Component Based MMIX Simulator using Multiple Programming Paradigms

A dissertation submitted in partial fulfillment of the requirements for the MSc in Advanced Computing Technologies

by Stephen Edmans

Department of Computer Science and Information Systems

Birkbeck College, University of London

September 2015

This report is substantially the result of my own work except where explicitly indicated in the text. I give my permission for it to be submitted to the JISC Plagiarism Detection Service. I have read and understood the sections on plagiarism in the Programme Handbook and the College website.

Abstract

There are currently over 2,500[1] different programming languages, with more created every year. These programming languages can get grouped together in numerous different ways. This makes the decision of what language to use when starting a new project extremely difficult.

There are several ways in which we can reach this decision; choose the language that your team knows best; choose the language that makes the most sense to implement the critical part of your system; choose a simple general purpose language; choose a language that has got an active community. There is no acknowledged best approach to take.

Another approach would be to split your application up into separate components and using a different programming language for each component. This allows us choose the most appropriate programming language for each component.

The purpose of this project is to examine this approach. The application that we will create will be a simulator for an artificial machine language. The artificial machine language that we will use is called MMIX, it was developed by Donald Knuth as part of his seminal work The Art of Computer Programming[Knu11].

[1] From the language list[Kin]

Contents

List of Figures

Acknowledgments

I could not have completed this project without the help and patience of some important people. I would like to thank Dr. Keith Mannock for all his help and patience during the production of this project. I would also like to thank Dr. Nigel Martin for his support and encouragement. I would also like to thank Dr. Johnny Kelsey for his support and encouragement on an almost daily basis. Lastly, and definitely not leastly, I would like to thank Sam Forster for all of his support and encouragement, while putting up with me endlessly talking about this project.

Chapter 1

Introduction

As software systems get larger and more complex there is a need to handle this complexity. There is a prevailing design paradigm, which addresses these issues, that is to break these systems up into smaller components. This is a sentiment mentioned by Turner [Tur90] He calls these components "collections of modules", these components will interact with each other to make the complete system.

When you have control over the development of more than one of these components it is a traditional approach to use a single programming paradigm for your components. There is, however, no reason that you cannot use different languages and paradigms for these for each components. The goal of this project is to create a relatively complex system that is made up of multiple components where each component uses the most appropriate programming paradigm for the relevant component.

The system that we have created in this project was inspired by Jeliot [oJ07], which is a tool that is used as an aid in the teaching of Java. The Jeliot system allows a user to give it a piece of Java source code and it will show the user what the underlying java virtual machine is doing when it runs the code.

In his seminal work The Art of Computer Programming [Knu11] Professor Donald Knuth designed an artificial machine language that he called MIX. In a later volume of his work Professor Knuth updated this machine architecture, which he calls MMIX. He later detailed this new version of the architecture in a fascicle [Knu]. This project will create a system that take MMIX assembly code and shows the user, graphically, what the simulated machine is doing. It should be noted that all definitions of an MMIX computer and the assembly language used to program it come directly from either one of these sources.

Chapter 2

Assembler

2.1 Introduction

The first component that we developed takes a text file containing the MMIX assembly language code and translated it into a binary representation of the code. This component it typically called a *compiler*, to quote [ALSU06]

> A compiler is a program that can read a program in one language – the *source* language – and translate it into an equivalent program in another language – the *target* language.

A compiler operates as a series of phases, each of which transforms one representation of the source program into another. A typical decomposition of a compiler into phases, taken from [ALSU06] is shown in Figure 2.1.

A number if these phases are used to convert a higher level language down into a specific machine language. In this project we already start with a machine language, which means that we do not need these phases. A program that takes an assembly language file and translates it into machine language is typically called an *assembler*.

The four phases that we need for our project are Lexical Analysis, Syntax Analysis, Semantic Analysis and Code Generation. There are two of these phases, syntax analysis and semantic analysis, which are usually combined into a single phase, which is typically called a *Parser*.

The first thing that we need to do is decide which programming language is the most appropriate for this component. The component takes a fixed input and always produces the same output. The component does not contain any user interaction and it does not need a user interface. These requirements led us to choose a functional language for this component. The language we chose was Haskell.

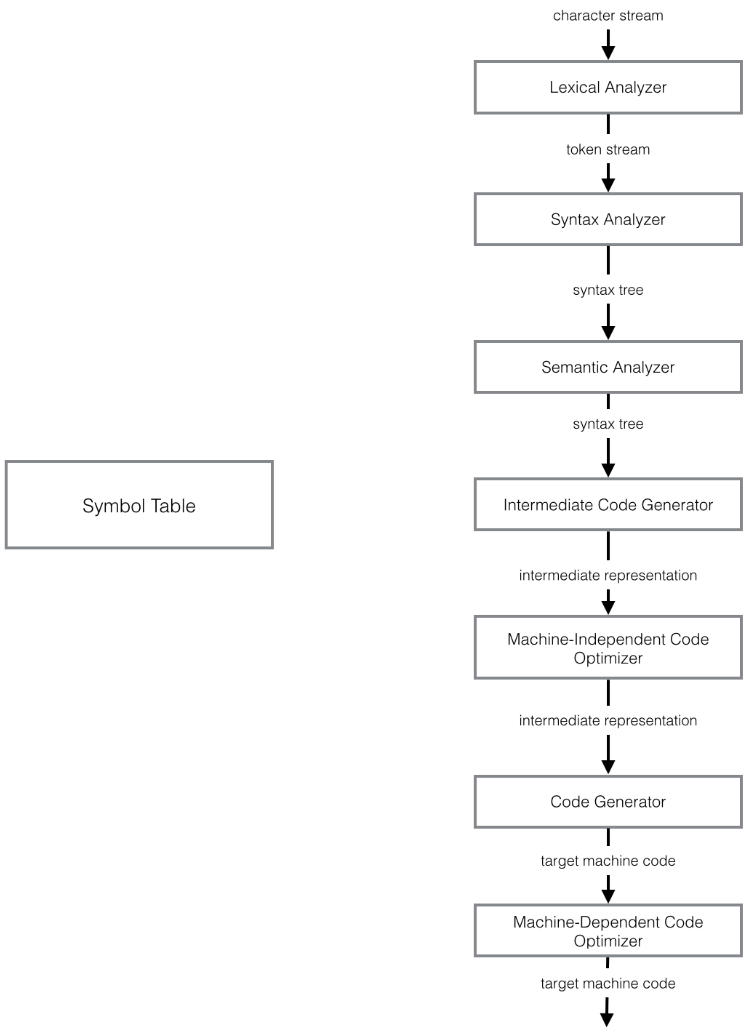

Figure 2.1: Phases of a compiler

We describe how each of these phases are implemented in Haskell in the next few sections.

2.2 Lexer

The initial phase of compilation, lexical analysis, takes a stream of characters and converts them into tokens. Lexical analysis is a well know problem and there are many tools that have been created to make this task simpler. There are several lexers that already exist for Haskell and we have chosen to use a lexer called Alex[Mar].

The first thing that you need to do when creating a lexer with Alex is to determine what set of tokens you will need to create. There are several parts to the MMIX assembly language (Mmixal). The first part is the way you define what operation should be performed by the computer at a specific point in the program. This is acheived with a machine language instrustion, which is typically called an Opcode. Each instruction needs some additional parameters which informs the computer on what the instruction should operate on, these parameters are typically called operands. There are two distinct types of Opcodes in Mmixal. The first type does not vary based on what operands are used with it. The second type does vary based on the operands, and the binary representation of the Opcode is different for the different set of operands. At the end of the assembly process these instructions will be converted into a binary representation that will be stored in the memory of an MMIX computer.

The next type of instruction is used by the assembler specify either the initial state of the computer or assign internal details used as part of the assembly process. The instructions are called, in fascile 1[Knu], pseudo instructions. These pseudo instructions do not necessarily result in anything being stored in memory.

The mmixal language also defines labels, registers, expressions and a few other things.

For this project we are using three basic groupings of tokens. The first group contains a single token, this token is for opcodes that do not vary based on their operands, we have called this token *TOpCodeSimple*. The second group of tokens also contains a single token, this is the token for opcodes that do vary based on their operands, we have called this token *TOpCode*. The third group contains all of the other tokens, see the code listing in Appendix B.1.1 for a complete list of these.

When you have determined what tokens are allowed you need to describe what sequences of characters should be converted to the individual tokens. The way that you describe the tokens in Alex is to create a list of regular expressions, for each regular expression you specify which token should be created if this sequence is found.

The Alex lexer tool allows us to insert code at specific points in the process. The functions that we are using allows us to simplify the process of creating tokens.

The Alex lexer requires us to store all of these definitions in a file with an extension of x. When all of the definitions have been completed you run the definition file through the Alex lexer tool and it generates a haskell source file that will perform the lexical analysis for you.

2.3 Parser

Once we have got a stream of tokens from the lexer, we need to perform syntax analysis and semantic analysis to make sure that the supplied program is both syntactically and semantically correct. Both of these steps are usually performed at the same time with a component called a *Parser*. Parsing, like lexical analysis, is a well know problem and there are many tools that have been created to make this task simpler. These tools are generally called parser generators as they take a definition file and produce the actual parser. There are several parser generators that already exist for Haskell and we have chosen to use a parser generator called Happy[GM].

The requirement for a parser is to take a stream of tokens, make sure that the stream is syntactically and semantically correct, and then output an intermediate representation of the code that can be used to generate the final binary representation.

The first thing that we did was to design our intermediate representation, there are four different type of code lines in mmixal, as shown below.

BUF	OCTA	0	*%Labelled Pseudo Instruction Line*
	LOC	#100	*%Plain Pseudo Instruction Line*
Main	JMP	9F	*%Labelled Opcode Line*
	STWU	n,ptop,jj	*%Plain Opcode Line*

The way that the Happy parser generator works is that we need to create a definition file that specifies all of the syntactically and semantically correct types of statements in our language. We specify the valid statements using a context free grammar. A context free grammar contains two basic parts, an identifier and list of tokens, or identifiers, that the identifier represents. A cut down version of the parser definition file we have used can be found in Figure 2.2 and a full description of the intermediate representation can be found in Appendix A. The complete definition file can be found in Appendix B.1.2.

The Happy parser generator requires us to store all of these definitions in a file with an extension of *y*. When all of the definitions have been completed you run the definition file through the Happy parser generator tool and it generates a Haskell source file that will perform the syntactic and semantic analysis for you.

```
Program         : AssignmentLines { reverse $1 }

AssignmentLines : {- empty -}        {[]}
                | AssignmentLines AssignmentLine { $2 : $1 }

AssignmentLine :: {Line}
AssingmentLine : OP_CODE OperatorList { defaultPlainOpCodeLine }
               | Identifier PI { defaultLabelledPILine }
               | Identifier OP_CODE OperatorList { defaultLabelledOpCodeLine }
               | PI { defaultPlainPILine }
               | OP_CODE_SIMPLE OperatorList { defaultPlainOpCodeLine }
               | Identifier OP_CODE_SIMPLE OperatorList { defaultLabelledOpCodeLine }
```

Figure 2.2: Sample Context Free Grammar

2.4 Code Generation

Now that we have got our program converted into an intermediate representation, in our case a list of *Lines*, we need to convert this into a binary representation. The mmixal language contains a set of features called *Local Labels* and processing these is the first step we perform when generating the code.

2.4.1 Local Labels

Local labels help compilers and programmers use names temporarily. They create symbols which are guaranteed to be unique over the entire scope of the input source code and which can be referred to by a simple notation. There are two parts to consider when implementing the local labels, the first is the location of the label itself, and the other is references to these labels used elsewhere in the program.

The location of a local label is specified by placing a single digit followed directly by an H, i.e. *0H* as a label at the start of a line. It should be noted that an individual local label can be specified many times in a single program, when they are referenced the closest label in the required direction is used. The way that we handle this is that we rename all of the local labels to system generated labels, these labels are actually illegal for user so we know that we are not creating duplicates. The format that we use is *??LS#H** where # is the local label number and * is a counter. We keep note of a separate counter for each of the possible local label numbers.

The local labels are referenced as either forward or backward references, i.e. *2B* specifies that we should look backwards in the code until we find a *2H* local label and use that local. To achieve this we start off by creating two separate maps, one for forward references and one for backward references. Initially the forward references point to the first possible local label for the mapped digit, the backward references do not contain a reference and any use of it is a semantic error in the program. When we have these maps we iterate through the program replacing any reference to a local label with the appropriate system generated label from the specific map. We then check to see if the line actually contains a local label specification, if it does we update both the forward and backward maps with the appropriate changes.

At the end of this process we have converted all local labels into system generated labels that can be handled as if they were ordinary user specified labels.

2.4.2 Symbol Table

As we can see in Figure 2.1 one of the data structures that we need to create as part of the code generation process is called a *Symbol Table*. This is simply a map that is used to record what labels have been specified and where in the program they actually point to. To create this we simply iterate through each of the lines of the program and if they contain a label we firstly check to see if it is already present and if it does not we add the label with the current location to the symbol table.

The symbol table is used extensively in the later steps of the code generation.

2.4.3 Assembler Directives

There are a number of pseudo instructions that give the assembler instructions on how we should direct the assembly of the program. There are a number of these directive including: -

- **IS** Defines the value of the label to be the value of the expression.
- **LOC** Changes the current location.
- **GREG** Allocates a new general purpose register, see section 2.4.4.
- **BYTE** Store an array of bytes in the current location.
- **WYDE** Store an array of wydes in the current location.
- **TETRA** Store an array of tetrabytes in the current location.
- **OCTA** Store an array of octabytes in the current location.

2.4.4 Automatically Assigned Registers

An MMix computer, by definition, contains 256 general purpose registers. The programmer can either specify which register to use directly or they can get the assembler to assign one automatically for them. A new general purpose register is allocated every time the assembler comes across a *GREG* pseudo instruction. The first register that is automatically assigned is $FE (254). Every, subsequent, automatically generated register uses the next lowest register. We have achieved this by iterating over the lines, sending along the value of the next assignable register. If the line is a GREG instruction then we change the command to one that contains the assigned register, and we then decrement the next assignable register before passing it on the the next line.

2.4.5 Handling Operands

Each opcode instructions can be supplied one, two or three operands to specify exactly what we expect to happen. The majority of opcodes can either be supplied with three registers, or it can be supplied with two registers and an immediate value. The registers could, of course, be replaced by labels which represent registers. If the line specifies three registers then the plain opcode is used when we generate the code. In the other case then when we generate the code we increment the opcode by one to let the computer know not to look for this register.

If the opcode is for a branching instruction then it will be supplied with a register and an address. For these instructions we need to determine the number of instructions between the current memory location and the memory location of the required address. We need the number of instructions, not just the difference in memory locations. This is calculated by determining the difference between the memory addresses and dividing this value by four. This address could be either ahead of, or before, the current location. If it is ahead of the current location then we use the plain opcode when generating the code. If the address is before the current location the we generate the code we increment the opcode by one.

2.4.6 Generating the Output

The final stage of the assembler is the actual outputting of the binary representation of the program. The way that we have achieved this is a two stage process. In the first stage we convert the intermediate representation of the code into a new representation of the code that makes creating the output file simpler.

```
CodeLine {
        cl_address = 256,
        cl_size = 4,
        cl_code = "\240\NUL\NUL\ETB"
```

```
}
```

This representation includes the start address for this line of code, the size of the code and the binary representation of that line of code.

The final stage is to output these code lines to a file. The structure of the file contains two separate parts, the first part contains the data the needs to be placed in memory, and the second part contains the initial values that the used registers need to be set to.

To create the first part we group the code lines together into contiguous blocks. The first four bytes of this section contains the number of blocks that we have got, we then include the details for each block. The first four bytes of each block contains the start address of the block, next next four bytes of the block contains the size of the block, after this we include the actual code for the block.

The first four bytes of the second part contains the number of registers we are defining. We then include the details for each register. The first byte of the register is register number, the next eight bytes are the initial value of that register.

2.5 Executable

Haskell, as a language, allows you to create an executable file from a collection of source code files. As with most programming languages that allow you to create an executable the language specifies the first function that is called when the application is run. In the case of Haskell that function is called main and it must return an IO monad.

The main function in our assembler application is stored in a source code file called Main.hs. The instruction that you need to run to create the executable, in this case the executable is called MMixAssembler, is: -

```
ghc -o MMixAssembler --make Main
```

What the function does is that it, firstly, is that it makes sure that the application is supplied with a single argument. This argument should be the name of the file containing the mmix assembly language source.

The application will compile the source code and output the assembled program in a file in the same folder as the source file, where the file extension has been changed to “se”.

2.6 Component Testing

When it comes to testing this component we tested it these levels.

- We tested the lexer on its own.
- We tested the lexer and parser together.
- We tested the Local Label generation on its own.
- We tested the Automatically Assigned Registers on their own.
- We tested the Code Generation on its own.
- We tested the component as a whole.

One of the questions that came out of the development of the assembler is given that we use some tools that generate code for us, how much should we test the code generation. As these components have been successfully used for a number of years we decided that we did not need to test the code generation in this case.

There are several sample programs that are written in mmixal, we used several of these when testing the assembler. The main mmixal program we used to test this component is one that determines the first 500 prime numbers. A fuller description of this program can be found at Chapter 5.2.1.

Chapter 3

Virtual Machine

3.1 Introduction

The majority of computers that currently exist are based on the Von Neumann Model, a description of this can be found at [Fus10]. The main parts of a computer can be summarised as a control unit, a processing unit, memory and some for of Input/Output devices.

To execute a program a Von Neumann computer repeatedly performs the following cycle of event:

1. Fetch instruction from memory.
2. Decode instruction.
3. Evaluate address.
4. Fetch operands from memory.
5. Execute operation.
6. Store results.
7. Increment the program counter.
8. Go back to step 1

This is typically called the fetch, decode, execute cycle.

The virtual machine that we have created has most of these parts, but it relies on the GUI for all of its Input/Output devices. The virtual machine that we have created does fetch, decode, execute cycle described above.

The first thing that we need to do is decide which programming language is the most appropriate for this component. The component takes a is supplied with a program and then it is continually asked to process the next statement. There is no shared state between the VM and any other components. These requirements led us to choose a message oriented language for this component. The language we chose was Erlang.

3.2 Memory

The way that memory is organized can be considered a hierarchy, to quote Aho et al[ALSU06]

> A memory hierarchy consists of several levels of storage with different speeds and sizes, with the levels closest to the processor being the fastest but smallest... Memory hierarchies are found in all machines. A processor usually has a small number of registers consisting of hundreds of bytes, several levels of caches containing kilobytes to megabytes, physical memory containing megabytes to gigabytes, and finally secondary storage that contains gigabytes and beyond.

For this project we will only be considering the physical memory and the registers.

Memory in an MMix computer works with patterns of 0s and 1s, commonly called *binary digits* or *bits*. A sequence of eight bits is commonly called a *byte*. An MMix computer can also handle 16 bit sequences, called a *wyde*, a 32 bit sequence, called a *tetrabyte* and a 64 bit sequence, called an *octabyte*.

When we are referencing memory we use $M[k]$ to denote that we are referencing the byte stored in memory address k. When we need to reference more that one byte at a time we use the following terminology.

Wyde

$M_2[0] = M_2[1] = M[0]M[1]$

TetraByte

$M_4[4k] = M_4[4k+1] = ... = M_4[4k+3] = M[4k]M[4k+1]...M[4k+3]$

OctaByte

$M_8[8k] = M_8[8k+1] = ... = M_8[8k+7] = M[8k]M[8k+1]...M[8k+7]$

Erlang contains a framework, called the open telecom platform, or OTP for short. This framework contains a number of implementations of common patterns, these patterns are called behaviours. We have simulated the memory inside the VM by using the *gen_server* OTP behaviour.

The gen_server behaviour not only creates a new actor but it also provides functions that handle the interaction with other actors and it handles the maintenance of state within the actor.

The data structure that we are using to contain the memory is called *Erlang Term Storage (ETS)*. To quote the Erlang documentation

> ETS provides the ability to store very large quantities of data in an Erlang runtime system, and to have constant access time to the data.

When you are using ETS you create individual tables that contain a collection of key value pair tuples. The table that we create for the memory uses the address as the key and the byte stored in the address as the value.

The data stored in an ETS table only exists while the process that owns it is in memory. When this process is terminated the table is automatically destroyed.

The other parts of the VM requests the memory server to either return or store bytes, wydes, tetrabytes or octabytes. There are a few extra requests that can be made that allow bulk operations, such as storing a completely new program.

3.3 Registers

An MMIX computer contains two distinct types of registers, 256 general purpose registers and 32 special purpose registers. A complete list of the special registers can be found in Figure 3.1. A Von Neumann computer uses a program counter to keep track of which instruction to execute next. This is not stored as a register in the definition of an MMix computer but to aid understanding of what the computer is doing we have included it as one of the special registers.

We have currently only implemented a small number of these special registers.

3.3.1 rA Arithmetic Status Register

The Arithmetic Status Register is used to keep track of any required arithmetic events. The eight bytes in the register are used for different flags. The least significant byte contains eight event bits. These bits are commonly referred to as DVWIOUZX, the meanings of these flags can be seen in Figure 3.2.

Identifier	Description
rA	Arithmetic Status Register
rB	Bootstrap Register
rC	Continuation Register
rD	Dividend Register
rE	Epsilon Register
rF	Failure Location Register
rG	Global Threshold Register
rH	Himult Register
rI	Interval Counter
rJ	Return-Jump Register
rK	Interrupt Mask Register
rL	Local Threshold Register
rM	Multiplex Mask Register
rN	Serial Number
rO	Register Stack Offset
rP	Prediction Register
rQ	Interrupt Request Register
rR	Remainder Register
rS	Register Stack Pointer
rT	Trap Address Register
rU	Usage Counter
rV	Virtual Translation Register
rW	Where Interrupted Register
rX	Execution Register
rY	Y Operand
rZ	Z Operand
rBB	Bootstrap Register
rTT	Dynamic Trap Address Register
rWW	Where Interrupted Register
rXX	Execution Register
rYY	Y Operand
rZZ	Z Operand

Figure 3.1: Special Registers

The next least significant byte contains eight "enable" bits with the same name DVWIOUZX and the same meanings.

When an exceptional condition occurs, there are two cases: If the corresponding enable bit is 0, the corresponding event bit is set to 1; but if the corresponding enable bit is 1, MMIX interrupts its current instruction stream and execute a special "exception handler". Thus, the event bits record exceptions that have not been "tripped".

Flag	**Description**
D	Integer Divide Check
V	Integer Overflow
W	Float-to-Fix Overflow
I	Invalid Operation
O	Floating Overflow
U	Floating Underflow
Z	Division by Zero
X	Floating Inexact

Figure 3.2: rA Register Flags

This leaves six high order bytes. At present, only two of those 48 bits are defined. The two bits corresponding to 2^{17} and 2^{16} in rA specify a rounding mode, as follows: -

00	Round to the nearest
01	Round off
10	Round up
11	Round down

We have implemented the arithmetic status register as a separate actor. This actor contains three pieces of state, the rounding mode, a set of flags to keep track of the enable bits that have been set, and a set of flags to keep track of the event buts that have been set.

This actor allows us to do four this, it allows us to change the rounding mode, it allows us to potentially flag an event as having happened, it allows us to remove a flag if it exists, and it allows us to find out what the actual value is recorded in the register. The way we have implemented the get value functionality is that we have assumed that this is the last thing that happens when processing an instruction. As such we reset all of the set event flags after we have determined the current value of the register.

3.3.2 General Purpose Registers

We are storing the general purpose registers inside a separate ETS table. This table uses the register identifier as the key and the octabyte stored in it as the value. When we start the VM, or when we send a new set of registers from the GUI, we create a new version of the ETS table.

3.4 Central Processing Unit

The Central Processing Unit (CPU) is at the heart of the virtual machine. The CPU is responsible for steps 2 - 7 in the fetch, decode, execute cycle. The first stage of fetch, decode, execute cycle that the CPU is responsible for is the decode stage. This is simple achieved with a function that pattern matches against the value and runs the relevant decoded function.

There are many different forms of instructions that the CPU could be running, as described in Chapter 2.

One of the sets of instruction can be specified in one of two ways, with three registers or with two registers and an immediate value, are all implemented in a similar manner. In the case where we have three registers we obtain the value stored in the third register and then both versions functions call a shared function which performs the actual operation.

Another set of instructions are the branching instructions. These instructions all take a single register and a sixteen bit number. The instructions look at the value in the register and if it matches a boolean function then processing moves, either forward or backwards, the number of instructions specified in the sixteen bit number. The actual new location is the current location, either plus or minus, four times the specified number of instructions.

Another set of instructions are used to conditionally set the value in a register. These instructions take two registers and a number. We use the same set of boolean functions as we have in the branching instructions against the second register. If the boolean function returns true then we set the first register to the number specified, if it returns true then we leave it alone. There is another set of instructions that are identical to this however if the boolean function returns false it set the first register to zero.

The execution of MMIX programs can be interrupted in several ways. The main way that it can happen in our VM is through the TRIP and TRAP instructions. The difference between these two instructions is that TRIPs interrupt the current execution and execute some user code, whereas TRAPs interrupt the current execution and execute an operating system command. We have not implemented the TRIPs but we have implemented a small number of TRAPs.

