " + (elfHeader.e_phoff + numBytesRead
) +
                                ", Offset = " + programHeader.p_offset +
                                ", Size = " + programHeader.p_filesz);
                }

                stream.Seek(programHeader.p_offset, SeekOrigin.Begin);  // Go to
 program content in ELF file
                data = new byte[programHeader.p_filesz];
                stream.Read(data, 0, (int)programHeader.p_filesz);      // Load
content into byte array

                // Write contents to virtual memory
                for (int j = 0; j < data.Length; ++j)
                {
                    if ((programHeader.p_vaddr + j) > memSize)
                    {
                        retCode = 7;
                        if (loggingEnabled == true)
                        {
                            log.Info("Simulator.LoadFile: Memory needs exceed what was provided
");
                        }
                        return;
                    }
                    memory.WriteByte((uint)(programHeader.p_vaddr + j), data[j])
;
                }

                // Set stream for next iteration at the next program header
                stream.Seek(elfHeader.e_phoff + numBytesRead, SeekOrigin.Begin);
            }

            // Seek to the second section header - .text
            stream.Seek(elfHeader.e_shoff + elfHeader.e_shentsize, SeekOrigin.Be
gin);

            Section sectionHeader;  // struct to hold individual section header
info
            byte[] sectionHeaderData;
            numBytesRead = 0;   // for record of offset in elfHeader

            sectionHeaderData = new byte[elfHeader.e_shentsize];   // create arr
ay of size specified in elfHeader
            numBytesRead += stream.Read(sectionHeaderData, 0, (int)elfHeader.e_s
hentsize);
            sectionHeader = ByteArrayToStructure<Section>(sectionHeaderData);

            sim.textStart = sectionHeader.s_off;
        }

        /// <summary>
        /// Set up stream to get contents of ELF file
        /// </summary>
        public void ReadFile()
        {
            try
            {
                // Open ELF file and automatically close when finsihed
                using (FileStream stream = new FileStream(fileName, FileMode.Ope
n))
                {
                    if (loggingEnabled == true)
                    {
                        log.Info("Simulator.ReadFile: Reading " + fileName + "...");
                    }

                    elfHeader = new ELF();  // Declare new ELF struct to hold EL
F header data
                    byte[] data = new byte[Marshal.SizeOf(elfHeader)];  // Decla
re new byte array with size equal to ELF header struct
                    stream.Read(data, 0, data.Length);  // Read ELF header into
data byte array
                    elfHeader = ByteArrayToStructure<ELF>(data);    // Convert b
yte array to struct

                    // ELF signature found at https://en.wikipedia.org/wiki/List
_of_file_signatures
                    if (!(elfHeader.EI_MAG0 == 0x7F &&
                          elfHeader.EI_MAG1 == 0x45 &&
                          elfHeader.EI_MAG2 == 0x4C &&
                          elfHeader.EI_MAG3 == 0x46))
                    {
                        retCode = 4;
                        if (loggingEnabled == true)
                        {
                            log.Info("Simulator.ReadFile: File is not in the ELF format");
                        }
                        return;
                    }

                    if (elfHeader.EI_CLASS != 1)
                    {
                        retCode = 6;
                        if (loggingEnabled == true)
                        {
                            log.Info("Simulator.ReadFile: File is not in a 32−bit format");
                        }
                        return;
                    }

                    LoadFile(stream, elfHeader);
                }
            }
            catch (IOException)
            {
                retCode = 5;
                return;
            }
```

```
        }

        /// <summary>
        /// Verify options before reading the file
        /// </summary>
        /// <returns>The retCode:
        ///     0: Load success
        ///     1: Not a .exe file
        ///     2: Invalid memory size specified
        ///     3: File does not exist
        ///     4: File not in ELF format
        ///     5: Cannot access file
        ///     6: File is not 32-bit format
        ///     7: Not enough space in memory
        /// </returns>
        public void Continue()
        {
            // Verify that the filename has the .exe extension
            if (fileName.Length < 5)    // need this for if Load is clicked on G
UI and nothing selected on close of app
            {
                retCode = 1;
                if (loggingEnabled == true)
                {
                    log.Info("Simulator.Continue: File does not have the .exe extension");
                }
            }
            else if (!fileName.Substring(fileName.Length − 4).Equals(".exe"))
            {
                retCode = 1;
                if (loggingEnabled == true)
                {
                    log.Info("Simulator.Continue: File does not have the .exe extension");
                }
                return;
            }