3.5 Calling the Operating System

There are many reasons that a program might need to interact with the operating system. It might need to read data in from secondary storage, it might need to output text to the user, it might need to be told that the current program has finished, it might need to obtain data entered in by the user.

These operations are executed through the TRAP instruction, for example

```
TRAP 0,Fputs,StdOut
```

tells the program to look at the memory location stored in general purpose register \$255. It will assume that this memory location is the start of a null terminated string. Therefore, it will read all of subsequent memory locations until it finds a location containing 0. It will assume that all of the other values for part of the string and it will send it to the operating systems currently defined standard output.

For the purpose of our application we consider the GUI to be the operating system. The way that the VM interacts with the GUI, in the case of TRAPs, is that it creates a list of tuples, the first element of the tuple contains a symbol that is known by the GUI. The remaining elements in the tuple are the data the GUI will need to perform the operation.

There is not a definitive list of what TRAP instructions there should be, but there is a list of rudimentary I/O commands that it should execute. We have actually only implemented a couple of the possible TRAP instructions.

3.6 Communication

The way we start the VM is by executing a function that starts all of the appropriate actors and the starts a UDP server, waiting for instructions from the GUI. When the VM receive a message it processes that message and then it will send a response back to the GUI.

The main message that we receive is the *process next statement* message. This message tells the VM that we need to perform the next fetch, decode and execute cycle. The response that is sent back to the GUI from this message contains three parts. The first part is a textual representation of the decoded instruction that we have just processed. The second part is a list of tuples containing the details of any registers that have changed value. The third part is tuples containing extra messages for the GUI, these extra messages contain instructions like changes to memory locations, text to be displayed on the console.

Another message that we might receive is a request to list all of the known registers and what values they have. The response to this message is a pair tuple containing a symbol telling the GUI that we are sending it all registers and a list of pair tuples that contain the register identifier and the value of the register for all of the known registers.

All other messages that are received are processed and then we respond with a symbol telling the GUI that we have finished processing.

3.7 Component Testing

The way that we have architected the VM we were able to do some extensive unit testing. We decided, however, that as the memory and register sections are basically extension to the erlang language ETS feature that we did not need to do extensive testing of this. The only part of this we tested was in the memory actor where we split integers up into bytes, or conversly, converting bytes back into integers.

One of the core parts of the virtual machine is the way it handles conditional statements and we performed some specific testing to confirm that this works properly.

3.7.1 Conditional Statements

The Mmix computer performs nine different types conditional statements, eight of these statements are supplied with a register, they check the value stored in the register, and then return either true or false. The other type of conditional statement takes two registers, or a register and a value, and compares these values. It puts a value into a third register based on the comparison performed.

To test this we put the following know values into some registers.

```
\$1 = 0
\$2 = -1
\$3 = 1
\$4 = 202
\$5 = 0xFFFFFFFFFFFFFFFF
```

With these values in mind we created the following test matrix. This shows us the results we would expect to get for each of our registers when used in the specific comparison test.

	Neg	**Zero**	**Pos**	**Odd**	**Not Neg**	**Not Zero**	**Not Pos**	**Even**
$1	F	T	F	F	T	F	T	T
$2	T	F	F	T	F	T	T	F
$3	F	F	T	T	T	T	F	F
$4	F	F	T	F	T	T	F	T
$5	T	F	F	T	F	T	T	F

Figure 3.3: Conditional Statement Test Matrix

We ran all of these tests until we managed to get the code to successfully pass all of these tests.

Chapter 4

Graphical User Interface

4.1 Introduction

One of the key objectives of the project chosen was to assist students understanding of how a computer program works. This objective makes the graphical user interface (GUI) a key component. A sample screenshot can be seen in Figure 4.1.

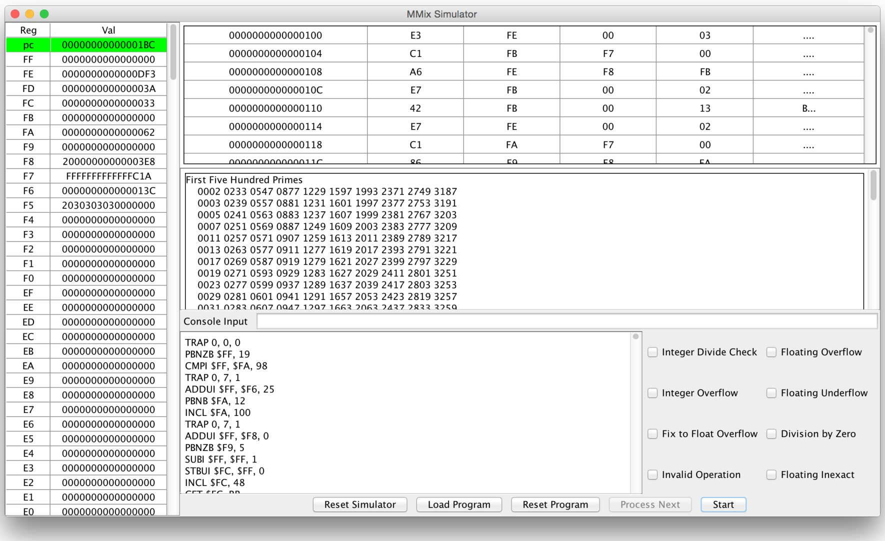

Figure 4.1: GUI Sample Screenshot

The first thing that we need to do is decide which programming language is the most appropriate for this component. This component needs to be able to display a graphical user interface and it also needs to be able to communicate with the Virtual Machine, see Chapter 3. These requirements led us to a more general purpose language and we chose to use Scala.

4.2 User Interface Design

We decided to split the user interface(UI) up into a number of smaller regions, which are typically called panels.

4.2.1 Console Panel

The console panel is a fairly simple panel that simulates the interaction between the computer and the end user. This panel contains two separate areas, the main area shows details that the programmer wants to inform the user about. The other area allows the user to enter some data and send it to the program, when they are so prompted.

4.2.2 Controls Panel

The controls panel is the area of the GUI that contains the buttons which allow the user to interact with the simulator. The controls panel contains the following buttons: -

- **Reset Simulator** – This button completely resets the simulator, removing any currently loaded program from memory.
- **Load Program** – This button allows the user to choose a file that contains the binary representation of an MMIX program. This program will then be loaded in from the file and passed on to the virtual machine. When this process has finished the application will be ready to start simulating the program. This process obviously has to be able to understand the binary representation of the mmixal program that our assembler creates.
- **Reset Program** – This button will reset the GUI and the virtual machine back to the same state it was in when the application was first loaded.
- **Process Next** – This button, when enabled, will process the next available statement.
- **Start / Stop** – This button allows the GUI to automate the processing of the next statements, freeing the user up from having to constantly press the process next button.

4.2.3 Main State Panel

The main state panel contains two separate areas, the main area contains a list of all of the instructions that the VM has processed. This list is displayed in reverse order so that the last statement executed is at the top of the list. The other area shows a number of flags, these flags are stored in the arithmetic status register, which is described in section 3.3.1.

4.2.4 Memory Panel

The memory panel contains a table that describes the current state of the memory in the VM. The each row in the table starts with a column the tells the user the starting memory address for this row. The next four columns give a hexadecimal representation of the contents of those memory locations. The final column contains an ASCII representation of the data in those four memory locations, if the cell contains a non printable ASCII character then it is substituted with a '.'.

If the processing of an instruction changes the content of a memory location then those memory locations are highlighted in green.

4.2.5 Registers Panel

The registers panel contains the current values stored in all of the general purpose and special registers. It also includes the current value of the additional program counter register we are using to keep count of the location of the next instruction to be processed.

If the processing of an instruction changes the content of a register then those registers are highlighted in green.

4.3 Asynchronous UI Programming with Actors

The development of a graphical user interface will always contain a few non-functional requirements, one that should always be present is that the GUI should be responsive. We mean, by this, that the GUI should always respond to what the user does with it, no matter what processing it is also doing at the same time. This has traditional been the domain of multi-threaded programming, however there is now a new approach available to us, we can use an actor library.

When we use an actor library you gain a mechanism for easily handling concurrency and parallelism. A highly regarded actor library that is available in Scala is called Akka [Inc]. To quote the Akka website: -

> Actors give you:
> - Simple and high-level abstractions for concurrency and parallelism.
> - Asynchronous, non-blocking and highly performant event-driven programming model.
> - Very lightweight event-driven processes (several million actors per GB of heap memory).

The way that we have designed the GUI is that we have created a separate actor for each of the panels and a separate actor that is responsible for communicating with the virtual machine. The panel actors are responsible for both updating the panels with information from the VM and handling any input from the users. The virtual machine actor is responsible for communicating with the VM.

4.4 Communication

There are two types of communication that we need to handle in the project, communication between the GUI and the BM, along with communication between the various UI panels.

4.4.1 GUI to VM Communication

We need a mechanism for transferring data and commands between the GUI written in Scala and the VM written in Erlang. There is a built in function inside Erlang which will convert a term, which is the name Erlang uses for a data structure, into binary. This function is called *term_to_binary* and it takes a value as a parameter. The function creates a binary representation based on a fixed format the can be found on the Erlang website [Eri].

We have written an object, in Scala, that will perform the same translation. It will convert binary streams from Erlang into Scala data structures and it will convert Scala data structures into a binary stream that Erlang can understand. This object has been embedded inside the Virtual Machine Actor, the full source code is available in Appendix B.2.10.

This object allows us the ability to marshal data structures between the two components but we also need to handle the actual transferral action. We have decided to, initially, require that both the GUI and VM processes should run on the same physical machine. This requirement allows us to simple create a UDP connection between the two processes.

4.4.2 UI Panel Communication

The communication between the UI panels, and the virtual machine actor all happens with Akka messages. As an example when we ask the VM to process the next statement it will respond with not only all of the things that have changed but what instruction was executed and all some updates to the console. This results in the virtual machine actor sending messages to the main state panel actor, the memory panel actor, the registers panel actor and potentially the console panel actor.

4.4.3 Automation

There is one of the features of the GUI we have not discussed yet, this is the automation feature. The initial version of the application just had a "process next statement", which works well for small programs. When you start using a bigger program like our test application, see section 5.2.1, this take a considerable amount of effort.

When we started simulating our test application we decided we wished to have a way that would automate the processing. The decision to use Akka actors made the implementation of this quite simple. The way that we implemented this was to add an automation state variable to the controls panel actor. We also added a button to the GUI so that the user can toggle between the *stopped* and *running* states.

With the state in place we amened the GUIs virtual machine actor to send an *automate* message to the controls panel whenever we have received the *update* message from the VM. When the controls panel actor receives the automate message it checks the current state. If we are in the *running* state it will use an Akka scheduler to simulate the pressing of the "process next" button after a short delay.

The current version of the application has a fixed delay used in the automation but it should be a simple matter to allow the user to alter this delay through the UI.

4.5 Component Testing

The GUI performs very little complex processing, the initial task is reading the data from our own specific binary file format. This is simply tested by reading in a number of different files.

The main complex task we have to perform is the communction between the GUI and the VM. We tested this by transferring a large number of different data structures from erlang to scala and vice versa.

Chapter 5

Simulator Application

5.1 Introduction

In the previous chapters we have described the individual components that make up the Simulator Application.

5.2 Integration Testing

```
        LOC     Data_Segment
        GREG    @
txt     BYTE    "Hello world!",10,0

        LOC     #100

Main    LDA     $255,txt
        TRAP    0,Fputs,StdOut
        TRAP    0,Halt,0
```

5.2.1 Generate Prime Numbers Sample Application

Conclusion

We found that having a single developer

Pattern Matching

Lists

Tuples

Possible Future Enhancements

Extra TRAP instructions.

Extra special registers.

Directly link the GUI with the Assembler.

References

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Eri] Ericsson. Erlang external term format. <`http://erlang.org/doc/apps/erts/erl_ext_dist.html` >[Access 7 September 2015].

[Fus10] Don Fussell. Von neumann model. Available at: <`http://www.cs.utexas.edu/users/fussell/cs310h/lectures/Lecture_9-310h.pdf`, 2010.

[GM] Andy Gill and Simon Marlow. Happy: The parser generator for haskell. <`https://www.haskell.org/happy/` >[Access 7 September 2015].

[Inc] Typesafe Inc. Akka toolkit. <`http://akka.io/` >[Access 24 August 2015].

[Kin] Bill Kinnersley. The language list. <`http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm` >[Access 7 September 2015].

[Knu] D.E. Knuth. The art of computer programming fascicle 1 mmix [e-book]. Stanford University: Addison Wesley Available through: Stanford University <`http://www-cs-faculty.stanford.edu/~uno/fasc1.ps.gz`>[Access 7 April 2013].

[Knu90] D.E. Knuth. *MMIXware A RISC Computer for the Third Millennium*. Springer, 1990.

[Knu11] D.E. Knuth. *The Art of Computer Programming*, volume 1-4a. 1st ed. Addison Wesley, 2011.

[Mar] Simon Marlow. Alex: A lexical analyser generator for haskell. <`https://www.haskell.org/alex/` >[Access 7 September 2015].

[oJ07] University of Joensuu. Jeliot 3. Available at: <`http://cs.joensuu.fi/jeliot/`, 2007.

[Ruc12] Martin Ruckert. Mmix quick reference card. <`http://mmix.cs.hm.edu/doc/mmix-refcard-a4.pdf` >[Access 24 August 2015], 2012.

[Tur90] D. Turner. Research topics in functional programming, addison-wesley. Available through <`http://www.cs.utexas.edu/~shmat/courses/cs345/whyfp.pdf`, 1990.

Appendix A

Intermediate Assembler Representations

A.1 Definitions

As we described in section 2.3 there are four different types of lines in an mmix application. These can be grouped together into two different sets. The first set is whether or not the line contains a pseudo instruction or an operation. The second set is whether or not the line contains a label.

The haskell language allows you to define a number of related types that contain different attributes, these are called *Algebraic Data Types* (ADT). We use this feature to define an algebraic data type with four different constructors to represent each line of code.

- PlainPILine
- LabelledPILine
- PlainOpCodeLine
- LabelledOpCodeLine

We use this ADT in our assembler as our intermediate representation.

A.2 Test Application

To demonstrate the intermediate representation we will include the source code for our test application and then we will show you the intermediate representation of this program.

A.2.1 Sample Test MMIXAL Code

The sample mmixal application I am using to test the system is taken from Fascile 1[Knu]. The complete code listing is: -

```
L      IS    500
t      IS    $255
n      GREG  0
q      GREG  0
r      GREG  0
jj     GREG  0
kk     GREG  0
pk     GREG  0
mm     IS    kk
       LOC   Data_Segment
PRIME1 WYDE  2
       LOC   PRIME1+2*L
ptop   GREG  @
j0     GREG  PRIME1+2-@
BUF    OCTA  0
       LOC   #100
Main   GREG  @
       SET   n,3
       SET   jj,j0
2H     STWU  n,ptop,jj
       INCL  jj,2
3H     BZ    jj,2F
4H     INCL  n,2
5H     SET   kk,j0
6H     LDWU  pk,ptop,kk
       DIV   q,n,pk
       GET   r,rR
       BZ    r,4B
7H     CMP   t,q,pk
       BNP   t,2B
8H     INCL  kk,2
       JMP   6B
       GREG  @
Title  BYTE  "First Five Hundred Primes"
NewLn  BYTE  #a,0
Blanks BYTE  "   ",0
2H     LDA   t,Title
       TRAP  0,Fputs,StdOut
       NEG   mm,2
3H     ADD   mm,mm,j0
       LDA   t,Blanks
       TRAP  0,Fputs,StdOut
2H     LDWU  pk,ptop,mm
0H     GREG  #2030303030000000
       STOU  0B,BUF
       LDA   t,BUF+4
1H     DIV   pk,pk,10
       GET   r,rR
       INCL  r,'0'
       STBU  r,t,0
       SUB   t,t,1
       PBNZ  pk,1B
       LDA   t,BUF
       TRAP  0,Fputs,StdOut
       INCL  mm,2*L/10
       PBN   mm,2B
       LDA   t,NewLn
       TRAP  0,Fputs,StdOut
       CMP   t,mm,2*(L/10-1)
       PBNZ  t,3B
       TRAP  0,Halt,0
```

A.2.2 Parsed Sample File

The final intermediate representation of the parsed source code for the test application is: -

```
LabelledPILine {
        lppl_id = IsNumber 500, lppl_ident = Id "L", lppl_loc = 0
}
LabelledPILine {
```

```
        lppl_id = IsRegister 255, lppl_ident = Id "t", lppl_loc = 0
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\254' (ExpressionNumber 0)), lppl_ident = Id "n"
             , lppl_loc = 0
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\253' (ExpressionNumber 0)), lppl_ident = Id "q"
             , lppl_loc = 0
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\252' (ExpressionNumber 0)), lppl_ident = Id "r"
             , lppl_loc = 0
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\251' (ExpressionNumber 0)), lppl_ident = Id "jj
             ", lppl_loc = 0
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\250' (ExpressionNumber 0)), lppl_ident = Id "kk
             ", lppl_loc = 0
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\249' (ExpressionNumber 0)), lppl_ident = Id "pk
             ", lppl_loc = 0
}
LabelledPILine {
        lppl_id = IsIdentifier (Id "kk"), lppl_ident = Id "mm", lppl_loc = 0
}
PlainPILine {
        ppl_id = LocEx (ExpressionNumber 2305843009213693952), ppl_loc = 2305843009213693952
}
LabelledPILine {
        lppl_id = WydeArray "\STX", lppl_ident = Id "PRIME1", lppl_loc = 2305843009213693952
}
PlainPILine {
        ppl_id = LocEx (ExpressionNumber 2305843009213694952), ppl_loc = 2305843009213694952
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\248' (ExpressionNumber 2305843009213694952)),
             lppl_ident = Id "ptop", lppl_loc = 2305843009213694952
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\247' (ExpressionNumber (-998))), lppl_ident =
             Id "j0", lppl_loc = 2305843009213694952
}
LabelledPILine {
        lppl_id = OctaArray "\NUL", lppl_ident = Id "BUF", lppl_loc = 2305843009213694952
}
PlainPILine {
        ppl_id = LocEx (ExpressionNumber 256), ppl_loc = 256
}
LabelledPILine {
        lppl_id = Set (Expr (ExpressionIdentifier (Id "n")),Expr (ExpressionNumber 3)),
             lppl_ident = Id "Main", lppl_loc = 256
}
PlainPILine {
        ppl_id = Set (Expr (ExpressionIdentifier (Id "jj")),Expr (ExpressionIdentifier (Id "j0
             "))), ppl_loc = 260
}
LabelledOpCodeLine {
        lpocl_code = 166, lpocl_ops = [Expr (ExpressionIdentifier (Id "n")),Expr (
             ExpressionIdentifier (Id "ptop")),Expr (ExpressionIdentifier (Id "jj"))],
             lpocl_ident = Id "??2H0", lpocl_loc = 264, lpocl_sim = False
}
PlainOpCodeLine {
        pocl_code = 231, pocl_ops = [Expr (ExpressionIdentifier (Id "jj")),Expr (
             ExpressionNumber 2)], pocl_loc = 268, pocl_sim = True
}
LabelledOpCodeLine {
        lpocl_code = 66, lpocl_ops = [Expr (ExpressionIdentifier (Id "jj")),Ident (Id "??2H1")
             ], lpocl_ident = Id "??3H0", lpocl_loc = 272, lpocl_sim = False
}
LabelledOpCodeLine {
        lpocl_code = 231, lpocl_ops = [Expr (ExpressionIdentifier (Id "n")),Expr (
             ExpressionNumber 2)], lpocl_ident = Id "??4H0", lpocl_loc = 276, lpocl_sim =
             True
}
LabelledPILine {
        lppl_id = Set (Expr (ExpressionIdentifier (Id "kk")),Expr (ExpressionIdentifier (Id "
             j0"))), lppl_ident = Id "??5H0", lppl_loc = 280
}
LabelledOpCodeLine {
```

```
        lpocl_code = 134, lpocl_ops = [Expr (ExpressionIdentifier (Id "pk")),Expr (
             ExpressionIdentifier (Id "ptop")),Expr (ExpressionIdentifier (Id "kk"))],
             lpocl_ident = Id "??6H0", lpocl_loc = 284, lpocl_sim = False
}
PlainOpCodeLine {
        pocl_code = 28, pocl_ops = [Expr (ExpressionIdentifier (Id "q")),Expr (
             ExpressionIdentifier (Id "n")),Expr (ExpressionIdentifier (Id "pk"))], pocl_loc
             = 288, pocl_sim = False
}
PlainOpCodeLine {
        pocl_code = 254, pocl_ops = [Expr (ExpressionIdentifier (Id "r")),Expr (
             ExpressionIdentifier (Id "rR"))], pocl_loc = 292, pocl_sim = True
}
PlainOpCodeLine {
        pocl_code = 66, pocl_ops = [Expr (ExpressionIdentifier (Id "r")),Ident (Id "??4H0")],
             pocl_loc = 296, pocl_sim = False
}
LabelledOpCodeLine {
        lpocl_code = 48, lpocl_ops = [Expr (ExpressionIdentifier (Id "t")),Expr (
             ExpressionIdentifier (Id "q")),Expr (ExpressionIdentifier (Id "pk"))],
             lpocl_ident = Id "??7H0", lpocl_loc = 300, lpocl_sim = False
}
PlainOpCodeLine {
        pocl_code = 76, pocl_ops = [Expr (ExpressionIdentifier (Id "t")),Ident (Id "??2H0")],
             pocl_loc = 304, pocl_sim = False
}
LabelledOpCodeLine {
        lpocl_code = 231, lpocl_ops = [Expr (ExpressionIdentifier (Id "kk")),Expr (
             ExpressionNumber 2)], lpocl_ident = Id "??8H0", lpocl_loc = 308, lpocl_sim =
             True
}
PlainOpCodeLine {
        pocl_code = 240, pocl_ops = [Ident (Id "??6H0")], pocl_loc = 312, pocl_sim = False
}
PlainPILine {
        ppl_id = GregEx (ExpressionRegister '\246' ExpressionAT), ppl_loc = 316
}
LabelledPILine {
        lppl_id = ByteArray "First Five Hundred Primes", lppl_ident = Id "Title", lppl_loc =
             316
}
LabelledPILine {
        lppl_id = ByteArray "\n\NUL", lppl_ident = Id "NewLn", lppl_loc = 341
}
LabelledPILine {
        lppl_id = ByteArray "   \NUL", lppl_ident = Id "Blanks", lppl_loc = 343
}
LabelledOpCodeLine {
        lpocl_code = 34, lpocl_ops = [Expr (ExpressionIdentifier (Id "t")),Expr (
             ExpressionIdentifier (Id "Title"))], lpocl_ident = Id "??2H1", lpocl_loc = 348,
             lpocl_sim = False
}
PlainOpCodeLine {
        pocl_code = 0, pocl_ops = [Expr (ExpressionNumber 0),PseudoCode 7,PseudoCode 1],
             pocl_loc = 352, pocl_sim = True
}
PlainOpCodeLine {
        pocl_code = 52, pocl_ops = [Expr (ExpressionIdentifier (Id "mm")),Expr (
             ExpressionNumber 2)], pocl_loc = 356, pocl_sim = False
}
LabelledOpCodeLine {
        lpocl_code = 32, lpocl_ops = [Expr (ExpressionIdentifier (Id "mm")),Expr (
             ExpressionIdentifier (Id "mm")),Expr (ExpressionIdentifier (Id "j0"))],
             lpocl_ident = Id "??3H1", lpocl_loc = 360, lpocl_sim = False
}
PlainOpCodeLine {
        pocl_code = 34, pocl_ops = [Expr (ExpressionIdentifier (Id "t")),Expr (
             ExpressionIdentifier (Id "Blanks"))], pocl_loc = 364, pocl_sim = False
}
PlainOpCodeLine {
        pocl_code = 0, pocl_ops = [Expr (ExpressionNumber 0),PseudoCode 7,PseudoCode 1],
             pocl_loc = 368, pocl_sim = True
}
LabelledOpCodeLine {
        lpocl_code = 134, lpocl_ops = [Expr (ExpressionIdentifier (Id "pk")),Expr (
             ExpressionIdentifier (Id "ptop")),Expr (ExpressionIdentifier (Id "mm"))],
             lpocl_ident = Id "??2H2", lpocl_loc = 372, lpocl_sim = False
}
LabelledPILine {
        lppl_id = GregEx (ExpressionRegister '\245' (ExpressionNumber 2319406791617675264)),
             lppl_ident = Id "??0H0", lppl_loc = 376
}
PlainOpCodeLine {
```

```
        pocl_code = 174, pocl_ops = [Ident (Id "??0H0"),Expr (ExpressionIdentifier (Id "BUF"))
             ], pocl_loc = 376, pocl_sim = False
}
PlainOpCodeLine {
        pocl_code = 34, pocl_ops = [Expr (ExpressionIdentifier (Id "t")),Expr (
             ExpressionNumber 2305843009213694956)], pocl_loc = 380, pocl_sim = False
}
LabelledOpCodeLine {
        lpocl_code = 28, lpocl_ops = [Expr (ExpressionIdentifier (Id "pk")),Expr (
             ExpressionIdentifier (Id "pk")),Expr (ExpressionNumber 10)], lpocl_ident = Id "
             ??1H0", lpocl_loc = 384, lpocl_sim = False
}
PlainOpCodeLine {
        pocl_code = 254, pocl_ops = [Expr (ExpressionIdentifier (Id "r")),Expr (
             ExpressionIdentifier (Id "rR"))], pocl_loc = 388, pocl_sim = True
}
PlainOpCodeLine {
        pocl_code = 231, pocl_ops = [Expr (ExpressionIdentifier (Id "r")),Expr (
             ExpressionNumber 48)], pocl_loc = 392, pocl_sim = True
}
PlainOpCodeLine {
        pocl_code = 162, pocl_ops = [Expr (ExpressionIdentifier (Id "r")),Expr (
             ExpressionIdentifier (Id "t")),Expr (ExpressionNumber 0)], pocl_loc = 396,
             pocl_sim = False
}
PlainOpCodeLine {
        pocl_code = 36, pocl_ops = [Expr (ExpressionIdentifier (Id "t")),Expr (
             ExpressionIdentifier (Id "t")),Expr (ExpressionNumber 1)], pocl_loc = 400,
             pocl_sim = False
}
PlainOpCodeLine {
        pocl_code = 90, pocl_ops = [Expr (ExpressionIdentifier (Id "pk")),Ident (Id "??1H0")],
              pocl_loc = 404, pocl_sim = False
}
PlainOpCodeLine {
        pocl_code = 34, pocl_ops = [Expr (ExpressionIdentifier (Id "t")),Expr (
             ExpressionIdentifier (Id "BUF"))], pocl_loc = 408, pocl_sim = False
}
PlainOpCodeLine {
        pocl_code = 0, pocl_ops = [Expr (ExpressionNumber 0),PseudoCode 7,PseudoCode 1],
             pocl_loc = 412, pocl_sim = True
}
PlainOpCodeLine {
        pocl_code = 231, pocl_ops = [Expr (ExpressionIdentifier (Id "mm")),Expr (
             ExpressionNumber 100)], pocl_loc = 416, pocl_sim = True
}
PlainOpCodeLine {
        pocl_code = 80, pocl_ops = [Expr (ExpressionIdentifier (Id "mm")),Ident (Id "??2H2")],
              pocl_loc = 420, pocl_sim = False
}
PlainOpCodeLine {
        pocl_code = 34, pocl_ops = [Expr (ExpressionIdentifier (Id "t")),Expr (
             ExpressionIdentifier (Id "NewLn"))], pocl_loc = 424, pocl_sim = False
}
PlainOpCodeLine {
        pocl_code = 0, pocl_ops = [Expr (ExpressionNumber 0),PseudoCode 7,PseudoCode 1],
             pocl_loc = 428, pocl_sim = True
}
PlainOpCodeLine {
        pocl_code = 48, pocl_ops = [Expr (ExpressionIdentifier (Id "t")),Expr (
             ExpressionIdentifier (Id "mm")),Expr (ExpressionNumber 98)], pocl_loc = 432,
             pocl_sim = False
}
PlainOpCodeLine {
        pocl_code = 90, pocl_ops = [Expr (ExpressionIdentifier (Id "t")),Ident (Id "??3H1")],
             pocl_loc = 436, pocl_sim = False
}
PlainOpCodeLine {
        pocl_code = 0, pocl_ops = [Expr (ExpressionNumber 0),PseudoCode 0,Expr (
             ExpressionNumber 0)], pocl_loc = 440, pocl_sim = True
}
```

Appendix B

Source Code

This appendix contains the source code for all of our components.

B.1 Assembler

B.1.1 Lexer

```
{
module MMix_Lexer where

import Data.Char (chr)
import Numeric (readDec)
import Numeric (readHex)

import Debug.Trace
}

%wrapper "monadUserState"

$digit    = 0-9             -- digits
$alpha    = [a-zA-Z]        -- alphabetic characters
$hexdigit = [0-9a-eA-E]     -- hexadecimal digits

tokens :-

<0>$white+                                  ;
<0>[Ii][Ss]                         { mkT TIS }
<0>[Gg][Rr][Ee][Gg]                 { mkT TGREG }
<0>[Ll][Oo][Cc]                     { mkT TLOC }
<0>[Bb][Yy][Tt][Ee]                 { mkT TByte }
<0>[Ww][Yy][Dd][Ee]                 { mkT TWyde }
<0>[Tt][Ee][Tt][Rr][Aa]             { mkT TTetra }
<0>[Oo][Cc][Tt][Aa]                 { mkT TOcta }
<0>[Ss][Ee][Tt]                     { mkT TSet }
<0>($digit)H                        { mkLocalLabel }
<0>($digit)F                        { mkLocalForwardOperand }
<0>($digit)B                        { mkLocalBackwardOperand }
<0>Text_Segment                     { mkT TTextSegment }
<0>Data_Segment                     { mkT TDataSegment }
<0>Pool_Segment                     { mkT TPoolSegment }
<0>Stack_Segment                    { mkT TStackSegment }
<0>\@                               { mkT TAtSign }
<0>\+                               { mkT TPlus }
<0>\-                               { mkT TMinus }
<0>\*                               { mkT TMult }
<0>\/                               { mkT TDivide }
<0>\(                               { mkT TOpenParen }
<0>\)                               { mkT TCloseParen }
<0>\#$hexdigit+                     { mkHex }
```

```
<0>\$$digit+                            { mkRegister }
<0>[Tt][Rr][Aa][Pp]                     { mkT $ TOpCodeSimple 0x00 }
<0>[Ff][Cc][Mm][Pp]                     { mkT $ TOpCodeSimple 0x01 }
<0>[Ff][Uu][Nn]                         { mkT $ TOpCodeSimple 0x02 }
<0>[Ff][Ee][Qq][Ll]                     { mkT $ TOpCodeSimple 0x03 }
<0>[Ff][Aa][Dd][Dd]                     { mkT $ TOpCodeSimple 0x04 }
<0>[Ff][Ii][Xx]                         { mkT $ TOpCodeSimple 0x05 }
<0>[Ff][Ss][Uu][Bb]                     { mkT $ TOpCodeSimple 0x06 }
<0>[Ff][Ii][Xx][Uu]                     { mkT $ TOpCodeSimple 0x07 }
<0>[Ff][Ll][Oo][Tt]                     { mkT $ TOpCode 0x08 }
<0>[Ff][Ll][Oo][Tt][Uu]                 { mkT $ TOpCode 0x0A }
<0>[Ss][Ff][Ll][Oo][Tt]                 { mkT $ TOpCode 0x0C }
<0>[Ss][Ff][Ll][Oo][Tt][Uu]             { mkT $ TOpCode 0x0E }
<0>[Ff][Mm][Uu][Ll]                     { mkT $ TOpCodeSimple 0x10 }
<0>[Ff][Cc][Mm][Pp][Ee]                 { mkT $ TOpCodeSimple 0x11 }
<0>[Ff][Uu][Nn][Ee]                     { mkT $ TOpCodeSimple 0x12 }
<0>[Ff][Ee][Qq][Ll][Ee]                 { mkT $ TOpCodeSimple 0x13 }
<0>[Ff][Dd][Ii][Vv]                     { mkT $ TOpCodeSimple 0x14 }
<0>[Ff][Ss][Qq][Rr][Tt]                 { mkT $ TOpCodeSimple 0x15 }
<0>[Ff][Rr][Ee][Mm]                     { mkT $ TOpCodeSimple 0x16 }
<0>[Ff][Ii][Nn][Tt]                     { mkT $ TOpCodeSimple 0x17 }
<0>[Mm][Uu][Ll]                         { mkT $ TOpCode 0x18 }
<0>[Mm][Uu][Ll][Uu]                     { mkT $ TOpCode 0x1A }
<0>[Dd][Ii][Vv]                         { mkT $ TOpCode 0x1C }
<0>[Dd][Ii][Vv][Uu]                     { mkT $ TOpCode 0x1E }
<0>[Aa][Dd][Dd]                         { mkT $ TOpCode 0x20 }
<0>[Ll][Dd][Aa]                         { mkT $ TOpCode 0x22 }
<0>[Aa][Dd][Dd][Uu]                     { mkT $ TOpCode 0x22 }
<0>[Ss][Uu][Bb]                         { mkT $ TOpCode 0x24 }
<0>[Ss][Uu][Bb][Uu]                     { mkT $ TOpCode 0x26 }
<0>2[Aa][Dd][Dd][Uu]                    { mkT $ TOpCode 0x28 }
<0>4[Aa][Dd][Dd][Uu]                    { mkT $ TOpCode 0x2A }
<0>8[Aa][Dd][Dd][Uu]                    { mkT $ TOpCode 0x2C }
<0>16[Aa][Dd][Dd][Uu]                   { mkT $ TOpCode 0x2E }
<0>[Cc][Mm][Pp]                         { mkT $ TOpCode 0x30 }
<0>[Cc][Mm][Pp][Uu]                     { mkT $ TOpCode 0x32 }
<0>[Nn][Ee][Gg]                         { mkT $ TOpCode 0x34 }
<0>[Nn][Ee][Gg][Uu]                     { mkT $ TOpCode 0x36 }
<0>[Ss][Ll]                             { mkT $ TOpCode 0x38 }
<0>[Ss][Ll][Uu]                         { mkT $ TOpCode 0x3A }
<0>[Ss][Rr]                             { mkT $ TOpCode 0x3C }
<0>[Ss][Rr][Uu]                         { mkT $ TOpCode 0x3E }
<0>[Bb][Nn]                             { mkT $ TOpCode 0x40 }
<0>[Bb][Zz]                             { mkT $ TOpCode 0x42 }
<0>[Bb][Pp]                             { mkT $ TOpCode 0x44 }
<0>[Bb][Oo][Dd]                         { mkT $ TOpCode 0x46 }
<0>[Bb][Nn][Nn]                         { mkT $ TOpCode 0x48 }
<0>[Bb][Nn][Zz]                         { mkT $ TOpCode 0x4A }
<0>[Bb][Nn][Pp]                         { mkT $ TOpCode 0x4C }
<0>[Bb][Ee][Vv]                         { mkT $ TOpCode 0x4E }
<0>[Pp][Bb][Nn]                         { mkT $ TOpCode 0x50 }
<0>[Pp][Bb][Zz]                         { mkT $ TOpCode 0x52 }
<0>[Pp][Bb][Pp]                         { mkT $ TOpCode 0x54 }
<0>[Pp][Bb][Oo][Dd]                     { mkT $ TOpCode 0x56 }
<0>[Pp][Bb][Nn][Nn]                     { mkT $ TOpCode 0x58 }
<0>[Pp][Bb][Nn][Zz]                     { mkT $ TOpCode 0x5A }
<0>[Pp][Bb][Nn][Pp]                     { mkT $ TOpCode 0x5C }
<0>[Pp][Bb][Ee][Vv]                     { mkT $ TOpCode 0x5E }
<0>[Cc][Ss][Nn]                         { mkT $ TOpCode 0x60 }
<0>[Cc][Ss][Zz]                         { mkT $ TOpCode 0x62 }
<0>[Cc][Ss][Pp]                         { mkT $ TOpCode 0x64 }
<0>[Cc][Ss][Oo][Dd]                     { mkT $ TOpCode 0x66 }
<0>[Cc][Ss][Nn][Nn]                     { mkT $ TOpCode 0x68 }
<0>[Cc][Ss][Nn][Zz]                     { mkT $ TOpCode 0x6A }
<0>[Cc][Ss][Nn][Pp]                     { mkT $ TOpCode 0x6C }
<0>[Cc][Ss][Ee][Vv]                     { mkT $ TOpCode 0x6E }
<0>[Zz][Ss][Nn]                         { mkT $ TOpCode 0x70 }
<0>[Zz][Ss][Zz]                         { mkT $ TOpCode 0x72 }
<0>[Zz][Ss][Pp]                         { mkT $ TOpCode 0x74 }
<0>[Zz][Ss][Oo][Dd]                     { mkT $ TOpCode 0x76 }
<0>[Zz][Ss][Nn][Nn]                     { mkT $ TOpCode 0x78 }
<0>[Zz][Ss][Nn][Zz]                     { mkT $ TOpCode 0x7A }
<0>[Zz][Ss][Nn][Pp]                     { mkT $ TOpCode 0x7C }
<0>[Zz][Ss][Ee][Vv]                     { mkT $ TOpCode 0x7E }
<0>[Ll][Dd][Bb]                         { mkT $ TOpCode 0x80 }
<0>[Ll][Dd][Bb][Uu]                     { mkT $ TOpCode 0x82 }
<0>[Ll][Dd][Ww]                         { mkT $ TOpCode 0x84 }
<0>[Ll][Dd][Ww][Uu]                     { mkT $ TOpCode 0x86 }
<0>[Ll][Dd][Tt]                         { mkT $ TOpCode 0x88 }
<0>[Ll][Dd][Tt][Uu]                     { mkT $ TOpCode 0x8A }
<0>[Ll][Dd][Oo]                         { mkT $ TOpCode 0x8C }
<0>[Ll][Dd][Oo][Uu]                     { mkT $ TOpCode 0x8E }
<0>[Ll][Dd][Ss][Ff]                     { mkT $ TOpCode 0x90 }
<0>[Ll][Dd][Hh][Tt]                     { mkT $ TOpCode 0x92 }
```

```
<0>[Cc][Ss][Ww][Aa][Pp]                 { mkT $ TOpCode 0x94 }
<0>[Ll][Dd][Uu][Nn][Cc]                 { mkT $ TOpCode 0x96 }
<0>[Ll][Dd][Vv][Tt][Ss]                 { mkT $ TOpCode 0x98 }
<0>[Pp][Rr][Ee][Ll][Dd]                 { mkT $ TOpCode 0x9A }
<0>[Pp][Rr][Ee][Gg][Oo]                 { mkT $ TOpCode 0x9C }
<0>[Gg][Oo]                             { mkT $ TOpCode 0x9E }
<0>[Ss][Tt][Bb]                         { mkT $ TOpCode 0xA0 }
<0>[Ss][Tt][Bb][Uu]                     { mkT $ TOpCode 0xA2 }
<0>[Ss][Tt][Ww]                         { mkT $ TOpCode 0xA4 }
<0>[Ss][Tt][Ww][Uu]                     { mkT $ TOpCode 0xA6 }
<0>[Ss][Tt][Tt]                         { mkT $ TOpCode 0xA8 }
<0>[Ss][Tt][Tt][Uu]                     { mkT $ TOpCode 0xAA }
<0>[Ss][Tt][Oo]                         { mkT $ TOpCode 0xAC }
<0>[Ss][Tt][Oo][Uu]                     { mkT $ TOpCode 0xAE }
<0>[Ss][Tt][Ss][Ff]                     { mkT $ TOpCode 0xB0 }
<0>[Ss][Tt][Hh][Tt]                     { mkT $ TOpCode 0xB2 }
<0>[Ss][Tt][Cc][Oo]                     { mkT $ TOpCode 0xB4 }
<0>[Ss][Tt][Uu][Nn][Cc]                 { mkT $ TOpCode 0xB6 }
<0>[Ss][Yy][Nn][Cc][Dd]                 { mkT $ TOpCode 0xB8 }
<0>[Pp][Rr][Ee][Ss][Tt]                 { mkT $ TOpCode 0xBA }
<0>[Ss][Yy][Nn][Cc][Ii][Dd]             { mkT $ TOpCode 0xBC }
<0>[Pp][Uu][Ss][Hh][Gg][Oo]             { mkT $ TOpCode 0xBE }
<0>[Oo][Rr]                             { mkT $ TOpCode 0xC0 }
<0>[Oo][Rr][Nn]                         { mkT $ TOpCode 0xC2 }
<0>[Nn][Oo][Rr]                         { mkT $ TOpCode 0xC4 }
<0>[Xx][Oo][Rr]                         { mkT $ TOpCode 0xC6 }
<0>[Aa][Nn][Dd]                         { mkT $ TOpCode 0xC8 }
<0>[Aa][Nn][Dd][Nn]                     { mkT $ TOpCode 0xCA }
<0>[Nn][Aa][Nn][Dd]                     { mkT $ TOpCode 0xCC }
<0>[Nn][Xx][Oo][Rr]                     { mkT $ TOpCode 0xCE }
<0>[Bb][Dd][Ii][Ff]                     { mkT $ TOpCode 0xD0 }
<0>[Ww][Dd][Ii][Ff]                     { mkT $ TOpCode 0xD2 }
<0>[Tt][Dd][Ii][Ff]                     { mkT $ TOpCode 0xD4 }
<0>[Oo][Dd][Ii][Ff]                     { mkT $ TOpCode 0xD6 }
<0>[Mm][Uu][Xx]                         { mkT $ TOpCode 0xD8 }
<0>[Ss][Aa][Dd][Dd]                     { mkT $ TOpCode 0xDA }
<0>[Mm][Oo][Rr]                         { mkT $ TOpCode 0xDC }
<0>[Mm][Xx][Oo][Rr]                     { mkT $ TOpCode 0xDE }
<0>[Ss][Ee][Tt][Hh]                     { mkT $ TOpCodeSimple 0xE0 }
<0>[Ss][Ee][Tt][Mm][Hh]                 { mkT $ TOpCodeSimple 0xE1 }
<0>[Ss][Ee][Tt][Mm][Ll]                 { mkT $ TOpCodeSimple 0xE2 }
<0>[Ss][Ee][Tt][Ll]                     { mkT $ TOpCodeSimple 0xE3 }
<0>[Ii][Nn][Cc][Hh]                     { mkT $ TOpCodeSimple 0xE4 }
<0>[Ii][Nn][Cc][Mm][Hh]                 { mkT $ TOpCodeSimple 0xE5 }
<0>[Ii][Nn][Cc][Mm][Ll]                 { mkT $ TOpCodeSimple 0xE6 }
<0>[Ii][Nn][Cc][Ll]                     { mkT $ TOpCodeSimple 0xE7 }
<0>[Oo][Rr][Hh]                         { mkT $ TOpCodeSimple 0xE8 }
<0>[Oo][Rr][Mm][Hh]                     { mkT $ TOpCodeSimple 0xE9 }
<0>[Oo][Rr][Mm][Ll]                     { mkT $ TOpCodeSimple 0xEA }
<0>[Oo][Rr][Ll]                         { mkT $ TOpCodeSimple 0xEB }
<0>[Aa][Nn][Dd][Nn][Hh]                 { mkT $ TOpCodeSimple 0xEC }
<0>[Aa][Nn][Dd][Nn][Mm][Hh]             { mkT $ TOpCodeSimple 0xED }
<0>[Aa][Nn][Dd][Nn][Mm][Ll]             { mkT $ TOpCodeSimple 0xEE }
<0>[Aa][Nn][Dd][Nn][Ll]                 { mkT $ TOpCodeSimple 0xEF }
<0>[Jj][Mm][Pp]                         { mkT $ TOpCode 0xF0 }
<0>[Pp][Uu][Ss][Hh][Jj]                 { mkT $ TOpCodeSimple 0xF2 }
<0>[Gg][Ee][Tt][Aa]                     { mkT $ TOpCodeSimple 0xF4 }
<0>[Pp][Uu][Tt]                         { mkT $ TOpCode 0xF6 }
<0>[Pp][Oo][Pp]                         { mkT $ TOpCodeSimple 0xF8 }
<0>[Rr][Ee][Ss][Uu][Mm][Ee]             { mkT $ TOpCodeSimple 0xF9 }
<0>[Ss][Aa][Vv][Ee]                     { mkT $ TOpCodeSimple 0xFA }
<0>[Ss][Yy][Nn][Cc]                     { mkT $ TOpCodeSimple 0xFC }
<0>[Ss][Ww][Yy][Mm]                     { mkT $ TOpCodeSimple 0xFD }
<0>[Gg][Ee][Tt]                         { mkT $ TOpCodeSimple 0xFE }
<0>[Tt][Rr][Ii][Pp]                     { mkT $ TOpCodeSimple 0xFF }
<0>Fputs                                { mkT TFputS }
<0>StdOut                               { mkT TStdOut }
<0>Halt                                 { mkT THalt }
<0>$digit+                                  { mkInteger }
<0>\,                                   { mkT TComma }
<0>\"                                   { startString `andBegin` string }
<string>\\\"                            { addCharToString '\"' }
<string>\\\\                            { addCharToString '\\' }
<string>\"                              { endString `andBegin` state_initial }
<string>.                               { addCurrentToString }
<0>\'.\'                                { mkChar }
<0>$alpha [$alpha $digit \_ \']*        { mkIdentifier }
<0>.                                    { mkError }

{
data Token = LEOF
           | TIdentifier { tid_name :: String }
           | TError { terr_text :: String }
           | TInteger { tint_value :: Int }
```

```
            | THexLiteral { thex_value :: Int }
            | TRegister { treg_value :: Int }
            | TStringLiteral { tsl_text :: String }
            | TLocalForwardOperand { tlfo :: Int }
            | TLocalBackwardOperand { tlbo :: Int }
            | TLocalLabel { tll :: Int }
            | TIS
            | TByte
            | TGREG
            | TLOC
            | TWyde
            | TTetra
            | TOcta
            | TSet
            | TFputS
            | TStdOut
            | THalt
            | TOpCode { toc_value :: Int }
            | TOpCodeSimple { soc_value :: Int }
            | TTextSegment
            | TDataSegment
            | TPoolSegment
            | TStackSegment
            | TAtSign
            | TComma
            | TPlus
            | TMult
            | TMinus
            | TDivide
            | TOpenParen
            | TCloseParen
            | TByteLiteral Char
            | W String
            | CommentStart
            | CommentEnd
            | CommentBody String
            deriving (Show,Eq)

state_initial :: Int
state_initial = 0

data AlexUserState = AlexUserState
                   {
                     lexerStringState    :: Bool
                     , lexerStringValue  :: String
                   }

alexInitUserState :: AlexUserState
alexInitUserState = AlexUserState
                   {
                     lexerStringState    = False
                     , lexerStringValue  = ""
                   }

setLexerStringState :: Bool -> Alex ()
setLexerStringState ss = Alex $ \s -> Right (s{alex_ust=(alex_ust s){lexerStringState=ss}}, ()
     )

setLexerStringValue :: String -> Alex ()
setLexerStringValue ss = Alex $ \s -> Right (s{alex_ust=(alex_ust s){lexerStringValue=ss}}, ()
     )

getLexerStringValue :: Alex String
getLexerStringValue = Alex $ \s@AlexState{alex_ust=ust} -> Right (s, lexerStringValue ust)

addCharToLexerStringValue :: Char -> Alex ()
addCharToLexerStringValue c = Alex $ \s -> Right (s{alex_ust=(alex_ust s){lexerStringValue=c:
     lexerStringValue (alex_ust s)}}, ())

addCurrentToString :: (t, t1, t2, String) -> Int -> Alex Token
addCurrentToString input@(_, _, _, remaining) length =
    addCharToString c input length
    where
        c = if (length == 1)
            then head remaining
            else error "Invalid call to addCurrentString"

addCharToString :: Char -> t -> t1 -> Alex Token
addCharToString c _     _   =
    do
        addCharToLexerStringValue c
        alexMonadScan

word a@(_,c,_,inp) len = mkT (W (take len inp)) a len
```

```
extractValue :: Num a => [(a, String)] -> a
extractValue ((value, ""):_) = value
extractValue _ = error "Invalid Hex Value"

mkHex :: Monad m => (t, t1, t2, String) -> Int -> m Token
mkHex input length =
    mkT (THexLiteral decValue) input length
    where
        str = getStr input length
        hexPart = tail str
        decValue = extractValue $ readHex hexPart

mkLocalLabel input length = mkT (TLocalLabel val) input length
    where val = read (getStr input 1) :: Int

mkChar input length = mkT (TByteLiteral val) input length
    where val = (getStr input 2) !! 1 :: Char

mkLocalForwardOperand input length = mkT (TLocalForwardOperand val) input length
    where val = read (getStr input 1) :: Int

mkLocalBackwardOperand input length = mkT (TLocalBackwardOperand val) input length
    where val = read (getStr input 1) :: Int

mkInteger input length
    | val >= 0 && val < 256 = mkT (TByteLiteral (chr val)) input length
    | otherwise = mkT (TInteger val) input length
    where val = read (getStr input length) :: Int

mkRegister input length
    | val >=0 && val < 256 = mkT (TRegister val) input length
    | otherwise = mkT (TError ("Invalid Register " ++ registerText)) input length
    where
        registerText = getStr input length
        val = read (tail registerText) :: Int

mkIdentifier :: Monad m => (t, t1, t2, String) -> Int -> m Token
mkIdentifier input length =
    mkT (TIdentifier label) input length
    where label = getStr input length

mkError :: Monad m => (t, t1, t2, String) -> Int -> m Token
mkError input length =
    mkT (TError label) input length
    where label = getStr input length

getStr (_, _, _, remaining) length = take length remaining

mkT :: (Monad m) => Token -> t -> t1 -> m Token
mkT token _ _ = return $ token

alexEOF = return LEOF

startString _ _ =
    do
        setLexerStringValue ""
        setLexerStringState True
        alexMonadScan

endString input length =
    do
        s <- getLexerStringValue
        setLexerStringState False
        mkT (TStringLiteral (reverse s)) input length

tokens str = runAlex str $ do
               let loop = do tok <- alexMonadScan
                             if tok == LEOF
                               then return [ LEOF ]
                               else do toks <- loop
                                       return $ tok : toks
               loop
}
```