            // Verify that the memory size is within acceptable boundaries:
            // Between 0 and 1,000,000
            if ((memSize > 1000000) || (memSize < 0))
            {
                retCode = 2;
                if (loggingEnabled == true)
                {
                    log.Info("Simulator.Continue: Memory is not within bounds");
                }
                return;
            }

            // Verify that the file specified actually exists
            if (!File.Exists(fileName))
            {
                retCode = 3;
                if (loggingEnabled == true)
                {
                    log.Info("Simulator.Continue: File does not exist");
                }
                return;
            }
            else   // Otherwise, read the file
            {
                ReadFile();
                return;
            }
        }

        /// <summary>
        /// Convert a byte array to a <Program> struct
        /// From Dr. Schaub
        /// </summary>
        /// <typeparam name="T">the <Program> struct</typeparam>
        /// <param name="bytes">data to convert</param>
        static T ByteArrayToStructure<T>(byte[] bytes) where T : struct
        {
            GCHandle handle = GCHandle.Alloc(bytes, GCHandleType.Pinned);
            T stuff = (T)Marshal.PtrToStructure(handle.AddrOfPinnedObject(),
                typeof(T));
            handle.Free();
            return stuff;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace armsim
{
    class LoadStore : Instruction
    {
        /// <summary>
        /// Represents the information found in load/store instruction
        /// </summary>
        public LoadStore(uint instr) : base(instr)
        {
            p_LS = FindP();
            u_LS = FindU();
            b_LS = FindB();
            w_LS = FindW();
            l_LS = FindL();
            rn_LS = FindRn();
            rd_LS = FindRd();

            if (type == 0b010)       // load/store with immediate offset
            {
                typeLS = TypeLS.Imm;
                imm_LS = FindImm();
            }
            else if (type == 0b011)  // load/store with register/shift involved
            {
                typeLS = TypeLS.Register;
                shiftAmount_LS = FindShiftAmount();
                shiftCode_LS = FindShiftCode();
                rm_LS = FindRm();
            }
            else { }

            Disassemble();
        }

        /// <summary>
        /// Extracts the P bit and
        /// stores in the instance variable p
        /// </summary>
        private uint FindP()
        {
            return Memory.ShiftToEnd(instr, 24, 24);
        }

        /// <summary>
        /// Extracts the U bit and
        /// stores in the instance variable u
        /// </summary>
        private uint FindU()
        {
            return Memory.ShiftToEnd(instr, 23, 23);
        }

        /// <summary>
        /// Extracts the B bit and
        /// stores in the instance variable b
        /// </summary>
        private uint FindB()
        {
            return Memory.ShiftToEnd(instr, 22, 22);
        }

        /// <summary>
        /// Extracts the W bit and
        /// stores in the instance variable w
        /// </summary>
        private uint FindW()
        {
            return Memory.ShiftToEnd(instr, 21, 21);
        }

        /// <summary>
        /// Extracts the L bit and
        /// stores in the instance variable l
        /// </summary>
        private uint FindL()
        {
            return Memory.ShiftToEnd(instr, 20, 20);
        }

        /// <summary>
        /// Extracts the Rn nibble and
        /// stores in the instance variable rn
        /// </summary>
        private uint FindRn()
        {
            return Memory.ShiftToEnd(instr, 16, 19);
        }

        /// <summary>
        /// Extracts the Rd nibble and
        /// stores in the instance variable rd
        /// </summary>
        private uint FindRd()
        {
            return Memory.ShiftToEnd(instr, 12, 15);
        }

        /// <summary>
        /// Extracts the 12 bit imm value and
        /// stores in the instance variable imm
        /// </summary>
        private uint FindImm()
        {
            return Memory.ShiftToEnd(instr, 0, 11);
        }

        /// <summary>
        /// Extracts the Rm nibble and
        /// stores in the instance variable rm
        /// </summary>
        private uint FindRm()
        {
            return Memory.ShiftToEnd(instr, 0, 3);
        }

        /// <summary>
        /// Extracts the 2 bit shift code and
        /// stores in the instance variable shiftCode
        /// </summary>
        private uint FindShiftCode()
        {
            return Memory.ShiftToEnd(instr, 5, 6);
        }

        /// <summary>
        /// Extracts the 5 bit shift amount and
        /// stores in the instance variable shiftAmount
        /// </summary>
        private uint FindShiftAmount()
        {
            return Memory.ShiftToEnd(instr, 7, 11);
        }