B.1.2 Parser

```
{
module MMix_Parser where

import Data.Char
import MMix_Lexer
```

```
}

%name parseFile
%tokentype { Token }
%error { parseError }
%monad { Alex }
%lexer { lexwrap } { LEOF }

%token
    OP_CODE         { TOpCode $$ }
    OP_CODE_SIMPLE  { TOpCodeSimple $$ }
    SET             { TSet }
    COMMA           { TComma }
    HALT            { THalt }
    FPUTS           { TFputS }
    STDOUT          { TStdOut }
    BYTE_LIT        { TByteLiteral $$ }
    ID              { TIdentifier $$ }
    REG             { TRegister $$ }
    INT             { TInteger $$ }
    LOCAL_LABEL     { TLocalLabel $$ }
    FORWARD         { TLocalForwardOperand $$ }
    BACKWARD        { TLocalBackwardOperand $$ }
    LOC             { TLOC }
    IS              { TIS }
    WYDE            { TWyde }
    TETRA           { TTetra }
    OCTA            { TOcta }
    GREG            { TGREG }
    PLUS            { TPlus }
    MINUS           { TMinus }
    MULTIPLY        { TMult }
    DIVIDE          { TDivide }
    AT              { TAtSign }
    TS              { TTextSegment }
    DS              { TDataSegment }
    PS              { TPoolSegment }
    SS              { TStackSegment }
    BYTE            { TByte }
    STR             { TStringLiteral $$ }
    HEX             { THexLiteral $$ }
    OPEN            { TOpenParen }
    CLOSE           { TCloseParen }
%%

Program         : AssignmentLines { reverse $1 }

AssignmentLines : {- empty -}        {[]}
                | AssignmentLines AssignmentLine { $2 : $1 }

AssignmentLine :: {Line}
AssingmentLine : OP_CODE OperatorList { defaultPlainOpCodeLine { pocl_code = $1, pocl_ops = (
     reverse $2) } }
               | Identifier PI { defaultLabelledPILine { lppl_id = $2, lppl_ident = $1 } }
               | Identifier OP_CODE OperatorList { defaultLabelledOpCodeLine { lpocl_code = $2
                   , lpocl_ops = (reverse $3), lpocl_ident = $1 }  }
               | PI { defaultPlainPILine { ppl_id = $1 } }
               | OP_CODE_SIMPLE OperatorList { defaultPlainOpCodeLine { pocl_code = $1,
                   pocl_ops = (reverse $2), pocl_sim = True } }
               | Identifier OP_CODE_SIMPLE OperatorList { defaultLabelledOpCodeLine {
                   lpocl_code = $2, lpocl_ops = (reverse $3), lpocl_ident = $1, lpocl_sim =
                   True }  }

OperatorList : OperatorElement { $1 : [] }
    | OperatorList COMMA OperatorElement { $3 : $1 }

OperatorElement : HALT       { PseudoCode 0 }
                | FPUTS      { fputs }
                | STDOUT     { PseudoCode 1 }
                | REG        { Register (chr $1) }
                | FORWARD    { LocalForward $1 }
                | BACKWARD   { LocalBackward $1 }
                | Expression { Expr $1 }

Identifier : ID { Id $1 }
           | LOCAL_LABEL { LocalLabel $1 }

PI : LOC Expression      { LocEx $2 }
   | GREG Expression     { GregEx $2 }
   | SET OperatorElement COMMA OperatorElement { Set ($2, $4) }
   | BYTE Byte_Array     { ByteArray (reverse $2) }
   | WYDE Byte_Array     { WydeArray (reverse $2) }
   | TETRA Byte_Array    { TetraArray (reverse $2) }
   | OCTA Byte_Array     { OctaArray (reverse $2) }
```

```
   | IS INT              { IsNumber $2 }
   | IS BYTE_LIT         { IsNumber (ord $2) }
   | IS REG              { IsRegister $2 }
   | IS Identifier       { IsIdentifier $2 }

Byte_Array : STR { reverse $1 }
           | HEX { (chr $1) : [] }
           | BYTE_LIT { $1 : [] }
           | Byte_Array COMMA STR { (reverse $3) ++ $1 }
           | Byte_Array COMMA BYTE_LIT { $3 : $1 }
           | Byte_Array COMMA HEX { (chr $3) : $1 }

GlobalVariables : TS { 0x0000000000000000 }
                | DS { 0x2000000000000000 }
                | PS { 0x4000000000000000 }
                | SS { 0x6000000000000000 }

Expression : Term { $1 }
           | Expression PLUS Term { ExpressionPlus $1 $3 }
           | Expression MINUS Term { ExpressionMinus $1 $3 }

Term : Primary_Expression { $1 }
     | Term MULTIPLY Primary_Expression { ExpressionMultiply $1 $3 }
     | Term DIVIDE Primary_Expression { ExpressionDivide $1 $3 }

Primary_Expression : INT                    { ExpressionNumber $1 }
                   | Identifier             { ExpressionIdentifier $1 }
                   | BYTE_LIT               { ExpressionNumber (ord $1) }
                   | AT                     { ExpressionAT }
                   | HEX                    { ExpressionNumber $1 }
                   | GlobalVariables        { ExpressionNumber $1 }
                   | OPEN Expression CLOSE { $2 }


{
data Line = PlainOpCodeLine { pocl_code :: Int, pocl_ops :: [OperatorElement], pocl_loc :: Int
     , pocl_sim :: Bool }
          | LabelledOpCodeLine { lpocl_code :: Int, lpocl_ops :: [OperatorElement],
              lpocl_ident :: Identifier, lpocl_loc :: Int, lpocl_sim :: Bool }
          | PlainPILine { ppl_id :: PseudoInstruction, ppl_loc :: Int }
          | LabelledPILine { lppl_id :: PseudoInstruction, lppl_ident :: Identifier, lppl_loc
              :: Int }
          deriving (Eq, Show)

data Identifier = Id String
                | LocalLabel Int
                deriving (Eq, Show, Ord)

data OperatorElement = ByteLiteral Char
               | PseudoCode Int
               | Register Char
               | Ident Identifier
               | LocalForward Int
               | LocalBackward Int
               | Expr ExpressionEntry
               deriving (Eq, Show)

data ExpressionEntry = ExpressionNumber Int
                       | ExpressionRegister Char ExpressionEntry
                       | ExpressionIdentifier Identifier
                       | ExpressionGV Int
                       | Expression
                       | ExpressionAT
                       | ExpressionPlus ExpressionEntry ExpressionEntry
                       | ExpressionMinus ExpressionEntry ExpressionEntry
                       | ExpressionMultiply ExpressionEntry ExpressionEntry
                       | ExpressionDivide ExpressionEntry ExpressionEntry
                       | ExpressionOpen
                       | ExpressionClose
                       deriving (Eq, Show)

data PseudoInstruction = LOC Int
                       | LocEx ExpressionEntry
                       | GregAuto
                       | GregSpecific Char
                       | GregEx ExpressionEntry
                       | ByteArray [Char]
                       | WydeArray [Char]
                       | TetraArray [Char]
                       | OctaArray [Char]
                       | IsRegister Int
                       | IsNumber Int
                       | IsIdentifier Identifier
                       | Set (OperatorElement, OperatorElement)
```

```
                       deriving (Eq, Show)

-- fullParse "/home/steveedmans/test.mms"
-- fullParse "/home/steveedmans/hail.mms"

defaultPlainOpCodeLine = PlainOpCodeLine { pocl_loc = -1, pocl_sim = False }
defaultLabelledOpCodeLine = LabelledOpCodeLine { lpocl_loc = -1, lpocl_sim = False }
defaultPlainPILine = PlainPILine { ppl_loc = -1 }
defaultLabelledPILine = LabelledPILine { lppl_loc = -1 }

parseError m = alexError $ "WHY! " ++ show m

lexwrap :: (Token -> Alex a) -> Alex a
lexwrap cont = do
    token <- alexMonadScan
    cont token

fputs = PseudoCode 7

fullParse path = do
    contents <- readFile path
    print $ parseStr contents

parseStr str = runAlex str parseFile

}
```

B.1.3 Symbol Table

```
module SymbolTable where

import MMix_Parser
import Registers
import qualified Data.Map.Lazy as M
import Data.Char (chr, ord)
import Text.Regex.Posix
import DataTypes

type Table = M.Map String Int
type BaseTable = M.Map Char Int
type RegisterOffset = (Char, Int)
type CounterMap = M.Map Int Int

createSymbolTable :: Either String [Line] -> Either String SymbolTable
createSymbolTable (Left msg) = Left msg
createSymbolTable (Right lines) =
        let symbols = foldl getSymbol (Right M.empty) lines
            regs = createRegisterTable $ Right lines
        in symbols

getSymbol :: Either String SymbolTable -> Line -> Either String SymbolTable
getSymbol (Left errorMsg) _ = Left errorMsg
getSymbol (Right table) (LabelledPILine pi@(GregAuto) ident address)
          | M.member ident table = Left $ "Identifier already present " ++ (show ident)
          | otherwise = Right $ M.insert ident (address, Just pi) table
getSymbol (Right table) (LabelledPILine pi@(GregSpecific _) ident address)
          | M.member ident table = Left $ "Identifier already present " ++ (show ident)
          | otherwise = Right $ M.insert ident (address, Just pi) table
getSymbol (Right table) (LabelledPILine pi@(GregEx (ExpressionRegister reg _)) ident address)
          | M.member ident table = Left $ "Identifier already present " ++ (show ident)
          | otherwise = Right $ M.insert ident (address, Just (IsRegister (ord reg))) table
getSymbol (Right table) (LabelledPILine val@(IsNumber _) ident address)
          | M.member ident table = Left $ "Identifier already present " ++ (show ident)
          | otherwise = Right $ M.insert ident (address, Just val) table
getSymbol (Right table) (LabelledPILine val@(IsRegister _) ident address)
          | M.member ident table = Left $ "Identifier already present " ++ (show ident)
          | otherwise = Right $ M.insert ident (address, Just val) table
getSymbol (Right table) (LabelledPILine val@(IsIdentifier _) ident address)
          | M.member ident table = Left $ "Identifier already present " ++ (show ident)
          | otherwise = Right $ M.insert ident (address, Just val) table
getSymbol (Right table) (LabelledPILine val ident address)
          | M.member ident table = Left $ "Identifier already present " ++ (show ident)
          | otherwise = Right $ M.insert ident (address, Just(val)) table
getSymbol (Right table) (LabelledOpCodeLine _ _ ident address _)
          | M.member ident table = Left $ "Identifier already present " ++ (show ident)
          | otherwise = Right $ M.insert ident (address, Nothing) table
getSymbol (Right table) _ = Right $ table

getRegisterFromSymbol :: SymbolTable -> Identifier -> Int
getRegisterFromSymbol st id = reg
    where reg = case M.lookup id st of
                     Just(_, Just (IsRegister r)) -> r
```

```haskell
                    _                              -> -1

determineBaseAddressAndOffset :: (M.Map ExpressionEntry Char) -> RegisterAddress -> Maybe(
     RegisterOffset)
determineBaseAddressAndOffset rfa (required_address, _) =
  case (M.lookupLE (ExpressionNumber required_address) rfa) of
    Just((ExpressionNumber address), register) -> Just(register, offset)
      where offset = required_address - address
    _ -> Nothing

mapSymbolToAddress :: SymbolTable -> RegisterTable -> Identifier -> Maybe(RegisterOffset)
mapSymbolToAddress symbols registers identifier@(Id _)
    | M.member identifier symbols = result
    | otherwise = Just('b', 2)
      where registersByAddress = registersFromAddresses registers
            requiredAddress = symbols M.! identifier
            exactRegister   = extractRegister requiredAddress
            result          = case exactRegister of
                                Just(reg) -> Just(reg, 0)
                                _         -> determineBaseAddressAndOffset registersByAddress
                                      requiredAddress
mapSymbolToAddress _ _ _ = Nothing

extractRegister :: RegisterAddress -> Maybe(Char)
extractRegister (_, Just(IsRegister reg)) = Just(chr reg)
extractRegister _ = Nothing

getSymbolAddress :: SymbolTable -> Identifier -> Int
getSymbolAddress symbols identifier = add
    where Just(add, _) = M.lookup identifier symbols

update_counter :: Int -> Maybe Int -> Int -> CounterMap -> CounterMap
update_counter label (Just old_counter) adjustment counters = M.insert label new_counter
     counters
    where new_counter = old_counter + adjustment
update_counter _ _ _ counters = counters

updated_label :: Int -> Maybe Int -> Identifier
updated_label label (Just current_counter)  = Id $ system_symbol label current_counter
updated_label label _  = Id $ "??" ++ (show label) ++ "HMissing"

transformLocalSymbolLabel :: CounterMap -> Line -> (CounterMap, Line)
transformLocalSymbolLabel counters ln@(LabelledOpCodeLine _ _ (LocalLabel label) _ _) = (
     new_counters, ln{lpocl_ident=new_label})
    where current_counter = M.lookup label counters
          new_label = updated_label label current_counter
          new_counters = update_counter label current_counter 1 counters
transformLocalSymbolLabel counters ln@(LabelledPILine _ (LocalLabel label) _) = (new_counters,
      ln{lppl_ident=new_label})
    where current_counter = M.lookup label counters
          new_label = updated_label label current_counter
          new_counters = update_counter label current_counter 1 counters
transformLocalSymbolLabel counter line = (counter, line)

setLocalSymbolLabel :: CounterMap -> [Line] -> [Line] -> [Line]
setLocalSymbolLabel _ acc [] = reverse acc
setLocalSymbolLabel current_counters acc (x:xs) =
    let (new_counters, new_line) = transformLocalSymbolLabel current_counters x
        new_acc = new_line : acc
    in setLocalSymbolLabel new_counters new_acc xs

setLocalSymbolLabelAuto :: Either String [Line] -> Either String [Line]
setLocalSymbolLabelAuto (Right lns) = Right $ operands_set
    where labels_set = setLocalSymbolLabel localSymbolCounterMap [] lns
          operands_set = transformLocalSymbolLines initialForwardSymbolMap
               initialBackwardSymbolMap labels_set []
setLocalSymbolLabelAuto msg = msg

localSymbolCounterMap :: CounterMap
localSymbolCounterMap = M.fromList $ map (\x -> (x, 0)) [0..9]

initialForwardSymbolMap :: M.Map Int Identifier
initialForwardSymbolMap = M.fromList $ map (\x -> (x, (Id (system_symbol x 0)))) [0..9]

initialBackwardSymbolMap :: M.Map Int (Maybe Identifier)
initialBackwardSymbolMap = M.fromList $ map (\x -> (x, Nothing)) [0..9]

transformLocalSymbolLines :: M.Map Int Identifier -> M.Map Int (Maybe Identifier) -> [Line] ->
      [Line] -> [Line]
transformLocalSymbolLines _ _ [] acc = reverse acc
transformLocalSymbolLines f b (x:xs) acc = transformLocalSymbolLines f' b' xs new_acc
    where (f', b', new_line) = transformLocalSymbol f b x
          new_acc = new_line : acc
```

```
transformLocalSymbol :: M.Map Int Identifier -> M.Map Int (Maybe Identifier) -> Line -> (M.Map
      Int Identifier, M.Map Int (Maybe Identifier), Line)
transformLocalSymbol f b l = (f', b', l')
    where f' = transformForward f l
          b' = transformBackward b l
          l' = transformLocalSymbolLine f b l

transformLocalSymbolLine :: M.Map Int Identifier -> M.Map Int (Maybe Identifier) -> Line ->
     Line
transformLocalSymbolLine f b ln@(PlainOpCodeLine _ elements _ _) = ln{pocl_ops = new_elements}
    where new_elements = transformLocalSymbolElements f b elements []
transformLocalSymbolLine f b ln@(LabelledOpCodeLine _ elements _ _ _) = ln{lpocl_ops =
     new_elements}
    where new_elements = transformLocalSymbolElements f b elements []
transformLocalSymbolLine _ _ ln = ln

transformForward :: M.Map Int Identifier -> Line -> M.Map Int Identifier
transformForward f ln@(LabelledOpCodeLine _ _ (Id label) _ _)
    | is_system_id label = M.insert l new_id f
    | otherwise          = f
       where Just(l, c) = system_id label
             new_label  = system_symbol l (c + 1)
             new_id     = Id new_label
transformForward f ln@(LabelledPILine _ (Id label) _)
    | is_system_id label = M.insert l new_id f
    | otherwise          = f
       where Just(l, c) = system_id label
             new_label  = system_symbol l (c + 1)
             new_id     = Id new_label
transformForward f _ = f


transformBackward :: M.Map Int (Maybe Identifier) -> Line -> M.Map Int (Maybe Identifier)
transformBackward b ln@(LabelledOpCodeLine _ _ (Id label) _ _)
    | is_system_id label = M.insert l new_id b
    | otherwise          = b
       where Just(l, _) = system_id label
             new_id     = Just(Id label)
transformBackward b ln@(LabelledPILine _ (Id label) _)
    | is_system_id label = M.insert l new_id b
    | otherwise          = b
       where Just(l, _) = system_id label
             new_id     = Just(Id label)
transformBackward b l = b

transformLocalSymbolElements :: M.Map Int Identifier -> M.Map Int (Maybe Identifier) -> [
     OperatorElement] -> [OperatorElement] -> [OperatorElement]
transformLocalSymbolElements _ _ [] acc = reverse acc
transformLocalSymbolElements f b (x:xs) acc = transformLocalSymbolElements f b xs new_acc
    where new_value = transformLocalSymbolElement f b x
          new_acc = new_value : acc

transformLocalSymbolElement :: M.Map Int Identifier -> M.Map Int (Maybe Identifier) ->
     OperatorElement -> OperatorElement
transformLocalSymbolElement forwards _ (LocalForward label) = Ident (identifier)
    where Just identifier = M.lookup label forwards
transformLocalSymbolElement _ backwards elem@(LocalBackward label) = v
    where i = M.lookup label backwards
          v = extractWithDefault i elem
transformLocalSymbolElement _ _ element = element

extractWithDefault :: Maybe (Maybe Identifier) -> OperatorElement -> OperatorElement
extractWithDefault (Just (Just v)) _ = Ident v
extractWithDefault _ d = d

system_id :: String -> Maybe (Int, Int)
system_id label
    | is_system_id label = Just(l, c)
    | otherwise = Nothing
        where l = read $ drop 2 $ take 3 label
              c = read $ drop 4 label

system_symbol_pattern = "^\\?\\?[0-9]H[0-9]+$"

is_system_id :: String -> Bool
is_system_id symbol = symbol =~ system_symbol_pattern

system_symbol :: Int -> Int -> String
system_symbol label counter = "??" ++ (show label) ++ "H" ++ (show counter)
```

B.1.4 Common Data Types

```
module DataTypes where
import qualified Data.Map.Lazy as M
import MMix_Parser

type SymbolTable = M.Map Identifier RegisterAddress
type RegisterAddress = (Int, Maybe PseudoInstruction)

instance Ord ExpressionEntry where
    (ExpressionNumber num1) `compare` (ExpressionNumber num2) = num1 `compare` num2
```

B.1.5 Expressions

```
module Expressions where

import MMix_Parser
import DataTypes
import qualified Data.Map.Lazy as M

isSingleExprNumber :: ExpressionEntry -> Maybe Int
isSingleExprNumber (ExpressionNumber val) = Just val
isSingleExprNumber _ = Nothing

evaluateAllLocExpressions :: Either String [Line] -> Either String SymbolTable -> Either
     String [Line]
evaluateAllLocExpressions (Left msg) _ = Left msg
evaluateAllLocExpressions _ (Left msg) = Left msg
evaluateAllLocExpressions (Right lines) (Right st) = Right $ evaluateAllLocLines st lines []

evaluateAllLocLines :: SymbolTable -> [Line] -> [Line] -> [Line]
evaluateAllLocLines _ [] acc = reverse acc
evaluateAllLocLines st (ln:lns) acc = evaluateAllLocLines st lns (new_line : acc)
        where new_line = evaluateLocLine st ln

evaluateLocLine :: SymbolTable -> Line -> Line
evaluateLocLine st ln@(LabelledPILine (LocEx expr) _ address) = ln{lppl_id = (LocEx (
     ExpressionNumber v))}
   where v = evaluate expr address st
evaluateLocLine st ln@(PlainPILine (LocEx expr) address) = ln{ppl_id = (LocEx (
     ExpressionNumber v))}
   where v = evaluate expr address st
evaluateLocLine _ ln = ln

evaluateAllExpressions :: Either String [Line] -> Either String SymbolTable -> Either String [
     Line]
evaluateAllExpressions (Left msg) _ = Left msg
evaluateAllExpressions _ (Left msg) = Left msg
evaluateAllExpressions (Right lines) (Right st) = Right $ evaluateAllLines st lines []

evaluateAllLines :: SymbolTable -> [Line] -> [Line] -> [Line]
evaluateAllLines _ [] acc = reverse acc
evaluateAllLines st (ln:lns) acc = evaluateAllLines st lns (new_line : acc)
    where new_line = evaluateLine st ln

evaluateLine :: SymbolTable -> Line -> Line
evaluateLine st ln@(LabelledPILine (GregEx (ExpressionRegister reg expr)) _ address) =
     new_line
   where v = evaluate expr address st
         new_reg = ExpressionRegister reg (ExpressionNumber v)
         new_line = ln{lppl_id = (GregEx new_reg)}
evaluateLine st ln@(LabelledPILine (LocEx expr) _ address) = ln{lppl_id = (LocEx (
     ExpressionNumber v))}
   where v = evaluate expr address st
evaluateLine st ln@(PlainPILine (LocEx expr) address) = ln{ppl_id = (LocEx (ExpressionNumber v
     ))}
   where v = evaluate expr address st
evaluateLine st ln@(PlainOpCodeLine _ ops _ _) = ln{pocl_ops = updated_operands}
   where updated_operands = evaluateOperands st [] ops
evaluateLine st ln@(LabelledOpCodeLine _ ops _ _ _) = ln{lpocl_ops = updated_operands}
   where updated_operands = evaluateOperands st [] ops
evaluateLine _ ln = ln

evaluateOperands :: SymbolTable -> [OperatorElement] -> [OperatorElement] -> [OperatorElement]
evaluateOperands st acc [] = reverse acc
evaluateOperands st acc (op:ops) = evaluateOperands st (new_op : acc) ops
    where new_op = evaluateOperand st op

evaluateOperand :: SymbolTable -> OperatorElement -> OperatorElement
evaluateOperand _ op@(Expr (ExpressionNumber _)) = op
evaluateOperand _ op@(Expr (ExpressionRegister _ _)) = op
evaluateOperand _ op@(Expr (ExpressionIdentifier _)) = op
evaluateOperand _ op@(Expr (ExpressionGV _)) = op
evaluateOperand _ op@(Expr ExpressionAT) = op
```

```
evaluateOperand st (Expr expr) = Expr (ExpressionNumber val)
    where val = evaluate expr 0 st
evaluateOperand _ op = op

evaluate :: ExpressionEntry -> Int -> SymbolTable -> Int
evaluate (ExpressionNumber val) _ _ = val
evaluate ExpressionAT loc _ = loc
evaluate (ExpressionMinus expr1 expr2) loc st = v1 - v2
    where v1 = evaluate expr1 loc st
          v2 = evaluate expr2 loc st
evaluate (ExpressionPlus expr1 expr2) loc st = v1 + v2
    where v1 = evaluate expr1 loc st
          v2 = evaluate expr2 loc st
evaluate (ExpressionMultiply expr1 expr2) loc st = v1 * v2
    where v1 = evaluate expr1 loc st
          v2 = evaluate expr2 loc st
evaluate (ExpressionDivide expr1 expr2) loc st = quot v1 v2
    where v1 = evaluate expr1 loc st
          v2 = evaluate expr2 loc st
evaluate (ExpressionIdentifier id) _ st
    | M.member id st = v
        where Just(val, lv) = M.lookup id st
              v = evaluatePI lv val
evaluate _ _ _ = -999999

evaluatePI :: Maybe PseudoInstruction -> Int -> Int
evaluatePI (Just (IsNumber val)) _ = val
evaluatePI _ val = val
```

B.1.6 Locations

```
module Locations where

import MMix_Parser
import Expressions

setInnerLocation nextLoc acc [] = reverse acc
setInnerLocation nextLoc acc (ln:lns) = setInnerLocation newLoc newAcc lns
    where (newLoc, newLine) = setLocation nextLoc ln
          newAcc = newLine : acc

setLocation :: Int -> Line -> (Int, Line)
setLocation nextLoc ln@(PlainPILine (LocEx loc) _) =
    case isSingleExprNumber loc of
       Just val -> (val, ln { ppl_loc = val })
       _ -> (nextLoc, ln)
setLocation nextLoc ln@(LabelledPILine (LocEx loc) _ _) =
    case isSingleExprNumber loc of
       Just val -> (val, ln { lppl_loc = val })
       _ -> (nextLoc, ln)
setLocation nextLoc ln@(LabelledPILine (ByteArray arr) _ _) = (newLoc, ln { lppl_loc = nextLoc
       })
    where size = length arr
          adjustment = case (rem size 4) of
                          0 -> 0
                          x -> 4 - x
          newLoc = nextLoc + size
setLocation nextLoc ln@(PlainPILine (ByteArray arr) _) = (newLoc, ln { ppl_loc = nextLoc })
    where size = length arr
          adjustment = case (rem size 4) of
                          0 -> 0
                          x -> 4 - x
          newLoc = nextLoc + size
setLocation nextLoc ln@(LabelledPILine (WydeArray arr) _ _) = (newLoc, ln { lppl_loc =
      addjusted_loc })
    where addjusted_loc = case (rem nextLoc 2) of
                             0 -> nextLoc
                             x -> nextLoc + x
          newLoc = addjusted_loc + ((length arr) * 2)
setLocation nextLoc ln@(PlainPILine (WydeArray arr) _) = (newLoc, ln { ppl_loc = addjusted_loc
       })
    where addjusted_loc = case (rem nextLoc 2) of
                             0 -> nextLoc
                             x -> nextLoc + x
          newLoc = addjusted_loc + ((length arr) * 2)
setLocation nextLoc ln@(LabelledPILine (TetraArray arr) _ _) = (newLoc, ln { lppl_loc =
      addjusted_loc })
    where addjusted_loc = case (rem nextLoc 4) of
                             0 -> nextLoc
                             x -> nextLoc + (4 - x)
          newLoc = addjusted_loc + ((length arr) * 4)
```

```
setLocation nextLoc ln@(PlainPILine (TetraArray arr) _) = (newLoc, ln { ppl_loc =
     addjusted_loc })
    where addjusted_loc = case (rem nextLoc 4) of
                            0 -> nextLoc
                            x -> nextLoc + (4 - x)
          newLoc = addjusted_loc + ((length arr) * 4)
setLocation nextLoc ln@(LabelledPILine (OctaArray arr) _ _) = (newLoc, ln { lppl_loc =
     addjusted_loc })
    where addjusted_loc = case (rem nextLoc 8) of
                            0 -> nextLoc
                            x -> nextLoc + (8 - x)
          newLoc = addjusted_loc + ((length arr) * 8)
setLocation nextLoc ln@(PlainPILine (OctaArray arr) _) = (newLoc, ln { ppl_loc = addjusted_loc
      })
    where addjusted_loc = case (rem nextLoc 8) of
                            0 -> nextLoc
                            x -> nextLoc + (8 - x)
          newLoc = addjusted_loc + ((length arr) * 8)
setLocation nextLoc ln@(LabelledPILine (Set _) _ _) = (newLoc, ln { lppl_loc = nextLoc })
    where newLoc = nextLoc + 4
setLocation nextLoc ln@(PlainPILine (Set _) _) = (newLoc, ln { ppl_loc = nextLoc })
    where newLoc = nextLoc + 4
setLocation nextLoc ln@(PlainPILine _ _) = (nextLoc, ln { ppl_loc = nextLoc })
setLocation nextLoc ln@(LabelledPILine _ _ _) = (nextLoc, ln { lppl_loc = nextLoc })
setLocation nextLoc ln@(PlainOpCodeLine _ _ _ _) = (newLoc, ln { pocl_loc = adjusted_loc })
    where adjusted_loc = case (rem nextLoc 4) of
                           0 -> nextLoc
                           x -> nextLoc + (4 - x)
          newLoc = adjusted_loc + 4
setLocation nextLoc ln@(LabelledOpCodeLine _ _ _ _ _) = (newLoc, ln { lpocl_loc = adjusted_loc
      })
    where adjusted_loc = case (rem nextLoc 4) of
                           0 -> nextLoc
                           x -> nextLoc + (4 - x)
          newLoc = adjusted_loc + 4
```

B.1.7 Registers

```
module Registers
--(
--    RegisterAddress,
--    RegisterTable,
--    createRegisterTable
--)
where

import MMix_Parser
import qualified Data.Map.Lazy as M
import Data.Char (chr, ord)
import Expressions
import DataTypes

type RegisterTable = M.Map Char ExpressionEntry
type AlternativeRegisterTable = M.Map ExpressionEntry Char

setAlexGregAuto :: Either String [Line] -> Either String [Line]
setAlexGregAuto (Right lns) = Right $ setGregAuto 254 [] lns
setAlexGregAuto msg = msg

setGregAuto :: Int -> [Line] -> [Line] -> [Line]
setGregAuto _ acc [] = reverse acc
setGregAuto currentRegister acc (x:xs) =
    let (newLine, nextRegister) = specifyGregAuto x currentRegister
        newAcc = newLine : acc
    in setGregAuto nextRegister newAcc xs

specifyGregAuto :: Line -> Int -> (Line, Int)
specifyGregAuto ln@(LabelledPILine (GregEx val) _ loc) nxt = (new_line, new_counter)
   where new_line = ln{lppl_id = GregEx (ExpressionRegister (chr nxt) val)}
         new_counter = nxt - 1
specifyGregAuto ln@(PlainPILine (GregEx val) loc) nxt =  (new_line, new_counter)
   where new_line = ln{ppl_id = GregEx (ExpressionRegister (chr nxt) val)}
         new_counter = nxt - 1
specifyGregAuto line nxt = (line, nxt)


createRegisterTable :: Either String [Line] -> Either String RegisterTable
createRegisterTable (Left msg) = Left msg
createRegisterTable (Right lines) = foldl getRegister (Right M.empty) lines

getRegister :: Either String RegisterTable -> Line -> Either String RegisterTable
getRegister (Left msg) _ = Left msg
```

```
getRegister (Right table) (LabelledOpCodeLine _ _ (Id "Main") address _)
        | M.member (chr 255) table = Left $ "Duplicate Main section definition"
        | otherwise = Right $ M.insert (chr 255) (ExpressionNumber address) table
getRegister (Right table) (LabelledPILine _ (Id "Main") address)
        | M.member (chr 255) table = Left $ "Duplicate Main section definition"
        | otherwise = Right $ M.insert (chr 255) (ExpressionNumber address) table
getRegister (Right table) (PlainPILine pi@(GregEx (ExpressionRegister r ExpressionAT)) address
     ) =
        addRegister table r (ExpressionNumber address)
getRegister (Right table) (LabelledPILine pi@(GregEx (ExpressionRegister r ExpressionAT)) _
     address) =
        addRegister table r (ExpressionNumber address)
getRegister (Right table) (LabelledPILine pi@(GregEx (ExpressionRegister r (ExpressionNumber v
     ))) _ address) =
        addRegister table r (ExpressionNumber v)
getRegister (Right table) _ = Right $ table

addRegister table register address
    | M.member register table = Left $ "Duplicate register definition " ++ (show register)
    | otherwise = Right $ M.insert register address table

registersFromAddresses :: RegisterTable -> AlternativeRegisterTable
registersFromAddresses orig = M.foldrWithKey addNextRegister M.empty without_main
   where without_main = M.filterWithKey remove_main orig

remove_main :: Char -> ExpressionEntry -> Bool
remove_main k _
  | k == (chr 255) = False
  | otherwise      = True

addNextRegister :: Char -> ExpressionEntry -> AlternativeRegisterTable ->
     AlternativeRegisterTable
addNextRegister k v orig = M.insert v k orig

getRegisterDetails :: [ExpressionEntry] -> Maybe(Char, Int)
getRegisterDetails _ = Nothing
```

B.1.8 Code Generation

```
module CodeGen where

import SymbolTable
import qualified Data.Map.Lazy as M
import qualified Data.List.Ordered as O
import qualified Data.ByteString.Lazy as B
import Data.Binary
import MMix_Parser
import Data.Char (chr, ord)
import Registers
import Expressions as E
import Numeric (showHex)
import DataTypes

type AdjustedOperands = (Int, String)

data CodeLine = CodeLine { cl_address :: Int, cl_size :: Int, cl_code :: [Char] }
                deriving(Show)

instance Eq CodeLine where
    (CodeLine address1 _ _) == (CodeLine address2 _ _) = address1 == address2
instance Ord CodeLine where
    (CodeLine address1 _ _) `compare` (CodeLine address2 _ _) = address1 `compare` address2

encodeProgram :: Either String [CodeLine] -> Either String RegisterTable -> Either String
     String
encodeProgram (Left code_error) _ = Left code_error
encodeProgram _ (Left register_error) = Left register_error
encodeProgram (Right code) (Right regs) = Right $ map chr $ encodeProgramInt code regs

genCodeForLine :: SymbolTable -> RegisterTable -> Line -> Maybe(CodeLine)
genCodeForLine symbols registers (LabelledOpCodeLine opcode operands _ address simple_code) =
     genOpCodeOutput symbols registers opcode operands address simple_code
genCodeForLine symbols registers (PlainOpCodeLine opcode operands address simple_code) =
     genOpCodeOutput symbols registers opcode operands address simple_code
genCodeForLine _ _ (LabelledPILine (ByteArray arr) _ address) = Just(CodeLine {cl_address =
     address, cl_size = s, cl_code = arr})
    where s = length arr
genCodeForLine _ _ (LabelledPILine (WydeArray arr) _ address) = Just(CodeLine {cl_address =
     address, cl_size = s, cl_code = wyde_array})
    where wyde_array = make_bytes arr 2
          s = length wyde_array
```

```
genCodeForLine _ _ (LabelledPILine (TetraArray arr) _ address) = Just(CodeLine {cl_address =
     address, cl_size = s, cl_code = wyde_array})
    where wyde_array = make_bytes arr 4
          s = length wyde_array
genCodeForLine _ _ (LabelledPILine (OctaArray arr) _ address) = Just(CodeLine {cl_address =
     address, cl_size = s, cl_code = wyde_array})
    where wyde_array = make_bytes arr 8
          s = length wyde_array
genCodeForLine symbols registers (LabelledPILine (Set (e1, e2)) _ address) =
    genPICodeOutput symbols registers address e1 e2
genCodeForLine symbols registers (PlainPILine (Set (e1, e2)) address) =
    genPICodeOutput symbols registers address e1 e2
genCodeForLine _ _ _ = Nothing

genPICodeOutput :: SymbolTable -> RegisterTable -> Int -> OperatorElement -> OperatorElement
     -> Maybe(CodeLine)
genPICodeOutput symbols registers address i1@(Expr (ExpressionIdentifier _)) i2@(Expr (
     ExpressionIdentifier _)) = genOpCodeOutput symbols registers 193 operands address True
    where operands = i1 : i2 : (Expr (ExpressionNumber 0)) : []
genPICodeOutput symbols registers address i1@(Expr (ExpressionIdentifier _)) i2@(Expr (
     ExpressionNumber _)) = genOpCodeOutput symbols registers 227 operands address True
    where operands = i1 : (Expr (ExpressionNumber 0)) : i2 : []
genPICodeOutput symbols registers address r1@(Register _) r2@(Register _) = genOpCodeOutput
     symbols registers 192 operands address True
    where operands = r1 : r2 : (Expr (ExpressionNumber 0)) : []
genPICodeOutput symbols registers address r1@(Register _) r2@(Expr (ExpressionNumber _)) =
     genOpCodeOutput symbols registers 227 operands address True
    where operands = r1 : r2 : []
genPICodeOuput _ _ _ _ _ = Nothing

genOpCodeOutput :: SymbolTable -> RegisterTable -> Int -> [OperatorElement] -> Int -> Bool ->
     Maybe CodeLine
genOpCodeOutput symbols registers 254 operands address _ = Just(CodeLine {cl_address = address
     , cl_size = 4, cl_code = code})
    where code = splitSpecialRegisters symbols operands
genOpCodeOutput symbols registers opcode operands address False
    | opcode >= 60 && opcode <= 95 = Just(CodeLine {cl_address = address, cl_size = 4, cl_code
          = (chr (opcode + local_adjustment)) : code})
    | opcode == 240 = Just(CodeLine {cl_address = address, cl_size = 4, cl_code = (chr (opcode
          + jump_adjustment)) : jump_code})
    | opcode == 34 = case splitOperandsAddress symbols registers operands of
                         Just(address_adjustment, address_code) -> Just(CodeLine {cl_address =
                              address, cl_size = 4, cl_code = (chr (opcode +
                              address_adjustment)) : address_code})
                         _ -> Nothing
    | otherwise = case splitOperands symbols registers operands of
        Just((adjustment,params)) -> Just(CodeLine {cl_address = address, cl_size = 4, cl_code
               = (chr (opcode + adjustment)) : params})
        _ -> Nothing
      where (local_adjustment, code) = splitLocalOperands symbols operands address
            (jump_adjustment, jump_code) = jumpOperands symbols operands address
genOpCodeOutput symbols registers opcode operands address True =
    case splitOperands symbols registers operands of
        Just((adjustment,params)) -> Just(CodeLine {cl_address = address, cl_size = 4, cl_code
               = (chr opcode) : params})
        _ -> Nothing

localLabelOffset :: Int -> Int -> (Int, Int)
localLabelOffset current required
    | required < current = (1, (quot (current - required) 4))
    | otherwise          = (0, (quot (required - current) 4))

formatElement :: SymbolTable -> OperatorElement -> Char
formatElement _ (ByteLiteral b) = b
formatElement _ (PseudoCode pc) = chr pc
formatElement _ (Register r) = r
formatElement st (Expr x@(ExpressionIdentifier id)) =
    case M.lookup id st of
        (Just (_, Just (IsRegister r))) -> chr r
        (Just (_, Just (IsIdentifier r))) -> formatElement st (Expr (ExpressionIdentifier r))
        otherwise -> evaluateByteToChar st x
formatElement st (Expr x) = evaluateByteToChar st x

evaluateByteToChar :: SymbolTable -> ExpressionEntry -> Char
evaluateByteToChar st x = chr digit
    where plain_digit = (E.evaluate x 0 st)
          digit = if plain_digit < 0
                      then 256 + plain_digit
                      else plain_digit

jumpOperands :: SymbolTable -> [OperatorElement] -> Int -> (Int, String)
jumpOperands symbols ((Ident id):[]) address = (adjustment, code)
    where ro = getSymbolAddress symbols id
          (adjustment, offset) = localLabelOffset address ro
```

```
          b2 = rem offset 256
          q2 = quot offset 256
          b1 = rem q2 256
          b0 = quot q2 256
          code = (chr b0) : (chr b1) : (chr b2) : []

splitOperandsAddress :: SymbolTable -> RegisterTable -> [OperatorElement] -> Maybe(
     AdjustedOperands)
splitOperandsAddress symbols registers (x:(Expr (ExpressionNumber y)):[]) = Just(1, code)
    where registersByAddress = registersFromAddresses registers
          Just(reg, offset) = determineBaseAddressAndOffset registersByAddress (y, Nothing)
          formatted_x = formatElement symbols x
          code = formatted_x : reg : (chr offset) : []
splitOperandsAddress symbols registers (x:Expr (ExpressionIdentifier y):[]) =
    case mapSymbolToAddress symbols registers y of
       Just(y_reg, y_offset) -> Just(1, code)
          where formatted_x = formatElement symbols x
                code = formatted_x : y_reg : (chr y_offset) : []
       otherwise -> Nothing
splitOperandsAddress _ _ _ = Nothing

splitLocalOperands :: SymbolTable -> [OperatorElement] -> Int -> (Int, String)
splitLocalOperands symbols (x:(Ident id):[]) address = (adjustment, code)
    where ro = getSymbolAddress symbols id
          formatted_x = formatElement symbols x
          (adjustment, offset) = localLabelOffset address ro
          b1 = chr (quot offset 256)
          b2 = chr (rem  offset 256)
          code = formatted_x : b1 : b2 : []

splitOperands :: SymbolTable -> RegisterTable -> [OperatorElement] -> Maybe(AdjustedOperands)
splitOperands symbols registers ((Ident id):[]) = Just(1, code)
    where ro = mapSymbolToAddress symbols registers id
          code = case ro of
                     Just((base,offset)) -> (chr 0) : base : (chr offset) : []
splitOperands symbols registers ((Ident id1):(Expr (ExpressionIdentifier id2)):[]) = Just(1,
     code)
    where ro1 = mapSymbolToAddress symbols registers id1
          ro2 = mapSymbolToAddress symbols registers id2
          code = case (ro1, ro2) of
                     (Just((base1,_)), Just((base2,offset2))) -> base1 : base2 : (chr offset2)
                             : []
                     otherwise -> []
splitOperands symbols registers (x:(Expr (ExpressionNumber y)):[]) = Just(1, code)
    where ops = map chr $ drop 2 $ char4 y
          formatted_x = formatElement symbols x
          code = formatted_x : ops
splitOperands symbols registers (x:(Ident id):[]) = Just(1, code)
    where ro = mapSymbolToAddress symbols registers id
          formatted_x = formatElement symbols x
          code = case ro of
                     Just((base,offset)) -> formatted_x : base : (chr offset) : []
                     otherwise -> []
splitOperands symbols registers (x:(Expr (ExpressionIdentifier id)):[]) = Just(1, code)
    where ro = mapSymbolToAddress symbols registers id
          formatted_x = formatElement symbols x
          code = case ro of
                     Just((base,offset)) -> formatted_x : base : (chr offset) : []
                     otherwise -> []
splitOperands symbols registers (x : y : z@(Expr (ExpressionNumber _)) : []) = Just(1, code)
    where formatted_x = formatElement symbols x
          formatted_y = formatElement symbols y
          formatted_z = formatElement symbols z
          code = formatted_x : formatted_y : formatted_z : []
splitOperands symbols registers (x : y : z : []) = Just(0, code)
    where formatted_x = formatElement symbols x
          formatted_y = formatElement symbols y
          formatted_z = formatElement symbols z
          code = formatted_x : formatted_y : formatted_z : []
splitOperands _ _ _ = Nothing

splitSpecialRegisters :: SymbolTable -> [OperatorElement] -> String
splitSpecialRegisters st (x:(Expr (ExpressionIdentifier (Id special))):[]) = (chr 254) : reg :
      special_register_to_operand special
    where reg = formatElement st x

special_register_to_operand :: String -> String
special_register_to_operand "rA"  = (chr 0) : (chr 21) : []
special_register_to_operand "rB"  = (chr 0) : (chr 0)  : []
special_register_to_operand "rC"  = (chr 0) : (chr 8)  : []
special_register_to_operand "rD"  = (chr 0) : (chr 1)  : []
special_register_to_operand "rE"  = (chr 0) : (chr 2)  : []
special_register_to_operand "rF"  = (chr 0) : (chr 22) : []
special_register_to_operand "rG"  = (chr 0) : (chr 19) : []
```

```
special_register_to_operand "rH"  = (chr 0) : (chr 3)  : []
special_register_to_operand "rI"  = (chr 0) : (chr 12) : []
special_register_to_operand "rJ"  = (chr 0) : (chr 4)  : []
special_register_to_operand "rK"  = (chr 0) : (chr 15) : []
special_register_to_operand "rL"  = (chr 0) : (chr 20) : []
special_register_to_operand "rM"  = (chr 0) : (chr 5)  : []
special_register_to_operand "rN"  = (chr 0) : (chr 9)  : []
special_register_to_operand "rO"  = (chr 0) : (chr 10) : []
special_register_to_operand "rP"  = (chr 0) : (chr 23) : []
special_register_to_operand "rQ"  = (chr 0) : (chr 16) : []
special_register_to_operand "rR"  = (chr 0) : (chr 6)  : []
special_register_to_operand "rS"  = (chr 0) : (chr 11) : []
special_register_to_operand "rT"  = (chr 0) : (chr 13) : []
special_register_to_operand "rU"  = (chr 0) : (chr 17) : []
special_register_to_operand "rV"  = (chr 0) : (chr 18) : []
special_register_to_operand "rW"  = (chr 0) : (chr 24) : []
special_register_to_operand "rX"  = (chr 0) : (chr 25) : []
special_register_to_operand "rY"  = (chr 0) : (chr 26) : []
special_register_to_operand "rZ"  = (chr 0) : (chr 27) : []
special_register_to_operand "rBB" = (chr 0) : (chr 7)  : []
special_register_to_operand "rTT" = (chr 0) : (chr 14) : []
special_register_to_operand "rWW" = (chr 0) : (chr 28) : []
special_register_to_operand "rXX" = (chr 0) : (chr 29) : []
special_register_to_operand "rYY" = (chr 0) : (chr 30) : []
special_register_to_operand "rZZ" = (chr 0) : (chr 31) : []

make_bytes :: [Char] -> Int -> [Char]
make_bytes arr size = make_inner_bytes size arr []

make_inner_bytes _ [] acc = acc
make_inner_bytes size (x:xs) acc = make_inner_bytes size xs new_acc
    where extended_byte = make_byte x size []
          new_acc = acc ++ extended_byte

make_byte :: Char -> Int -> [Char] -> [Char]
make_byte _ 0 acc = reverse acc
make_byte b 1 acc = make_byte b 0 (b : acc)
make_byte b n acc = make_byte b (n-1) ((chr 0):acc)

type BlockSummary = (Int, Int, [Int]) -- Starting Address, Size, Data in block

blocks :: [CodeLine] -> [BlockSummary]
blocks [] = []
blocks lines = nextBlock [] (O.sort lines)

nextBlock :: [BlockSummary] -> [CodeLine] -> [BlockSummary]
nextBlock [] (currentLine:rest) = nextBlock [((cl_address currentLine), (cl_size currentLine),
      (map ord (cl_code currentLine)))] rest
nextBlock (currentBlock:blocks) (currentLine:rest)
    | (start + size) == (cl_address currentLine) = nextBlock (updateBlock:blocks) rest
    | otherwise = nextBlock (newBlock:currentBlock:blocks) rest
        where (start, size, code) = currentBlock
              extraCode = map ord (cl_code currentLine)
              updatedCode = code ++ extraCode
              updateBlock = (start, size + (cl_size currentLine), updatedCode)
              newBlock = ((cl_address currentLine), (cl_size currentLine), extraCode)
nextBlock result [] = O.sort result

encodeProgramInt :: [CodeLine] -> RegisterTable -> [Int]
encodeProgramInt prog regs = hdr ++ tbl
    where hdr = header prog
          tbl = encodeRegisterTable regs

header :: [CodeLine] -> [Int]
header program = details
    where bs = blocks program
          num_bs = char4 . length $ bs
          bh = blockDetails (O.sort bs) []
          details = num_bs ++ bh

encodeRegisterTable :: RegisterTable -> [Int]
encodeRegisterTable regs = size ++ vals
    where vals = M.foldrWithKey encodeRegister [] regs
          size = char4 $ M.size regs

encodeRegister :: Char -> ExpressionEntry -> [Int] -> [Int]
encodeRegister r (ExpressionNumber v) a = nextPart ++ a
    where nextPart = (ord r) : char8 v

blockDetails :: [BlockSummary] -> [Int] -> [Int]
blockDetails [] final = final
blockDetails (currentBlock:rest) acc = blockDetails rest newAcc
    where newAcc = acc ++ (blockDetail currentBlock)
```

```
blockDetail :: BlockSummary -> [Int]
blockDetail (start, size, code) = startc ++ sizec ++ code
    where startc = char8 start
          sizec  = char4 size

char4 :: Int -> [Int]
char4 val = char4tail [] val

char4tail :: [Int] -> Int -> [Int]
char4tail acc val
    | (val == -1) && ((length acc) == 4) = acc
    | (val < 0) = case (divMod val 256) of
                     (m, r) -> char4tail (r : acc) m
    | (val == 0) && ((length acc) == 4) = acc
    | (val == 0) = char4tail (0 : acc) val
    | otherwise = case (divMod val 256) of
        (m, r) -> char4tail (r : acc) m

char8 :: Int -> [Int]
char8 val = char8tail [] val

char8tail :: [Int] -> Int -> [Int]
char8tail acc val
    | (val == -1) && ((length acc) == 8) = acc
    | (val < 0) = case (divMod val 256) of
                     (m, r) -> char8tail (r : acc) m
    | (val == 0) && ((length acc) == 8) = acc
    | (val == 0) = char8tail (0 : acc) val
    | otherwise = case (divMod val 256) of
        (m, r) -> char8tail (r : acc) m
```