        /// <summary>
```

```
        /// Produce the textual representation
        /// </summary>
        public void Disassemble()
        {
            string typeString = "";
            string lengthString = "";
            string signString = "";

            if (l_LS == 1)
            {
                typeString = "ldr";
            }
            else
            {
                typeString = "str";
            }

            if (b_LS == 1)
            {
                lengthString = "b";
            }

            if (u_LS == 0)
            {
                signString = "-";
            }

            if (type == 0b010)       // load/store with immediate offset
            {
                text = typeString + lengthString + FindCondString() + "r" + rd_L
S + ", [r" + rn_LS + ", #" + signString + imm_LS + "]";
            }
            else if (type == 0b011)  // load/store with register/shift involved
            {
                if ((shiftAmount_LS == 0) &&    // register offset
                    (shiftCode_LS == 0))
                {
                    text = typeString + lengthString + FindCondString() + "r" +
rd_LS + ", [r" + rn_LS + ", r" + rm_LS + "]";
                }
                else    // scaled register offset
                {
                    text = typeString + lengthString + FindCondString() + "r" +
rd_LS + ", [r" + rn_LS + ", r" + rm_LS + ", " + FindShiftString() + " #" + shiftAmoun
t_LS + "]";
                }
            }
            else { }

            if (w_LS == 1)
            {
                text += "!";
            }
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace armsim
{
    class LoadStoreMultiple : Instruction
    {
        /// <summary>
        /// Represents the information found in load/store multiple instruction
        /// </summary>
        public LoadStoreMultiple(uint instr) : base(instr)
        {
            regsList_LSM = FindRegsList();
            p_LSM = FindP();
            u_LSM = FindU();
            s_LSM = FindS();
            w_LSM = FindW();
            l_LSM = FindL();
            rn_LSM = FindRn();

            Disassemble();
        }

        private List<int> FindRegsList()
        {
            List<int> regs_List = new List<int>();

            for (uint i = 0; i <= 15; ++i)
            {
                if (Memory.ShiftToEnd(instr, i, i) == 1)
                {
                    regs_List.Add((int) i);
                }
            }

            return regs_List;
        }

        /// <summary>
        /// Extracts the P bit and
        /// stores in the instance variable p
        /// </summary>
        private uint FindP()
        {
            return Memory.ShiftToEnd(instr, 24, 24);
        }

        /// <summary>
        /// Extracts the U bit and
        /// stores in the instance variable u
        /// </summary>
        private uint FindU()
        {
            return Memory.ShiftToEnd(instr, 23, 23);
        }

        /// <summary>
        /// Extracts the B bit and
        /// stores in the instance variable b
        /// </summary>
        private uint FindS()
        {
            return Memory.ShiftToEnd(instr, 22, 22);
        }

        /// <summary>
        /// Extracts the W bit and
```

```
        /// stores in the instance variable w
        /// </summary>
        private uint FindW()
        {
            return Memory.ShiftToEnd(instr, 21, 21);
        }

        /// <summary>
        /// Extracts the L bit and
        /// stores in the instance variable l
        /// </summary>
        private uint FindL()
        {
            return Memory.ShiftToEnd(instr, 20, 20);
        }

        /// <summary>
        /// Extracts the Rn nibble and
        /// stores in the instance variable rn
        /// </summary>
        private uint FindRn()
        {
            return Memory.ShiftToEnd(instr, 16, 19);
        }

        /// <summary>
        /// Produce the textual representation
        /// </summary>
        public void Disassemble()
        {
            string typeString = "";
            string wString = "";

            if (l_LSM == 1)
            {
                typeString = "ldm";
            }
            else
            {
                typeString = "stm";
            }

            if (w_LSM == 1)
            {
                wString = "!";
            }

            text = typeString + FindCondString() + " r" + rn_LSM + wString + ",{"
;

            for (int i = 0; i < regsList_LSM.Count; ++i)
            {
                if (i < regsList_LSM.Count - 1)
                {
                    text += "r" + regsList_LSM[i] + ",";
                }
                else
                {
                    text += "r" + regsList_LSM[i];
                }
            }

            text += "}";
        }
    }
}
```

```
// Memory.cs
// Acts as the virtual memory with
// methods to read, write, and manipulate

using System;
using System.Collections.Generic;
using System.Diagnostics.Eventing.Reader;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Automation.Peers;

namespace armsim
{
    // Virtual memory for microprocessor
    public class Memory
    {
        public byte[] memArray;    // holds the contents of virtual memory

        /// <summary>
        /// Constructor for the Memory class
        /// </summary>
        /// <param name="memArraySize">Size of virutal memory</param>
        public Memory(int memArraySize)
        {
            memArray = new byte[memArraySize];
        }

        /// <summary>
        /// Reads a word (32 bits) from virtual memory
        /// </summary>
        /// <param name="addr">The address in virtual memory to begin reading</p
aram>
        /// <returns>An unsigned int comprised of the 4 bytes read</returns>
        public uint ReadWord(uint addr)
        {
            // If address given is not divisible by 4, throw an exception
            if ((addr % 4) != 0)
            {
                throw new Exception("ReadWord: Attmpted to read a word at an address not divisible
by 4.");
            }
            else
            {
                uint tempMem = 0;

                // OR all 4 bytes together, shifting the bits left by 8 each tim
e
                for (int i = 0; i < 4; ++i)
                {
                    tempMem |= (uint)((memArray[addr + i]) << (8 * i));
                }

                return tempMem;
            }
        }

        /// <summary>
        /// Reads half of a word (16 bits) from virtual memory
        /// </summary>
        /// <param name="addr">The address in virtual memory to begin reading</p
aram>
        /// <returns>An unsigned short comprised of the 2 bytes read</returns>
        public ushort ReadHalfWord(uint addr)
        {
            // If address given is not divisible by 2, throw an exception
            if ((addr % 2) != 0)
            {
                throw new Exception("ReadHalfWord: Attmpted to read a half−word at an address not
```

```
divisible by 2.");
            }
            else
            {
                ushort tempMem = 0;

                // OR both bytes together, shifting the bits left by 8 each time
                for (int i = 0; i < 2; ++i)
                {
                    tempMem |= (ushort)((memArray[addr + i]) << (8 * i));
                }

                return tempMem;
            }
        }

        /// <summary>
        /// Reads one byte (8 bits) from virtual memory
        /// </summary>
        /// <param name="addr">The address in virtual memory to begin reading</p
aram>
        /// <returns>The byte at the address specified</returns>
        public byte ReadByte(uint addr)
        {
            return memArray[addr];
        }

        /// <summary>
        /// Writes a word (32 bits) to virtual memory
        /// </summary>
        /// <param name="addr">The address in virtual memory to begin writing</p
aram>
        /// <param name="data">An unsigned int to write to virtual memory</param
>
        public void WriteWord(uint addr, uint data)
        {
            // If address given is not divisible by 4, throw an exception
            if ((addr % 4) != 0)
            {
                throw new Exception("WriteWord: Attmpted to write a word to an address not divisibl
e by 4.");
            }
            else
            {
                // Takes one byte at a time from the uint and writes to virtual
memory
                for (int i = 0; i < 4; ++i)
                {
                    memArray[addr + i] = (byte)(data & 0xFF);
                    data = (uint)(data >> 8);
                }
            }
        }

        /// <summary>
        /// Writes a word (16 bits) to virtual memory
        /// </summary>
        /// <param name="addr">The address in memory to begin writing</param>
        /// <param name="data">An unsigned short to write to virtual memory</par
am>
        public void WriteHalfWord(uint addr, ushort data)
        {
            // If address given is not divisible by 2, throw an exception
            if ((addr % 2) != 0)
            {
                throw new Exception("WriteHalfWord: Attmpted to write a half−word to an address n
ot divisible by 2.");
            }
            else
            {
                // Takes one byte at a time from the ushort and writes to virtua
l memory
                for (int i = 0; i < 2; ++i)
                {
                    memArray[addr + i] = (byte)(data & 0xFF);
                    data = (ushort)(data >> 8);
                }
            }
        }

        /// <summary>
        /// Writes a byte (8 bits) to virtual memory
        /// </summary>
        /// <param name="addr">the address in memory to begin writing</param>
        /// <param name="data">A byte to write to virtual memory</param>
        public void WriteByte(uint addr, byte data)
        {
            memArray[addr] = data;
        }

        /// <summary>
        /// Calculates the checksum of based on the virtual memory for testing p
urposes
        /// </summary>
        /// <returns>An int that is the checksum</returns>
        public int CalculateChecksum()
        {
            int checksum = 0;
            for (int i = 0; i < memArray.Length; ++i)
            {
                checksum += memArray[i] ^ i;   // XORs each byte with its addres
s and add them together
            }
            return checksum;
        }