B.1.9 External Interface

```
module Main where

import MMix_Lexer
import MMix_Parser
import Text.Printf
import qualified Data.Map.Lazy as M
import qualified Data.List.Ordered as O
import Data.Char
import SymbolTable
import CodeGen
import Locations
import Registers
import DataTypes
import Expressions as E
import System.Environment
import Data.List
import System.FilePath
import System.Exit

main :: IO ExitCode
main = do
       args <- getArgs
       case args of
            (source:[]) -> process source
            _           -> displayUsage

process :: [Char] -> IO ExitCode
process sourceFile = do
       let outputFile = replaceExtension sourceFile "se"
       result <- contents sourceFile outputFile
       case result of
           Left _ -> exitFailure
           Right _ -> exitSuccess

displayUsage :: IO ExitCode
displayUsage = do
       putStrLn "usage: MMixAssembler sourceFile"
       exitSuccess

contents ifs ofs = do
    x <- readFile ifs
    let s0 = parseStr x
    let s1 =  setLocalSymbolLabelAuto s0
    let s2 = setAlexLoc s1
    let initial_st = createSymbolTable s2
    let s3 = evaluateAllLocExpressions s2 initial_st
    let s4 = setAlexLoc s3
    let s5 = setAlexGregAuto s4
    let st = createSymbolTable s5
```

```
    let s6 = evaluateAllExpressions s5 st
    let regs = createRegisterTable s6
    let st2 = createSymbolTable s6
    let code = acg st2 regs s6
    --print code
    let pg = encodeProgram code regs
    case pg of
        Right encoded_program -> writeFile ofs encoded_program
        Left error            -> putStrLn error
    return pg

setAlexLoc :: Either String [Line] -> Either String [Line]
setAlexLoc (Right lns) = Right $ setLoc 0 lns
setAlexLoc m = m

setLoc :: Int -> [Line] -> [Line]
setLoc startLoc lns = setInnerLocation startLoc [] lns

showAlexLocs :: Either String [Line] -> Either String [Int]
showAlexLocs (Right lns) = Right $ showLocs lns
showAlexLocs (Left msg) = Left msg

showLocs :: [Line] -> [Int]
showLocs lns = foldr showLoc [] lns

showLoc :: Line -> [Int] -> [Int]
showLoc (PlainPILine _ loc) acc =  loc : acc
showLoc (LabelledPILine _ _ loc) acc =  loc : acc
showLoc (PlainOpCodeLine _ _ loc _) acc =  loc : acc
showLoc (LabelledOpCodeLine _ _ _ loc _) acc =  loc : acc


acg (Right sym) (Right regs) (Right lns) = Right $ cg sym regs [] lns
acg _ _ _ = Left "Something is missing!!!"

--cg :: (M.Map String RegisterAddress) -> (M.Map Char Int) -> [Line] -> [Line]
cg _ _ acc [] = acc
cg s r acc (ln:lns) = cg s r newAcc lns
    where cgl = genCodeForLine s r ln
          newAcc = case cgl of
              Just(codeline) -> codeline : acc
              Nothing -> acc
                  where newline = CodeLine {cl_address = 0, cl_size = 0, cl_code = (show ln)}
```

B.2 Graphical User Interface

B.2.1 Scala Build Tool

```
name := "MMixGUI"

version := "1.0"

resolvers += "Typesafe Repositorty" at "http://repo.typesafe.com/typesafe/releases"

libraryDependencies ++= Seq(
  "org.scala-lang" % "scala-swing" % "2.10+",
  "com.typesafe.akka" %% "akka-actor" % "2.3.4"
)
```

B.2.2 Console Panel

```
package com.steveedmans.mmix.panels

import akka.actor.{Props, ActorLogging, Actor, ActorSystem}
import com.steveedmans.mmix.panels.ConsolePanel.{ClearPanel, DisplayText}
import com.steveedmans.mmix.{GuiEvent, GUIProgressEventHandler}
import scala.swing.GridBagPanel.Fill
import scala.swing._
import scala.swing.BorderPanel.Position._

object ConsolePanel {
  case class DisplayText(txt : String) extends GuiEvent
  case object ClearPanel extends GuiEvent
}

class ConsolePanel(system: ActorSystem) extends BorderPanel with GUIProgressEventHandler {
```

```
  val worker = createWorkerActor()
  preferredSize = new Dimension(100, 200)
  minimumSize = new Dimension(100, 200)

  lazy val lbl = new Label {
    text = "Console Input "
    border = Swing.EmptyBorder(5,5,5,5)
  }

  lazy val user_input = new TextField

  lazy val labeled_field = new GridBagPanel {
    val c = new Constraints
    layout(lbl) = c
    c.weightx = 2.0
    c.fill = Fill.Horizontal
    layout(user_input) = c
  }

  lazy val console = new TextArea {
    editable = false
    border = Swing.CompoundBorder(Swing.EmptyBorder(5), Swing.LineBorder(java.awt.Color.BLACK)
        )
  }

  val scroller = new ScrollPane(console)

  layout(labeled_field) = South
  layout(scroller) = Center

  override def handleGuiProgressEvent(event: GuiEvent): Unit = {
    event match {
      case DisplayText(txt) =>
        val new_text = console.text + txt
        console.text = new_text
      case ClearPanel =>
        console.text = ""
    }
  }

  def createWorkerActor() = {
    val guiUpdateActor = system.actorOf(
      Props(new ConsoleUpdateActor(this)), name = "consoleUpdateActor")
  }

  class ConsoleUpdateActor(handler: GUIProgressEventHandler) extends Actor with ActorLogging {
    def receive = {
      case msg : GuiEvent => handler.handleGuiProgressEvent(msg)
      case msg =>
        log.debug(s"WE HAVE A MESSAGE $msg")
    }
  }
}
```

B.2.3 Controls Panel

```
package com.steveedmans.mmix.panels

import akka.actor._
import com.steveedmans.mmix.actors.VirtualMachineActor
import com.steveedmans.mmix.panels.ConsolePanel.ClearPanel
import com.steveedmans.mmix.panels.MainStatePanel.ResetMainState
import com.steveedmans.mmix.{GUIProgressEventHandler, MMixFile, GuiEvent}

import scala.swing.event.ButtonClicked
import scala.swing.{Swing, Orientation, BoxPanel, Button, FileChooser}

object ControlsPanel {
  case class GuiProgressEvent(percentage: Int) extends GuiEvent
  object ProcessingFinished extends GuiEvent
  case object ProgramFinished extends GuiEvent
  case object ProgramReady extends GuiEvent
}

class ControlsPanel(system: ActorSystem) extends BoxPanel(orientation = Orientation.Horizontal
     ) with GUIProgressEventHandler {
  import ControlsPanel._
  import ProcessNextActor._

  val worker = createActorSystemWithWorkerActor()

  lazy val resetSimulator = new Button {
```

```
  text = "Reset Simulator"
}
listenTo(resetSimulator)
contents += resetSimulator
lazy val loadProgram = new Button {text = "Load Program"}
listenTo(loadProgram)
contents += loadProgram
lazy val resetProgram = new Button {text = "Reset Program"}
listenTo(resetProgram)
contents += resetProgram
lazy val processNext = new Button {text = "Process Next"}
listenTo(processNext)
contents += processNext
lazy val automation = new Button {text = "Start"}
listenTo(automation)
contents += automation

reactions += {
  case m : ButtonClicked if m.source == resetSimulator => println("RESET SIMULATOR")
  case m : ButtonClicked if m.source == loadProgram => loadProgramFile()
  case m : ButtonClicked if m.source == resetProgram => resetProgramAction()
  case m : ButtonClicked if m.source == processNext => processNextAction()
  case m : ButtonClicked if m.source == automation => toggleAutomation()
  case m : ButtonClicked => println("A button has been pressed")
}

def resetProgramAction() = {
  worker ! Stop
}

def toggleAutomation() = {
  automation.text match {
    case "Start" => Swing.onEDT {
      automation.text = "Stop"
      worker ! StartAutomation
    }
    case _ => Swing.onEDT {
      automation.text = "Start"
      worker ! StopAutomation
    }
  }
}

def loadProgramFile() = {
  var mmixFile = new FileChooser()
  if (mmixFile.showOpenDialog(this) == FileChooser.Result.Approve) {
    val file = MMixFile.openFile(mmixFile.selectedFile.getAbsolutePath)
    system.actorSelection("/user/memoryUpdateActor") ! MemoryPanel.NewProgram(file)
    system.actorSelection("/user/mainStateUpdateActor") ! ResetMainState
    system.actorSelection("/user/consoleUpdateActor") ! ClearPanel
    worker ! LoadProgram(file)
  }
}

def handleGuiProgressEvent(event: GuiEvent) {
  event match {
    case GuiProgressEvent(pct) => println(pct)
    case ProcessingFinished => Swing.onEDT{
      processNext.enabled = true
    }
    case FinishProcessing => Swing.onEDT {
      processNext.enabled = true
    }
    case ProgramFinished => Swing.onEDT {
      processNext.enabled = false
      automation.text = "Start"
      worker ! StopAutomation
    }
    case ProgramReady => Swing.onEDT {
      processNext.enabled = true
    }
  }
}

def processNextAction() = {
  worker ! ProcessNext
}

def createActorSystemWithWorkerActor():ActorRef = {
  val guiUpdateActor = system.actorOf(
    Props(new GUIUpdateActor(this)), name = "controlsUpdateActor")

  val vmActor = system.actorOf(Props[VirtualMachineActor], "vmActor")
```

```
    val workerActor = system.actorOf(
      Props(new WorkerActor(guiUpdateActor, vmActor)), name = "workerActor")

    workerActor
  }

  class GUIUpdateActor(val gui:GUIProgressEventHandler) extends Actor {
    def receive = {
      case event: GuiEvent => gui.handleGuiProgressEvent(event)
    }
  }
}

class WorkerActor(val guiUpdateActor: ActorRef, val vmActor : ActorRef) extends Actor with
     ActorLogging {
  import ProcessNextActor._
  import VirtualMachineActor._
  import AutomationState._
  import scala.concurrent.duration._
  import context.dispatcher

  var currentState = STOPPED

  def receive = {
    case ProcessNext =>
      log.debug("Processing Next Statement")
      vmActor ! SendData(Symbol("process_next"))
    case LoadProgram(file) =>
      log.debug(s"LOAD PROGRAM")
      vmActor ! SendData((Symbol("program"), file.Memory))
      vmActor ! SendData((Symbol("registers"), file.Registers))
      vmActor ! SendData(Symbol("get_all_registers"))
      guiUpdateActor ! ControlsPanel.ProgramReady
    case Stop =>
      log.debug("Stopping the Virtual Machine")
      vmActor ! SendData(Symbol("stop"))
    case StartAutomation =>
      log.debug("Start Automation")
      currentState = RUNNING
      log.debug("Processing Next Statement")
      vmActor ! SendData(Symbol("process_next"))
    case StopAutomation =>
      log.debug("Stop Automation")
      currentState = STOPPED
    case Automate =>
      currentState match {
        case RUNNING =>
          context.system.scheduler.scheduleOnce(1 millis, self, ProcessNext)
        case STOPPED =>
          log.debug("We are not automating so we ignore this")
      }
    case msg =>
      log.debug(s"We have received an unknown message $msg")
  }
}

object AutomationState extends Enumeration {
  type AutomationState = Value
  val RUNNING, STOPPED = Value
}

object ProcessNextActor {
  case object ProcessNext
  case class LoadProgram(program : MMixFile)
  case object Stop
  case object StartAutomation
  case object StopAutomation
  case object Automate
  case object FinishProcessing extends GuiEvent
}
```

B.2.4 Main Form

```
package com.steveedmans.mmix

import akka.actor.ActorSystem
import com.steveedmans.mmix.panels._
import scala.swing._
import scala.swing.BorderPanel.Position._
import scala.swing.event.WindowClosing

object main_form extends SimpleSwingApplication {
```

```
  val system = ActorSystem("MMixGui")
  def top = new MainFrame {
    title = "MMix Simulator"
    preferredSize = new Dimension(1200, 700)
    minimumSize = new Dimension(1200, 700)
    lazy val mainPanel = new BorderPanel {
      lazy val left = new RegisterPanel(system)
      layout(left) = West
      lazy val memory = new MemoryPanel(system)
      lazy val console = new ConsolePanel(system)
      lazy val state = new MainStatePanel(system)
      lazy val controls = new ControlsPanel(system)
      lazy val rightPanel = new BoxPanel(orientation = Orientation.Vertical) {
        contents += memory
        contents += console
        contents += state
        contents += controls
      }
      layout(rightPanel) = Center
    }
    contents = mainPanel
    listenTo(mainPanel)
    reactions += {
      case WindowClosing(_) => system.shutdown()
    }
  }
}
```

B.2.5 Main State Panel

```
package com.steveedmans.mmix.panels

import java.awt.Color

import akka.actor.{ActorLogging, Actor, Props, ActorSystem}
import com.steveedmans.mmix.panels.MainStatePanel.{UpdateMainStateRegisters, ResetMainState,
     RecordStatement}
import com.steveedmans.mmix.{GuiEvent, GUIProgressEventHandler}

import scala.collection.BitSet
import scala.swing.BorderPanel.Position._
import scala.swing._
import scala.util.Random

object MainStatePanel {
  case class RecordStatement(statement : String) extends GuiEvent
  case class UpdateMainStateRegisters(registers : List[Tuple2[Any, Long]]) extends GuiEvent
  case object ResetMainState extends GuiEvent
}

class MainStatePanel (system: ActorSystem) extends BorderPanel with GUIProgressEventHandler {
  val worker = createWorkerActor()
  preferredSize = new Dimension(100, 200)
  minimumSize = new Dimension(100, 200)

  lazy val console = new TextArea {
    editable = false
    border = Swing.EmptyBorder(5)
  }

  lazy val divide_check = new CheckBox("Integer Divide Check")
  lazy val overflow = new CheckBox("Integer Overflow")
  lazy val fix_to_float = new CheckBox("Fix to Float Overflow")
  lazy val invalid = new CheckBox("Invalid Operation")
  lazy val floating_overflow = new CheckBox("Floating Overflow")
  lazy val underflow = new CheckBox("Floating Underflow")
  lazy val zero = new CheckBox("Division by Zero")
  lazy val Inexact = new CheckBox("Floating Inexact")

  lazy val arithmetic_flags = new GridPanel(4, 2) {
    contents += divide_check
    contents += floating_overflow
    contents += overflow
    contents += underflow
    contents += fix_to_float
    contents += zero
    contents += invalid
    contents += Inexact
  }

  val scroller = new ScrollPane(console)
```

```
  var statements = List.empty[String]

  layout(scroller) = Center
  layout(arithmetic_flags) = East

  override def handleGuiProgressEvent(event: GuiEvent): Unit = {
    event match {
      case RecordStatement(statement) =>
        statements = statement.toUpperCase :: statements.take(299)
        console.text = statements.mkString("\n")
      case ResetMainState =>
        console.text = ""
        statements = List.empty[String]
      case UpdateMainStateRegisters(registers) =>
        update_state(registers)
      case msg =>
        println(s"MAIN STATE PANEL EVENT HANDLER HAS RECEIVED A MESSAGE $msg")
    }
  }

  def update_state(registers : List[Tuple2[Any, Long]]) = {
    registers.foreach {
      case ('rA, value) => update_arithmetic_state(value)
      case _ => //Ignore
    }
  }

  def update_arithmetic_state(value : Long) = {
    Range(0,8).foldLeft(value){(acc, indx) =>
      indx match {
        case 0 => Inexact.selected = ((acc % 2) == 1)
        case 1 => zero.selected = ((acc % 2) == 1)
        case 2 => underflow.selected = ((acc % 2) == 1)
        case 3 => floating_overflow.selected = ((acc % 2) == 1)
        case 4 => invalid.selected = ((acc % 2) == 1)
        case 5 => fix_to_float.selected = ((acc % 2) == 1)
        case 6 => overflow.selected = ((acc % 2) == 1)
        case 7 => divide_check.selected = ((acc % 2) == 1)
      }
      acc / 2
    }
  }

  def createWorkerActor() = {
    val guiUpdateActor = system.actorOf(
      Props(new MainStatePanelUpdateActor(this)), name = "mainStateUpdateActor")
  }

  class MainStatePanelUpdateActor(handler: GUIProgressEventHandler) extends Actor with
       ActorLogging {
    def receive = {
      case msg : GuiEvent => handler.handleGuiProgressEvent(msg)
      case msg =>
        log.debug(s"MAIN STATE PANEL HAS RECEIVED A MESSAGE $msg")
    }
  }
}
```

B.2.6 Memory Panel

```
package com.steveedmans.mmix.panels

import akka.actor.{Actor, Props, ActorSystem}
import com.steveedmans.mmix.{MMixFile, GuiEvent, GUIProgressEventHandler}

import scala.collection.SortedMap
import scala.swing._
import com.steveedmans.mmix.Utilities._
import java.awt.Color

class HexString(val s : String) {
  def hex = java.lang.Long.parseLong(s, 16)
}

object MemoryPanel {
  val BLOCK_SIZE : Int = 4
  val td = Map.empty[String, String]

  case class NewProgram(file : MMixFile) extends GuiEvent
  case class UpdateAddress(address : Long, value : Byte) extends GuiEvent
  case object StartNewSetOfUpdates extends GuiEvent
  case object RefreshTable extends GuiEvent
```

```
  implicit def str2hex(str: String) : HexString = new HexString(str)
  // 16 hex digits for a memory address

  def mem_address(address : Long) : String = {
    "%016X".format(address)
  }

  def mem_contents(value : Int) : String = {
    "%02X".format(value.toByte)
  }
}

class MemoryPanel(system: ActorSystem) extends ScrollPane with GUIProgressEventHandler {
  import MemoryPanel._

  val worker = createWorkerActor()

  lazy val data = new BoxPanel(orientation = Orientation.Vertical)
  var memory : Map[String, String] = Map.empty
  var main_memory = SortedMap[Long, Int]()
  var updated_locations : List[Long] = List.empty
  var panel_content = getContent()
  data.contents ++= panel_content
  preferredSize = new Dimension(100, 200)
  minimumSize = new Dimension(100, 200)
  border = Swing.CompoundBorder(Swing.EmptyBorder(5), Swing.LineBorder(java.awt.Color.BLACK))
  viewportView = data

  def reset(memory : Map[String, String]) = {
    println(s"About to set the memory to $memory")
  }

  def extract_blocks(memory : Map[String, String]): List[SortedMap[Long, Int]] = {
    val blocks = get_blocks(main_memory, List())
    blocks
  }

  def getContent() : List[Component] = {
    val blocks = extract_blocks(memory).reverse
    val final_column = BLOCK_SIZE + 1
    val table = new Table(blocks.length, BLOCK_SIZE + 2) {
      rowHeight = 25
      autoResizeMode = Table.AutoResizeMode.Off
      showGrid = true
      gridColor = new Color(150, 150, 150)

      override def rendererComponent(isSelected: Boolean, hasFocus : Boolean, row : Int,
           column: Int) : Component = {
        val data_row = blocks(row)
        val block_start = data_row.head._1
        val address = block_start + column - 1
        new Label {
          text = column match {
            case 0 =>
              mem_address(block_start)
            case _ if column == final_column =>
              data_row.values.map(b => {
                if (b >= 32 && b <= 127)
                  b.toChar
                else
                  '.'
              }).mkString
            case _ if data_row.contains(block_start + column - 1) =>
              mem_contents(data_row(address))
            case _ => ""
          }
          xAlignment = Alignment.Center
          if (hasFocus) {
            opaque = true
            if ((column == final_column) || (column == 0)) {
              val addresses = block_start to block_start + 3
              background = if (addresses.exists(updated_locations.contains(_))) Color.GREEN
                   else Color.cyan
            }
            else
              background = if (updated_locations.contains(address)) Color.GREEN else Color.
                   cyan
          } else {
            if (isSelected) {
              if ((column == final_column) || (column == 0)) {
                val addresses = block_start to block_start + 3
                background = if (addresses.exists(updated_locations.contains(_))) Color.GREEN
                     else Color.cyan
```

```
            }
            else
              background = if (updated_locations.contains(address)) Color.GREEN else Color.
                   cyan
            opaque = true
          } else {
            background = Color.GREEN
            if ((column == final_column) || (column == 0)) {
              val addresses = block_start to block_start + 3
              opaque = addresses.exists(updated_locations.contains(_))
            }
            else
              opaque = updated_locations.contains(address)
          }
        }
      }
    }
  }
  val model = table.peer.getColumnModel
  model.getColumn(0).setPreferredWidth(140)
  for (counter <- 0 until BLOCK_SIZE) {
    model.getColumn(counter + 1).setPreferredWidth(20)
  }
  table.peer.getColumnModel.getColumn(final_column).setPreferredWidth(10 * BLOCK_SIZE)

  List(table)
}

def to_main_memory(blocks : Map[String, String]) : SortedMap[Long, Int] = {
  blocks.foldLeft(SortedMap[Long, Int]())((acc, block) => {
    block match {
      case (start_location, data) =>
        process_block(hex2dec(start_location).toLong, data, acc)
      case _ =>
        acc
    }
  })
}

def process_block(start_location : Long, data : String, old_map : SortedMap[Long, Int]) = {
  val as_int = hex2bytes(data)
  as_int.view.zipWithIndex.foldLeft(old_map)((acc, item) => {
    item match {
      case (value, index) =>
        val location = start_location + index
        acc + (location -> value)
      case _ =>
        acc
    }
  })
}

def get_blocks(memory : SortedMap[Long, Int], blocks : List[SortedMap[Long, Int]]) : List[
     SortedMap[Long, Int]] = {
  memory match {
    case _ if memory.size > 0 =>
      val (next_block, remaining) = get_block(memory)
      get_blocks(remaining, next_block :: blocks)
    case _ => blocks
  }
}
def get_block(memory : SortedMap[Long, Int]) = {
  val block_data = memory.take(BLOCK_SIZE)
  val block_start= block_data.head._1
  val data = block_data.filter {
    case (cell, value) =>
      (cell - block_start) < BLOCK_SIZE
  }
  (data, memory.drop(data.size))
}

def createWorkerActor() = {
  val guiUpdateActor = system.actorOf(
    Props(new MemoryUpdateActor(this)), name = "memoryUpdateActor")
}

override def handleGuiProgressEvent(event: GuiEvent): Unit = {
  event match {
    case NewProgram(file) =>
      main_memory = to_main_memory(file.MemoryBlocks)
      refreshTable()
    case UpdateAddress(location, value) =>
      main_memory += (location -> value)
      updated_locations = updated_locations.::(location)
```

```
      case StartNewSetOfUpdates =>
        updated_locations = List.empty
      case RefreshTable =>
        refreshTable()
    }
  }

  def refreshTable() {
    val contents = getContent()
    data.contents --= panel_content
    panel_content = contents
    data.contents ++= panel_content
    data.revalidate()
    data.repaint()
  }

  class MemoryUpdateActor(handler: GUIProgressEventHandler) extends Actor {
    def receive = {
      case msg : GuiEvent =>
        handler.handleGuiProgressEvent(msg)
    }
  }
}
```

B.2.7 MMix File

```
package com.steveedmans.mmix

import scala.io.Source

sealed trait MMixFile {
  def MemoryBlocks : Map[String, String]
  def Memory : List[(Long, List[Byte])]
  def Registers : List[(Any, Long)]
}

object MMixFile {

  def openFile(fileName : String) : MMixFile = {
    val file = readFile(fileName)
    val (numBlocks, rest) = Utilities.nextChar4(file)
    val bs = List.fill(numBlocks)(0)

    val (regs_data, regs_code) = bs.foldLeft((rest, List[(String, String)]())){
      (a, b) => {
        val (c, d) = a
        val (e, f, g) = readBlock(c)
        val h = (f, g) :: d
        (e, h)
      }
    }

    val (numRegs, rest_r) = Utilities.nextChar4(regs_data)

    val bsr = List.fill(numRegs)(0)

    val (remaining, regs) = bsr.foldLeft(rest_r, List[(Byte, Long)]()){
      (a, b) => {
        val (c, d) = a
        val (e, f, g) = readRegister(c)
        val h = (f, g) :: d
        (e, h)
      }
    }

    new MMixFileImpl(regs_code, regs)
  }

  private def readRegister(data: List[Byte]) : (List[Byte], Byte, Long) = {
    val reg = data.head
    val (value, remaining) = Utilities.nextChar8(data.tail)
    (remaining, reg, value)
  }

  private def readBlock(data : List[Byte]) : (List[Byte], String, String) = {
    val (address, s1) = Utilities.nextChar8Hex(data)
    val (size, s2)    = Utilities.nextChar4(s1)
    val (code_list, rest) = s2.splitAt(size)
    val code = Utilities.bytes2hex(code_list)
    (rest, address, code)
  }
```

```
  private def readFile(fileName : String) = {
    Source.fromFile(fileName).map(_.toByte).toList
  }

  private class MMixFileImpl(code : List[(String, String)], registers : List[(Byte, Long)])
       extends MMixFile {
    override def toString = {
      s"MMIX File with Code = $code & Registers = $registers"
    }

    override def MemoryBlocks: Map[String, String] = {
      code.toMap[String, String]
    }

    override def Memory : List[(Long, List[Byte])] = {
      code map {
        case (address, memory) => BlockOfMemory(address, memory)
      }
    }

    def BlockOfMemory(address : String, memory : String) : (Long, List[Byte]) = {
      //val start_address = Integer.parseInt(address, 16)
      val start_address = Utilities.hex2dec(address).toLong
      val memory_list = Utilities.hex2bytes(memory)
      (start_address, memory_list)
    }

    override def Registers : List[(Any, Long)] = {
      registers
    }
  }
}
```

B.2.8 Register Panel

```
package com.steveedmans.mmix.panels

import java.awt.Color
import java.util.Date

import com.steveedmans.mmix.panels.MainStatePanel.UpdateMainStateRegisters

import scala.collection.mutable.Map
import akka.actor.{ActorLogging, ActorSystem, Actor, Props}
import com.steveedmans.mmix.{Utilities, GuiEvent, GUIProgressEventHandler}

import scala.swing._

object RegisterPanel {
  case class FullRegisterSet(registers : List[(Any, Long)]) extends GuiEvent
  case class UpdatedRegisters(registers : List[(Any, Long)]) extends GuiEvent

  def createRegister(key : Any, value : Long, modified : Date) : Register = {
    (key, value) match {
      case (sym : Symbol, value : Long) => Register(SystemRegister(sym), value, modified)
      case (reg : Int, value : Long) => Register(UserRegister(reg), value, modified)
    }
  }

  sealed trait RegisterType {
    def key : String
  }

  case class SystemRegister(name : Symbol) extends RegisterType {
    def key : String = name.name
  }
  case class UserRegister(value : Long) extends RegisterType {
    def key : String = Utilities.byte2hex(value.toByte).toUpperCase
  }

  case class Register(registerType : RegisterType, var value : Long, var modifiedDate : Date)
       {
    def key : String = {
      registerType.key
    }

    def getValue : String = {
      Utilities.long2hex(value)
    }
  }
}
```

```
class RegisterPanel(system: ActorSystem) extends ScrollPane with GUIProgressEventHandler {
  import RegisterPanel._
  val worker = createWorkerActor()

  lazy val panel = new BoxPanel(orientation = Orientation.Vertical)
  var panel_content = createTable()
  panel.contents ++= panel_content
  viewportView = panel
  preferredSize = new Dimension(240, 10)
  var data : scala.collection.mutable.Map[String, Register] = Map.empty
  var data_keys : Array[Register] = _

  def createTable() : List[Component] = {
    val table = new Table(290, 2) {
      rowHeight = 20
      showGrid = true
      gridColor = new Color(150, 150, 150)

      override def rendererComponent(isSelected: Boolean, hasFocus : Boolean, row : Int,
           column: Int) : Component = {
        val last_modified = data_keys match {
          case dk if dk != null => if (data_keys.size > 0) data_keys(0).modifiedDate else new
               Date()
          case _ => new Date()
        }
        new Label {
          text = (row, column) match {
            case (0, 0) => "Reg"
            case (0, 1) => "Val"
            case (row : Int, 0) if row <= data.size =>
              data_keys(row - 1).key
            case (row : Int, 1) if row <= data.size =>
              data_keys(row -1).getValue.toUpperCase
            case _ => ""
          }
          xAlignment = Alignment.Center
          background = (row, data.size) match {
            case (r, s) if (r > 0) && (s > 0) && (r <= s) => if (data_keys(row - 1).
                 modifiedDate == last_modified) Color.green else Color.cyan
            case _ => Color.cyan
          }
          opaque = (hasFocus, isSelected, data.size) match {
            case (true, _, _) => true
            case (_, true, _) => true
            case (_, _, s) if (row > 0) && (s > 0) && (row <= s) => data_keys(row - 1).
                 modifiedDate == last_modified
            case _ => false
          }
        }
      }
    }
    val model = table.peer.getColumnModel
    val col0 = model.getColumn(0)
    col0.setPreferredWidth(60)
    col0.setHeaderValue("Reg")
    val col1 = model.getColumn(1)
    col1.setPreferredWidth(160)
    col1.setHeaderValue("Val")
    List(table)
  }

  def createWorkerActor() = {
    system.actorOf(Props(new RegistersUpdateActor(this)), name = "registersUpdateActor")
  }

  override def handleGuiProgressEvent(event: GuiEvent): Unit = {
    event match {
      case FullRegisterSet(registers) =>
        set_registers(registers)
      case UpdatedRegisters(registers) =>
        update_registers(registers)
      case _ =>
        println("Handle Registers Gui Event")
    }
  }

  def split_registers(registers: List[Tuple2[Any, Long]]) = {
    registers.partition(kvp => {
      kvp match {
        case ('rA, _) => false
        case _         => true
      }
    })
  }
```

```
  def set_registers(registers: List[Tuple2[Any, Long]]) = {
    val (user_regs, main_state_regs) = split_registers(registers)
    update_main_state(main_state_regs)
    val regs = convertToListOfRegisters(user_regs)
    data_keys = regs.sortWith(sort_register).toArray
    data = scala.collection.mutable.Map() ++ regs.map({ rr => (rr.key, rr) }).toMap
    panel.revalidate()
    panel.repaint()
  }

  def update_registers(regs : List[Tuple2[Any, Long]]) = {
    val today = new java.util.Date()
    val (user_regs, main_state_regs) = split_registers(regs)
    update_main_state(main_state_regs)
    convertToListOfRegisters(user_regs).foreach(reg => {
      val ev = data(reg.key)
      val nv = ev.copy(modifiedDate = today, value = reg.value)
      data(reg.key) = nv
    })
    data_keys = data.map(_._2).toList.sortWith(sort_register).toArray
    panel.revalidate()
    panel.repaint()
  }

  def update_main_state(ms_registers : List[Tuple2[Any, Long]]) = {
    system.actorSelection("/user/mainStateUpdateActor") ! UpdateMainStateRegisters(
         ms_registers)
  }

  def convertToListOfRegisters(registers : List[(Any, Long)]) : List[Register] = {
    val today = new java.util.Date()
    registers map { case (k, v) =>
      RegisterPanel.createRegister(k, v, today)
    }
  }

  def sort_register(r1: Register, r2: Register) = {
    (r2.registerType, r1.registerType) match {
      case (sr : SystemRegister, _ : UserRegister) if sr.key == "pc" => false
      case (_ : SystemRegister, _ : UserRegister) => true
      case (_ : UserRegister, sr : SystemRegister) if sr.key == "pc" => true
      case (_ : UserRegister, _ : SystemRegister) => false
      case (v1 : SystemRegister, v2 : SystemRegister) =>
        v1.name.name.compareTo(v2.name.name) > 0
      case (v1 : UserRegister, v2 : UserRegister) =>
        val v1T = Utilities.byte2hex(v1.value.toByte).toUpperCase
        val v2T = Utilities.byte2hex(v2.value.toByte).toUpperCase
        v2T.compareTo(v1T) > 0
    }
  }

  def sort_updated_registers(r1: Register, r2: Register) = {
    r1.modifiedDate.compareTo(r2.modifiedDate) match {
      case diff if diff == 0 =>
        sort_register(r1, r2)
      case diff => diff > 0
    }
  }

  class RegistersUpdateActor(handler: GUIProgressEventHandler) extends Actor with ActorLogging
        {
    import RegisterPanel._
    def receive = {
      case frs : FullRegisterSet =>
        handler.handleGuiProgressEvent(frs)
      case ur : UpdatedRegisters =>
        handler.handleGuiProgressEvent(ur)
      case msg =>
        log.debug(s"WE HAVE A MESSAGE $msg")
    }
  }
}
```

B.2.9 Utilities

```
package com.steveedmans.mmix

abstract class GuiEvent

trait GUIProgressEventHandler {
  def handleGuiProgressEvent(event: GuiEvent)
```

```
}

object Utilities {

  def fromChar4(original : List[Byte]) : Int = {
    original.foldLeft(0){
      (a,b) =>
        val next_val = if (b < 0) 256 + b else b
        a * 256 + next_val
    }
  }

  def fromChar8(original : List[Byte]) : Long = {
    original.foldLeft(0 : Long){
      (a,b) =>
        val next_val : Long = if (b < 0) 256 + b else b
        val new_acc : Long = a * 256 + next_val
        new_acc
    }
  }

  def nextChar4(start : List[Byte]) : (Int, List[Byte]) = {
    val (next, rest) = start.splitAt(4)
    (fromChar4(next), rest)
  }

  def nextChar8(start : List[Byte]) : (Long, List[Byte]) = {
    val (next, rest) = start.splitAt(8)
    (fromChar8(next), rest)
  }

  def nextChar4Hex(start : List[Byte]) : (String, List[Byte]) = {
    val (next, rest) = start.splitAt(4)
    (bytes2hex(next), rest)
  }

  def nextChar8Hex(start : List[Byte]) : (String, List[Byte]) = {
    val (next, rest) = start.splitAt(8)
    (bytes2hex(next), rest)
  }

// From https://gist.github.com/tmyymmt/3721117

  def hex2bytes(hex: String): List[Byte] = {
    if(hex.contains(" ")){
      hex.split(" ").map(Integer.parseInt(_, 16).toByte).toList
    } else if(hex.contains("-")){
      hex.split("-").map(Integer.parseInt(_, 16).toByte).toList
    } else {
      hex.sliding(2,2).toArray.map(Integer.parseInt(_, 16).toByte).toList
    }
  }

  def bytes2hex(bytes: List[Byte], sep: Option[String] = None): String = {
    sep match {
      case None =>  bytes.map("%02x".format(_)).mkString
      case _ =>  bytes.map("%02x".format(_)).mkString(sep.get)
    }
  }

  def byte2hex(byte: Byte) : String = {
    bytes2hex(List(byte))
  }

  def hex2dec(hex: String): BigInt = {
    hex.toLowerCase().toList.map(
      "0123456789abcdef".indexOf(_)).map(
        BigInt(_)).reduceLeft( _ * 16 + _)
  }

  def int2hex(num : Int) : String = {
    import scala.math._
    if (num == 0) {
      bytes2hex(List(0,0,0,0))
    } else {
      bytes2hex(Range(0, 4).foldLeft(List[Byte](), num) {
        (accumulator, _) =>
          val (workingList, remainder) = accumulator
          ((remainder % 256).toByte :: workingList, remainder >>> 8)
      }._1)
    }
  }

  def long2hex(num : Long) : String = {
```

```
    import scala.math._
    if (num == 0) {
      bytes2hex(List(0,0,0,0,0,0,0,0))
    } else {
      bytes2hex(Range(0, 8).foldLeft(List[Byte](), num) {
        (accumulator, _) =>
          val (workingList, remainder) = accumulator
          ((remainder % 256).toByte :: workingList, remainder >>> 8)
      }._1)
    }
  }
}
```

B.2.10 Virtual Machine Actor

```
package com.steveedmans.mmix.actors

import java.net.{InetAddress, DatagramPacket, DatagramSocket}

import akka.actor.{Actor, ActorLogging}
import com.steveedmans.mmix.panels.MainStatePanel.RecordStatement
import com.steveedmans.mmix.panels.MemoryPanel.{RefreshTable, StartNewSetOfUpdates}
import com.steveedmans.mmix.panels.ProcessNextActor.Automate
import com.steveedmans.mmix.panels.{RegisterPanel, ConsolePanel, ControlsPanel, MemoryPanel,
     MainStatePanel}
import com.steveedmans.mmix.Utilities

object VirtualMachineActor {
  case class SendData(data : Any)
  case object ProcessNextStatement

  private object Udp_Client {
    val bufsize = 50
    val port = 4000

    val SMALL_INTEGER_EXT = 97
    val INTEGER_EXT = 98
    val SMALL_TUPLE_EXT = 104
    val SMALL_BIG_EXT = 110
    val ATOM_EXT = 100
    val STRING_EXT = 107
    val LIST_EXT = 108
    val NIL_EXT = 106

    val PROCESS_NEXT = Symbol("ProcessNext")

    def term_to_binary(term : Any) : Array[Byte] = {
      val data : List[Int] = 131 :: construct_term(term)
      data.map(i => i.toByte).toArray
    }

    def construct_term(term : Any) : List[Int] = term match {
      case b : Byte => SMALL_INTEGER_EXT :: b.toInt :: Nil
      case b : Int if (b >= -128) && (b < 256) => SMALL_INTEGER_EXT :: b :: Nil
      case i : Int => INTEGER_EXT :: int_to_bytes(4, i)
      case l : Long => SMALL_BIG_EXT :: 8 :: 0 :: long_to_bytes(l)
      case atom : Symbol => ATOM_EXT :: symbol_to_bytes(atom)
      case list : List[_] => LIST_EXT :: list_to_bytes(list)
      case tup : Product =>  SMALL_TUPLE_EXT :: tuple_to_bytes(tup)
      case str : String =>  STRING_EXT :: string_to_bytes(str)
    }

    def list_to_bytes(value : List[_]) : List[Int] = {
      val (data, size) = value.foldRight(List[Int](), 0){
        (value, results : (List[Int], Int)) => (construct_term(value) ::: results._1, results.
            _2 + 1)
      }
      int_to_bytes(4, size) ::: data ::: List[Int](NIL_EXT)
    }

    def string_to_bytes(value : String) : List[Int] = {
      int_to_bytes(2, value.length) ::: value.toCharArray.map(c => c.toInt).toList
    }

    def tuple_to_bytes(tuple : Product) : List[Int] = {
      tuple.productArity :: tuple.productIterator.foldRight(List[Int]()) {
        (value, results : List[Int]) => construct_term(value) ::: results
      }
    }

    def symbol_to_bytes(atom : Symbol) : List[Int] = {
      val atomString = atom.toString()
```

```
  val size = int_to_bytes(2, atomString.length - 1)
  size ::: atomString.toList.drop(1).map(char => char.toInt)
}

def long_to_bytes(value : Long) : List[Int] = {
  Range(0, 8).foldLeft(List[Int](), value) {
    (accumulator, _) =>
      val (workingList, remainder) = accumulator
      val next_acc = (remainder % 256).toInt
      (next_acc :: workingList, remainder >>> 8)
  }._1.reverse
}

def int_to_bytes(length : Int, value : Int) : List[Int] = {
  Range(0, length).foldLeft(List[Int](), value) {
    (accumulator, _) =>
      val (workingList, remainder) = accumulator
      ((remainder % 256) :: workingList, remainder >>> 8)
  }._1
}

def binary_to_term(data : List[Byte]) : Tuple2[Any, List[Byte]] = {
  val version = data.head
  version match {
    case -125 => extract_term(data.tail)
    case _ => (Symbol("UNKNOWN"), List.empty)
  }
}

def extract_term(data : List[Byte]) : Tuple2[Any, List[Byte]] = {
  if (data.isEmpty)
    null
  else
    data.head match {
      case SMALL_INTEGER_EXT => getByte(data.tail)
      case INTEGER_EXT => getInteger(data.tail)
      case SMALL_BIG_EXT => getLong(data.tail)
      case SMALL_TUPLE_EXT => getTuple(data.tail)
      case ATOM_EXT => getAtom(data.tail)
      case LIST_EXT => getLargeList(data.tail)
      case STRING_EXT => getString(data.tail)
      case NIL_EXT => (List(), data.tail)
      case unknown =>
        println(s"Unknown Ext - $unknown")
        (data.head, Nil)
    }
}

def getByte(data : List[Byte]) = {
  (data.head : Int, data.tail)
}

def byte_to_int(data : List[Byte]) = {
  data.foldLeft(0)((acc, item) => {
    val unsigned_item = if (item < 0) 256 + item else item
    (acc << 8) + unsigned_item
  })
}

def getInteger(data: List[Byte]) = {
  (byte_to_int(data.take(4)), data.drop(4))
}

def getLong(data: List[Byte]) : Tuple2[Long, List[Byte]] = {
  val size = data.head
  val sign = data.tail.head
  val (bits_plus, rest) = data.splitAt(size+2)
  val bits = bits_plus.drop(2)
  (Utilities.hex2dec(bits.foldRight("")((item, acc) => {
    val item_hex = Utilities.byte2hex(item)
    acc + item_hex
  })).toLong, rest)
}

def getTuple(data : List[Byte]) = {
  val (items_as_list, remaining) = getList(1, data)
  items_as_list match {
    case a1 :: Nil => ((a1), remaining)
    case a1 :: a2 :: Nil => ((a1, a2), remaining)
    case a1 :: a2 :: a3 :: Nil => ((a1, a2, a3), remaining)
    case a1 :: a2 :: a3 :: a4 :: Nil => ((a1, a2, a3, a4), remaining)
    case _ => throw new IllegalArgumentException("Invalid length Tuple")
  }
}
```

```
    def getString(data : List[Byte]) = {
      val len = byte_to_int(data.take(2))
      (data.drop(2).take(len).map(b => b.asInstanceOf[Char]).mkString, data.drop(len + 2))
    }

    def getLargeList(data : List[Byte]) = {
      val (accumulated_term, remaining) = getList(4, data)
      if (remaining.head == NIL_EXT)
        (accumulated_term, remaining.tail)
      else {
        println("SHOULD NOT GET HERE!")
        (accumulated_term, remaining)
      }
    }

    def getList(size : Int, data : List[Byte]) = {
      val (result, remainingData) = Range(0, byte_to_int(data.take(size))).foldLeft(List[Any
           ](), data.drop(size)) {
        (accumulator, _) =>
          val(item, remainingData) = extract_term(accumulator._2)
          (item :: accumulator._1, remainingData)
      }
      (result.reverse, remainingData)
    }

    def getAtom(data : List[Byte]) = {
      val (result, remainingData) = getString(data)
      (Symbol(result), remainingData)
    }

    def receive_packet(socket: DatagramSocket): Any = {
      val in_buf = new Array[Byte](5000)
      var in_packet = new DatagramPacket(in_buf, in_buf.length)

      socket.receive(in_packet)
      val (term, remaining) = binary_to_term(in_buf.toList)
      term
    }
  }
}

class VirtualMachineActor extends Actor with ActorLogging {
  import VirtualMachineActor._

  val sock = new DatagramSocket()
  var buf = new Array[Byte](Udp_Client.bufsize)
  val in_buf = new Array[Byte](5000)
  var in_packet = new DatagramPacket(in_buf, in_buf.length)
  val local = InetAddress.getByName("localhost")

  def receive = {
    case ProcessNextStatement =>
      log.debug("PROCESS NEXT STATEMENT")
      val out = Udp_Client.term_to_binary(Udp_Client.PROCESS_NEXT)
      val out_packet = new DatagramPacket(out, out.length, local, Udp_Client.port)
      sock.send(out_packet)
    case SendData(data) =>
      log.debug(s"SEND DATA $data")
      val out = Udp_Client.term_to_binary(data)
      val out_packet = new DatagramPacket(out, out.length, local, Udp_Client.port)
      sock.send(out_packet)
      handle_response(Udp_Client.receive_packet(sock))
  }

  def handle_message(message : Any) = {
    message match {
      case ('display, txt : String) =>
        context.actorSelection("/user/consoleUpdateActor") ! ConsolePanel.DisplayText(txt)
      case 'halt =>
        context.actorSelection("/user/controlsUpdateActor") ! ControlsPanel.ProgramFinished
      case ('memory_change, changes : List[Tuple2[Any, Integer]]) =>
        changes.foreach {
          case (location: Integer, value) =>
            context.actorSelection("/user/memoryUpdateActor") ! MemoryPanel.UpdateAddress(
                 location.toLong, value.byteValue)
          case (location: Long, value) =>
            context.actorSelection("/user/memoryUpdateActor") ! MemoryPanel.UpdateAddress(
                 location, value.byteValue)
        }
        log.debug("Update memory locations")
      case msg =>
        log.debug(msg.toString)
    }
```

```
  }

  def convert_to_long(orignal_list : List[Tuple2[Any, Any]]) : List[Tuple2[Any, Long]] = {
    orignal_list.map {
      case (reg, value: Integer) =>
        (reg, value.longValue())
      case (reg, value: Long) =>
        (reg, value)
    }
  }

  def handle_response(response : Any) = {
    response match {
      case 'finished =>
        log.debug("FINISHED")
      case ('all_registers , registers : List[Tuple2[Any, Any]]) =>
        log.debug(s"ALL REGISTERS RETURNED SIZE = ${registers.size}")
        context.actorSelection("/user/registersUpdateActor") ! RegisterPanel.FullRegisterSet(
             convert_to_long(registers))
      case ('updates , (statement : String, registers : List[Tuple2[Any, Any]], messages :
           List[Any])) =>
        log.debug(s"REGISTERS UPDATES RETURNED SIZE = ${registers.size}")
        log.debug(s"THE NUMBER OF ADDITION MESSAGES ARE = ${messages.size}")
        log.debug(s"The command executed was $statement")
        context.actorSelection("/user/mainStateUpdateActor") ! RecordStatement(statement)
        context.actorSelection("/user/memoryUpdateActor") ! StartNewSetOfUpdates
        context.actorSelection("/user/registersUpdateActor") ! RegisterPanel.UpdatedRegisters(
             convert_to_long(registers))
        messages.foreach(handle_message)
        if (messages.count(_ == 'halt) == 0)
          context.actorSelection("/user/workerActor") ! Automate
        context.actorSelection("/user/memoryUpdateActor") ! RefreshTable
      case _ =>
        log.debug(s"UNKNOWN RESPONSE $response")
    }
  }
}
```

B.3 Virtual Machine

B.3.1 Branch

```
%%%-------------------------------------------------------------------
%%% @author steveedmans
%%% @copyright (C) 2015, <COMPANY>
%%% @doc
%%%
%%% @end
%%% Created : 08. Sep 2015 20:04
%%%-------------------------------------------------------------------
-module(branch).
-author("Steve Edmans").

%% API
-export([bn/1, bz/1, bp/1, bod/1, bnn/1, bnz/1, bnp/1, bev/1, jmp/1]).
-export([branch_forward/5, branch_backward/5]).

branch_forward(Fun, PC, RX, Address, Stmt) ->
  NewPC = case Fun(RX) of
            false -> PC + 4;
            true  -> PC + (4 * Address)
          end,
  {Stmt, [{pc, NewPC}], []}.

branch_backward(Fun, PC, RX, Address, Stmt) ->
  NewPC = case Fun(RX) of
            false -> PC + 4;
            true  -> PC - (4 * Address)
          end,
  {Stmt, [{pc, NewPC}], []}.

bn(RX) ->
  RXVal = registers:query_adjusted_register(RX),
  RXVal < 0.

bz(RX) ->
  RXVal = registers:query_adjusted_register(RX),
  RXVal == 0.
```

```
bp(RX) ->
  RXVal = registers:query_adjusted_register(RX),
  RXVal > 0.

bod(RX) ->
  RXVal = registers:query_adjusted_register(RX),
  (RXVal rem 2) /= 0.

bnn(RX) ->
  bn(RX) == false.

bnz(RX) ->
  bz(RX) == false.

bnp(RX) ->
  bp(RX) == false.

bev(RX) ->
  bod(RX) == false.

jmp(_RX) -> true.
```

B.3.2 Communication

```
%%%-------------------------------------------------------------------
%%% @author sedmans
%%% @copyright (C) 2014, <COMPANY>
%%% @doc
%%%
%%% @end
%%% Created : 25. Feb 2014 13:45
%%%-------------------------------------------------------------------
-module(comm).
-author("sedmans").

%% API
-export([start_vm/0, process_next_statement/0]).

-define(PORT, 4000).

start_vm () ->
  registers:init(),
  case whereis(register_ra) of
    undefined -> true;
    _Pid -> register_ra ! stop
  end,
  register_ra:start(),
  memory:start_link(),
  start_server().

start_server() ->
  case gen_udp:open(?PORT, [binary]) of
    {ok, Socket} -> loop(Socket);
    Error -> erlang:display(Error)
  end.


loop(Socket) ->
  receive
    {udp, Socket, Host, Port, Bin} ->
      N = binary_to_term(Bin),
      case process_message(N) of
        {updates, Updates} ->
          RV = {updates, Updates},
          TTB = term_to_binary(RV),
          gen_udp:send(Socket, Host, Port, TTB);
        {all_registers, Registers} ->
          gen_udp:send(Socket, Host, Port, term_to_binary({all_registers, Registers}));
        _ ->
          gen_udp:send(Socket, Host, Port, term_to_binary(finished))
      end,
      loop(Socket);
    stop ->
      gen_udp:close(Socket),
      erlang:display(registers:contents()),
      registers:stop(),
      register_ra ! stop,
      memory:contents(),
      memory:stop()
  end.

process_message(process_next) ->
```

```
  {updates, process_next_statement()};
process_message(stop) ->
  self() ! stop,
  stopping;
process_message({program, Code}) ->
  memory:store_program(Code),
  storing;
process_message({registers, Registers}) ->
  registers:stop(),
  registers:init(),
  lists:map(fun({X, Y}) -> {set_unadjusted_register(X, Y)} end, Registers),
  Pc = registers:query_register(255),
  registers:set_register(pc, Pc),
  updating;
process_message(get_all_registers) ->
  {all_registers, registers:contents()};
process_message(N) ->
  io:format("Unrecognized Message ~w~n", [N]),
  unknown.

set_unadjusted_register(R, V) ->
  registers:set_register(R, V).

process_next_statement() ->
  next_statement().

get_special_registers() ->
  Ra = get_register_ra(),
  [Ra].

get_register_ra() ->
  register_ra ! {self(), value},
  receive
    Ra ->
      {rA, Ra}
  end.

next_statement() ->
  PC = registers:query_register(pc),
  FullOpCode = memory:get_byte(PC),
  {Code, Updates, Msgs} = cpu:execute(FullOpCode, PC),
  Upd = get_special_registers(),
  FullUpdates = Updates ++ Upd,
  lists:map(fun({R, V}) -> {registers:set_register(R, V)} end, FullUpdates),
  io:format("We processed ~s~n", [Code]),
  {Code, FullUpdates, Msgs}.
```