        /// <summary>
        /// Reads the word at addr and returns information about a specific bit
in the word
        /// </summary>
        /// <param name="addr">The address in virtual memory to begin reading</p
aram>
        /// <param name="bit">The bit in the word to test, beginning at the leas
t significant bit</param>
        /// <returns>If bit is 1, return true.
        ///          If bit is 0, return false.</returns>
        public bool TestFlag(uint addr, int bit)
        {
            uint data = ReadWord(addr);
            uint result = data & (uint)(0x00000001 << bit);

            return result != 0;
        }

        /// <summary>
        /// Sets a specific bit in the word
        /// </summary>
        /// <param name="addr">The address in virtual memory to begin reading</p
aram>
        /// <param name="bit">The bit in the word to set, beginning at the least
 significant bit</param>
        /// <param name="flag">If flag is true, set bit to 1.
        ///                    If flag is false, set bit to 0.</param>
        public void SetFlag(uint addr, int bit, bool flag)
        {
            uint data = ReadWord(addr);
            uint result = 0;

            if (flag)
            {
```

```
                result = data | (uint)(0x00000001 << bit);
            }
            else
            {
                uint tempData = ~data;
                tempData |= (uint)(0x00000001 << bit);
                result = ~tempData;
            }

            WriteWord(addr, result);
        }

        /// <summary>
        /// Zeroes out all the bits in word except for those in the range startB
it...endBit
        /// </summary>
        /// <param name="word">The initial word that will be manipulated</param>
        /// <param name="startBit">The lowest bit in the range</param>
        /// <param name="endBit">The highest bit in the range</param>
        /// <returns>An unsigned int representing only the bits in the range spe
cified</returns>
        public static uint ExtractBits(uint word, uint startBit, uint endBit)
        {
            if (startBit > endBit)
            {
                throw new Exception("ExtractBits: Gave a startBit greater than endBit.");
            }
            else
            {
                uint mask = 0;
                for (int i = (int)startBit; i <= (int)endBit; ++i)
                {
                    mask |= (uint)(0x00000001 << i);
                }
                return word & mask;
            }
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="word"></param>
        /// <param name="startBit"></param>
        /// <param name="endBit"></param>
        /// <returns></returns>
        public static uint ShiftToEnd(uint word, uint startBit, uint endBit)
        {
            return ExtractBits(word, startBit, endBit) >> (int)startBit;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace armsim
{
    class Multiply : Instruction
    {
        /// <summary>
        /// Represents the information found in multiply instruction
        /// </summary>
        public Multiply(uint instr) : base(instr)
        {
            s_M = FindS();
            rd_M = FindRd();
            rs_M = FindRs();
            rm_M = FindRm();

            Disassemble();
        }

        /// <summary>
        /// Extracts the S bit and
        /// stores in the instance variable s
        /// </summary>
        private uint FindS()
        {
            return Memory.ShiftToEnd(instr, 20, 20);
        }

        /// <summary>
        /// Extracts the Rd nibble and
        /// stores in the instance variable rd
        /// </summary>
        private uint FindRd()
        {
            return Memory.ShiftToEnd(instr, 16, 19);
        }

        /// <summary>
        /// Extracts the Rd nibble and
        /// stores in the instance variable rd
        /// </summary>
        private uint FindRs()
        {
            return Memory.ShiftToEnd(instr, 8, 11);
        }

        /// <summary>
        /// Extracts the Rd nibble and
        /// stores in the instance variable rd
        /// </summary>
        private uint FindRm()
        {
            return Memory.ShiftToEnd(instr, 0, 3);
        }

        /// <summary>
        /// Produce the textual representation
        /// </summary>
        public void Disassemble()
        {
            text = "mul" + FindCondString() + " r" + rd_M + ",r" + rm_M + ",r" + r
s_M;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace armsim
{
    /// <summary>
    /// Represents the information found in swi instruction
    /// </summary>
    class SWI : Instruction
    {
        public SWI(uint instr) : base(instr)
        {
            offset_swi = FindOffset();

            Disassemble();
        }

        /// <summary>
        /// Extracts the first 24 bits of the encoding
        /// </summary>
        public uint FindOffset()
        {
            return Memory.ShiftToEnd(instr, 0, 23);
        }

        /// <summary>
        /// Produce the textual representation
        /// </summary>
        public void Disassemble()
        {
            text = "swi #" + offset_swi;
        }
    }
}
```