B.3.3 CPU Main Header

```
%%%-------------------------------------------------------------------
%%% @author steveedmans
%%% @copyright (C) 2015, <COMPANY>
%%% @doc
%%% This header file contains all of the definitions used by the cpu
%%% module
%%% @end
%%% Created : 18. Aug 2015 12:08
%%%-------------------------------------------------------------------
-author("steveedmans").

-define(TRAP,   0).
-define(FCMP,   16#01).
-define(FUN,    16#02).
-define(FEQL,   16#03).
-define(FADD,   16#04).
-define(FIX,    16#05).
-define(FSUB,   16#06).
-define(FIXU,   16#07).
-define(FLOT,   16#08).
-define(FLOTU,  16#0A).
-define(FLOTUI, 16#0B).
-define(SFLOT,  16#0C).
-define(SFLOTI, 16#0D).
-define(SFLOTU, 16#0E).
-define(SFLOTUI,16#0F).
-define(FMUL,   16#10).
-define(FCMPE,  16#11).
-define(FUNE,   16#12).
-define(FEQLE,  16#13).
-define(FDIV,   16#14).
-define(FSQRT,  16#15).
-define(FREM,   16#16).
```

```
-define(FINT,    16#17).
-define(MUL,     16#18).
-define(MULI,    16#19).
-define(MULU,    16#1A).
-define(MULUI,   16#1B).
-define(DIV,     16#1C).
-define(DIVI,    16#1D).
-define(DIVU,    16#1E).
-define(DIVUI,   16#1F).
-define(ADD,     16#20).
-define(ADDI,    16#21).
-define(ADDU,    16#22).
-define(ADDUI,   16#23).
-define(SUB,     16#24).
-define(SUBI,    16#25).
-define(SUBU,    16#26).
-define(SUBUI,   16#27).
-define(ADDU2,   16#28).
-define(ADDU2I,  16#29).
-define(ADDU4,   16#2A).
-define(ADDU4I,  16#2B).
-define(ADDU8,   16#2C).
-define(ADDU8I,  16#2D).
-define(ADDU16,  16#2E).
-define(ADDU16I,16#2F).
-define(CMP,     16#30).
-define(CMPI,    16#31).
-define(CMPU,    16#32).
-define(CMPUI,   16#33).
-define(NEG,     16#34).
-define(NEGI,    16#35).
-define(NEGU,    16#36).
-define(NEGUI,   16#37).
-define(SL,      16#38).
-define(SLI,     16#39).
-define(SLU,     16#3A).
-define(SLUI,    16#3B).
-define(SR,      16#3C).
-define(SRI,     16#3D).
-define(SRU,     16#3E).
-define(SRUI,    16#3F).
-define(BN,      16#40).
-define(BNB,     16#41).
-define(BZ,      16#42).
-define(BZB,     16#43).
-define(BP,      16#44).
-define(BPB,     16#45).
-define(BOD,     16#46).
-define(BODB,    16#47).
-define(BNN,     16#48).
-define(BNNB,    16#49).
-define(BNZ,     16#4A).
-define(BNZB,    16#4B).
-define(BNP,     16#4C).
-define(BNPB,    16#4D).
-define(BEV,     16#4E).
-define(BEVB,    16#4F).
-define(PBN,     16#50).
-define(PBNB,    16#51).
-define(PBZ,     16#52).
-define(PBZB,    16#53).
-define(PBP,     16#54).
-define(PBPB,    16#55).
-define(PBOD,    16#56).
-define(PBODB,   16#57).
-define(PBNN,    16#58).
-define(PBNNB,   16#59).
-define(PBNZ,    16#5A).
-define(PBNZB,   16#5B).
-define(PBNP,    16#5C).
-define(PBNPB,   16#5D).
-define(PBEV,    16#5E).
-define(PBEVB,   16#5F).
-define(CSN,     16#60).
-define(CSNI,    16#61).
-define(CSZ,     16#62).
-define(CSZI,    16#63).
-define(CSP,     16#64).
-define(CSPI,    16#65).
-define(CSOD,    16#66).
-define(CSODI,   16#67).
-define(CSNN,    16#68).
-define(CSNNI,   16#69).
-define(CSNZ,    16#6A).
```

```
-define(CSNZI,   16#6B).
-define(CSNP,    16#6C).
-define(CSNPI,   16#6D).
-define(CSEV,    16#6E).
-define(CSEVI,   16#6F).
-define(ZSN,     16#70).
-define(ZSNI,    16#71).
-define(ZSZ,     16#72).
-define(ZSZI,    16#73).
-define(ZSP,     16#74).
-define(ZSPI,    16#75).
-define(ZSOD,    16#76).
-define(ZSODI,   16#77).
-define(ZSNN,    16#78).
-define(ZSNNI,   16#79).
-define(ZSNZ,    16#7A).
-define(ZSNZI,   16#7B).
-define(ZSNP,    16#7C).
-define(ZSNPI,   16#7D).
-define(ZSEV,    16#7E).
-define(ZSEVI,   16#7F).
-define(LDB,     16#80).
-define(LDBI,    16#81).
-define(LDBU,    16#82).
-define(LDBUI,   16#83).
-define(LDW,     16#84).
-define(LDWI,    16#85).
-define(LDWU,    16#86).
-define(LDWUI,   16#87).
-define(LDT,     16#88).
-define(LDTI,    16#89).
-define(LDTU,    16#8A).
-define(LDTUI,   16#8B).
-define(LDO,     16#8C).
-define(LDOI,    16#8D).
-define(LDOU,    16#8E).
-define(LDOUI,   16#8F).
-define(LDSF,    16#90).
-define(LDSFI,   16#91).
-define(LDHT,    16#92).
-define(LDHTI,   16#93).
-define(CSWAP,   16#94).
-define(CSWAPI,  16#95).
-define(LDUNC,   16#96).
-define(LDUNCI,  16#97).
-define(LDVTS,   16#98).
-define(LDVTSI,  16#99).
-define(PRELD,   16#9A).
-define(PRELDI,  16#9B).
-define(PREGO,   16#9C).
-define(PREGOI,  16#9D).
-define(GO,      16#9E).
-define(GOI,     16#9F).
-define(STB,     16#A0).
-define(STBI,    16#A1).
-define(STBU,    16#A2).
-define(STBUI,   16#A3).
-define(STW,     16#A4).
-define(STWI,    16#A5).
-define(STWU,    16#A6).
-define(STWUI,   16#A7).
-define(STT,     16#A8).
-define(STTI,    16#A9).
-define(STTU,    16#AA).
-define(STTUI,   16#AB).
-define(STO,     16#AC).
-define(STOI,    16#AD).
-define(STOU,    16#AE).
-define(STOUI,   16#AF).
-define(STSF,    16#B0).
-define(STSFI,   16#B1).
-define(STHT,    16#B2).
-define(STHTI,   16#B3).
-define(STCO,    16#B4).
-define(STCOI,   16#B5).
-define(STUNC,   16#B6).
-define(STUNCI,  16#B7).
-define(SYNCD,   16#B8).
-define(SYNCDI,  16#B9).
-define(PREST,   16#BA).
-define(PRESTI,  16#BB).
-define(SYNCID,  16#BC).
-define(SYNCIDI,16#BD).
-define(PUSHGO,  16#BE).
```

```
-define(PUSHGOI,16#BF).
-define(OR,     16#C0).
-define(ORI,    16#C1).
-define(ORN,    16#C2).
-define(ORNI,   16#C3).
-define(NOR,    16#C4).
-define(NORI,   16#C5).
-define(XOR,    16#C6).
-define(XORI,   16#C7).
-define(AND,    16#C8).
-define(ANDI,   16#C9).
-define(ANDN,   16#CA).
-define(ANDNI,  16#CB).
-define(NAND,   16#CC).
-define(NANDI,  16#CD).
-define(NXOR,   16#CE).
-define(NXORI,  16#CF).
-define(BDIF,   16#D0).
-define(BDIFI,  16#D1).
-define(WDIF,   16#D2).
-define(WDIFI,  16#D3).
-define(TDIF,   16#D4).
-define(TDIFI,  16#D5).
-define(ODIF,   16#D6).
-define(ODIFI,  16#D7).
-define(MUX,    16#D8).
-define(MUXI,   16#D9).
-define(SADD,   16#DA).
-define(SADDI,  16#DB).
-define(MOR,    16#DC).
-define(MORI,   16#DD).
-define(MXOR,   16#DE).
-define(MXORI,  16#DF).
-define(SETH,   16#E0).
-define(SETMH,  16#E1).
-define(SETML,  16#E2).
-define(SETL,   16#E3).
-define(INCH,   16#E4).
-define(INCMH,  16#E5).
-define(INCML,  16#E6).
-define(INCL,   16#E7).
-define(ORH,    16#E8).
-define(ORMH,   16#E9).
-define(ORML,   16#EA).
-define(ORL,    16#EB).
-define(ANDNH,  16#EC).
-define(ANDNMH, 16#ED).
-define(ANDNML, 16#EE).
-define(ANDNL,  16#EF).
-define(JMP,    16#F0).
-define(JMPB,   16#F1).
-define(PUSHJ,  16#F2).
-define(PUSHJB, 16#F3).
-define(GETA,   16#F4).
-define(GETAB,  16#F5).
-define(PUT,    16#F6).
-define(PUTI,   16#F7).
-define(POP,    16#F8).
-define(RESUME, 16#F9).
-define(SAVE,   16#FA).
-define(UNSAVE, 16#FB).
-define(SYNC,   16#FC).
-define(SWYM,   16#FD).
-define(GET,    16#FE).
-define(TRIP,   16#FF).
```

B.3.4 CPU Execute Statment

```
%%%-------------------------------------------------------------------
%%% @author steveedmans
%%% @copyright (C) 2015, <COMPANY>
%%% @doc
%%% This header file contains the execute command
%%% @end
%%% Created : 17. Sep 2015 15:47
%%%-------------------------------------------------------------------
-author("steveedmans").

-include("cpu.hrl").

%% Determine which function we are executing
```

```
%% 00-0F

execute(?TRAP, PC) ->
  trap(PC);

%% 10-1F

execute(?MUL, PC) ->
  mul(PC);
execute(?MULI, PC) ->
  muli(PC);

execute(?MULU, PC) ->
  mulu(PC);
execute(?MULUI, PC) ->
  mului(PC);

execute(?DIV, PC) ->
  mmix_div(PC);
execute(?DIVI, PC) ->
  divi(PC);

execute(?DIVU, PC) ->
  divu(PC);
execute(?DIVUI, PC) ->
  divui(PC);

%% 20-2F

execute(?ADD, PC) ->
  add(PC);
execute(?ADDI, PC) ->
  addi(PC);

execute(?ADDU, PC) ->
  addu(PC);
execute(?ADDUI, PC) ->
  addui(PC);

execute(?SUB, PC) ->
  sub(PC);
execute(?SUBI, PC) ->
  subi(PC);

execute(?SUBU, PC) ->
  subu(PC);
execute(?SUBUI, PC) ->
  subui(PC);

execute(?ADDU2, PC) ->
  addu2(PC);
execute(?ADDU2I, PC) ->
  addu2i(PC);

execute(?ADDU4, PC) ->
  addu4(PC);
execute(?ADDU4I, PC) ->
  addu4i(PC);

execute(?ADDU8, PC) ->
  addu8(PC);
execute(?ADDU8I, PC) ->
  addu8i(PC);

execute(?ADDU16, PC) ->
  addu16(PC);
execute(?ADDU16I, PC) ->
  addu16i(PC);

%% 30-3F

execute(?CMP, PC) ->
  cmp(PC);
execute(?CMPI, PC) ->
  cmpi(PC);

execute(?NEG, PC) ->
  neg(PC);
execute(?NEGI, PC) ->
  negi(PC);

%% 40-4F

execute(?BN, PC) ->
```

```
  bn(PC);
execute(?BNB, PC) ->
  bnb(PC);
execute(?BZ, PC) ->
  bz(PC);
execute(?BZB, PC) ->
  bzb(PC);
execute(?BP, PC) ->
  bp(PC);
execute(?BPB, PC) ->
  bpb(PC);
execute(?BOD, PC) ->
  bod(PC);
execute(?BODB, PC) ->
  bodb(PC);
execute(?BNN, PC) ->
  bnn(PC);
execute(?BNNB, PC) ->
  bnnb(PC);
execute(?BNZ, PC) ->
  bnz(PC);
execute(?BNZB, PC) ->
  bnzb(PC);
execute(?BNP, PC) ->
  bnp(PC);
execute(?BNPB, PC) ->
  bnpb(PC);
execute(?BEV, PC) ->
  bev(PC);
execute(?BEVB, PC) ->
  bevb(PC);

%% 50-5F

execute(?PBN, PC) ->
  pbn(PC);
execute(?PBNB, PC) ->
  pbnb(PC);
execute(?PBZ, PC) ->
  pbz(PC);
execute(?PBZB, PC) ->
  pbzb(PC);
execute(?PBP, PC) ->
  pbp(PC);
execute(?PBPB, PC) ->
  pbpb(PC);
execute(?PBOD, PC) ->
  pbod(PC);
execute(?PBODB, PC) ->
  pbodb(PC);
execute(?PBNN, PC) ->
  pbnn(PC);
execute(?PBNNB, PC) ->
  pbnnb(PC);
execute(?PBNZ, PC) ->
  pbnz(PC);
execute(?PBNZB, PC) ->
  pbnzb(PC);
execute(?PBNP, PC) ->
  pbnp(PC);
execute(?PBNPB, PC) ->
  pbnpb(PC);
execute(?PBEV, PC) ->
  pbev(PC);
execute(?PBEVB, PC) ->
  pbevb(PC);

%% 60-6F

execute(?CSN, PC) ->
  csn(PC);
execute(?CSNI, PC) ->
  csni(PC);

execute(?CSZ, PC) ->
  csz(PC);
execute(?CSZI, PC) ->
  cszi(PC);

execute(?CSP, PC) ->
  csp(PC);
execute(?CSPI, PC) ->
  cspi(PC);
```

```
execute(?CSOD, PC) ->
  csod(PC);
execute(?CSODI, PC) ->
  csodi(PC);

execute(?CSNN, PC) ->
  csnn(PC);
execute(?CSNNI, PC) ->
  csnni(PC);

execute(?CSNZ, PC) ->
  csnz(PC);
execute(?CSNZI, PC) ->
  csnzi(PC);

execute(?CSNP, PC) ->
  csnp(PC);
execute(?CSNPI, PC) ->
  csnpi(PC);

execute(?CSEV, PC) ->
  csev(PC);
execute(?CSEVI, PC) ->
  csevi(PC);

%% 70-7F

execute(?ZSN, PC) ->
  zsn(PC);
execute(?ZSNI, PC) ->
  zsni(PC);

execute(?ZSZ, PC) ->
  zsz(PC);
execute(?ZSZI, PC) ->
  zszi(PC);

execute(?ZSP, PC) ->
  zsp(PC);
execute(?ZSPI, PC) ->
  zspi(PC);

execute(?ZSOD, PC) ->
  zsod(PC);
execute(?ZSODI, PC) ->
  zsodi(PC);

execute(?ZSNN, PC) ->
  zsnn(PC);
execute(?ZSNNI, PC) ->
  zsnni(PC);

execute(?ZSNZ, PC) ->
  zsnz(PC);
execute(?ZSNZI, PC) ->
  zsnzi(PC);

execute(?ZSNP, PC) ->
  zsnp(PC);
execute(?ZSNPI, PC) ->
  zsnpi(PC);

execute(?ZSEV, PC) ->
  zsev(PC);
execute(?ZSEVI, PC) ->
  zsevi(PC);

%% 80-8F

execute(?LDB, PC) ->
  ldb(PC);
execute(?LDBI, PC) ->
  ldbi(PC);
execute(?LDBU, PC) ->
  ldbu(PC);
execute(?LDBUI, PC) ->
  ldbui(PC);
execute(?LDW, PC) ->
  ldw(PC);
execute(?LDWI, PC) ->
  ldwi(PC);
execute(?LDWU, PC) ->
  ldwu(PC);
execute(?LDWUI, PC) ->
```

```
  ldwui(PC);
execute(?LDT, PC) ->
  ldt(PC);
execute(?LDTI, PC) ->
  ldti(PC);
execute(?LDTU, PC) ->
  ldtu(PC);
execute(?LDTUI, PC) ->
  ldtui(PC);
execute(?LDO, PC) ->
  ldo(PC);
execute(?LDOI, PC) ->
  ldoi(PC);
execute(?LDOU, PC) ->
  ldou(PC);
execute(?LDOUI, PC) ->
  ldoui(PC);

%% 90-9F

%% A0-AF

execute(?STB, PC) ->
  stb(PC);
execute(?STBI, PC) ->
  stbi(PC);
execute(?STBU, PC) ->
  stbu(PC);
execute(?STBUI, PC) ->
  stbui(PC);
execute(?STWU, PC) ->
  stwu(PC);
execute(?STWUI, PC) ->
  stwui(PC);

execute(?STOU, PC) ->
  stou(PC);
execute(?STOUI, PC) ->
  stoui(PC);

%% B0-BF

%% C0-CF

execute(?OR, PC) ->
  mmix_or(PC);
execute(?ORI, PC) ->
  ori(PC);

%% D0-DF

%% E0-EF

execute(?SETL, PC) ->
  setl(PC);
execute(?INCL, PC) ->
  incl(PC);
execute(?ORH, PC) ->
  orh(PC);
execute(?ORMH, PC) ->
  ormh(PC);
execute(?ORML, PC) ->
  orml(PC);
execute(?ORL, PC) ->
  orl(PC);

%% F0-FF

execute(?JMP, PC) ->
  jmp(PC);
execute(?JMPB, PC) ->
  jmpb(PC);
execute(?GET, PC) ->
  mmix_get(PC);

execute(OpCode, _PC) ->
  io:format("We encountered a command that we do not recognized ~w~n", [OpCode]),
  {"ERROR", [], []}.
```

B.3.5 CPU

```
%%%-------------------------------------------------------------------
```

```
%%% @author steveedmans
%%% @copyright (C) 2015, <COMPANY>
%%% @doc
%%%
%%% @end
%%% Created : 18. Aug 2015 16:56
%%%-------------------------------------------------------------------
-module(cpu).
-author("steveedmans").

%% API
-export([execute/2]).

-include("cpu_execute.hrl").

next_command(PC) ->
  {pc, PC + 4}.

%% Execute the individual instructions

%% 00-0F

trap(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Msgs = trap:process_trap(RX, RY, RZ),
  Updates = [next_command(PC)],
  Stmt = lists:flatten(io_lib:format("TRAP ~B, ~B, ~B", [RX, RY, RZ])),
  {Stmt, Updates, Msgs}.

%% 10-1F

mul(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_adjusted_register(RZ),
  Stmt = lists:flatten(io_lib:format("MUL $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  muli(PC, Stmt, RX, RY, RZVal).
muli(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("MULI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  muli(PC, Stmt, RX, RY, RZ).
muli(PC, Stmt, RX, RY, Z) ->
  RYVal = registers:query_adjusted_register(RY),
  NV = Z * RYVal,
  MinV = utilities:min_value(),
  MaxV = utilities:max_value(),
  NV2 = if
    NV > MaxV ->
      register_ra ! {event, overflow},
      NV rem MaxV;
    NV < MinV ->
      register_ra ! {event, overflow},
      NV rem MaxV;
    true      -> NV
  end,
  {Stmt, [{RX, NV2}, next_command(PC)], []}.

mulu(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("MULU $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  mului(PC, Stmt, RX, RY, RZVal).
mului(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("MULUI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  mului(PC, Stmt, RX, RY, RZ).
mului(PC, Stmt, RX, RY, Z) ->
  RYVal = registers:query_register(RY),
  NV = Z * RYVal,
  SV = NV rem (utilities:minus_one() + 1),
  HV = NV div (utilities:minus_one() + 1),
  {Stmt, [{RX, SV}, [rH, HV], next_command(PC)], []}.

mmix_div(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_adjusted_register(RZ),
  Stmt = lists:flatten(io_lib:format("DIV $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  divi(PC, Stmt, RX, RY, RZVal).
divi(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("DIVI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  divi(PC, Stmt, RX, RY, RZ).
divi(PC, Stmt, RX, RY, Z) ->
  RYVal = registers:query_adjusted_register(RY),
  Updates = [next_command(PC)],
```

```
  if
    Z == 0 ->
      register_ra ! {event, divide_check},
      XtraUpdates = [{RX, 0}, {rR, RYVal}];
    true ->
      Quot  = RYVal div Z,
      Rem   = RYVal rem Z,
      XtraUpdates = [{RX, Quot}, {rR, Rem}]
  end,
  FullUpdates = Updates ++ XtraUpdates,
  {Stmt, FullUpdates, []}.

divu(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("DIVU $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  divui(PC, Stmt, RX, RY, RZVal).
divui(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("DIVUI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  divui(PC, Stmt, RX, RY, RZ).
divui(PC, Stmt, RX, RY, Z) ->
  RYVal = registers:query_register(RY),
  Updates = [next_command(PC)],
  if
    Z == 0 ->
      register_ra ! {event, divide_check},
      XtraUpdates = [{RX, 0}, {rR, RYVal}];
    true ->
      Quot  = RYVal div Z,
      Rem   = RYVal rem Z,
      XtraUpdates = [{RX, Quot}, {rR, Rem}]
  end,
  FullUpdates = Updates ++ XtraUpdates,
  {Stmt, FullUpdates, []}.

%% 20-2F

add(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_adjusted_register(RZ),
  Stmt = lists:flatten(io_lib:format("ADD $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  addi(PC, Stmt, RX, RY, RZVal).
addi(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("ADDI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  addi(PC, Stmt, RX, RY, RZ).
addi(PC, Stmt, RX, RY, Z) ->
  RYVal = registers:query_adjusted_register(RY),
  {_Overflow, Result} = add_values(RYVal, Z),
  {Stmt, [{RX, Result}, next_command(PC)], []}.

addu(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("ADDU $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  addui(PC, Stmt, RX, RY, RZVal).
addui(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("ADDUI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  addui(PC, Stmt, RX, RY, Z).
addui(PC, Stmt, RX, RY, RZVal) ->
  {Overflow, NewValue} = immediate_address(RY, RZVal),
  Updates = [{RX, NewValue}, next_command(PC)],
  register_ra ! {remove, overflow},
  NewList = case Overflow of
              overflow ->
                [{rA, 1} | Updates];
              _ ->
                Updates
            end,
  {Stmt, NewList, []}.

sub(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("SUB $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  subi(PC, Stmt, RX, RY, RZVal).
subi(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("SUBI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  subi(PC, Stmt, RX, RY, RZ).
subi(PC, Stmt, RX, RY, Z) ->
  RYVal = registers:query_register(RY),
```

```
  ZNeg = utilities:twos_complement(Z),
  {_Overflow, Subtraction} = add_values(RYVal, ZNeg),
  {Stmt, [{RX, Subtraction}, next_command(PC)], []}.

subu(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("SUBU $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  Result = subi(PC, Stmt, RX, RY, RZVal),
  register_ra ! {remove, overflow},
  Result.
subui(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("SUBUI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  Result = subi(PC, Stmt, RX, RY, RZ),
  register_ra ! {remove, overflow},
  Result.

addu2(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_adjusted_register(RZ),
  Stmt = lists:flatten(io_lib:format("2ADDU $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  addu2i(PC, Stmt, RX, RY, RZVal).
addu2i(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("2ADDUI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  addu2i(PC, Stmt, RX, RY, RZ).
addu2i(PC, Stmt, RX, RY, Z) ->
  RYVal = 2 * registers:query_adjusted_register(RY),
  {_Overflow, Result} = add_values(RYVal, Z),
  {Stmt, [{RX, Result}, next_command(PC)], []}.

addu4(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_adjusted_register(RZ),
  Stmt = lists:flatten(io_lib:format("4ADDU $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  addu4i(PC, Stmt, RX, RY, RZVal).
addu4i(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("4ADDUI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  addu4i(PC, Stmt, RX, RY, RZ).
addu4i(PC, Stmt, RX, RY, Z) ->
  RYVal = 4 * registers:query_adjusted_register(RY),
  {_Overflow, Result} = add_values(RYVal, Z),
  {Stmt, [{RX, Result}, next_command(PC)], []}.

addu8(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_adjusted_register(RZ),
  Stmt = lists:flatten(io_lib:format("8ADDU $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  addu8i(PC, Stmt, RX, RY, RZVal).
addu8i(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("8ADDUI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  addu8i(PC, Stmt, RX, RY, RZ).
addu8i(PC, Stmt, RX, RY, Z) ->
  RYVal = 8 * registers:query_adjusted_register(RY),
  {_Overflow, Result} = add_values(RYVal, Z),
  {Stmt, [{RX, Result}, next_command(PC)], []}.

addu16(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_adjusted_register(RZ),
  Stmt = lists:flatten(io_lib:format("16ADDU $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  addu16i(PC, Stmt, RX, RY, RZVal).
addu16i(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("16ADDUI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  addu16i(PC, Stmt, RX, RY, RZ).
addu16i(PC, Stmt, RX, RY, Z) ->
  RYVal = 16 * registers:query_adjusted_register(RY),
  {_Overflow, Result} = add_values(RYVal, Z),
  {Stmt, [{RX, Result}, next_command(PC)], []}.

%% 30-3F

cmp(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("CMP $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  cmpi(PC, Stmt, RX, RY, RZVal).
cmpi(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("CMPI $~.16B, $~.16B, ~B", [RX, RY, Z])),
```

```
  cmpi(PC, Stmt, RX, RY, Z).
cmpi(PC, Stmt, RX, RY, Z) ->
  RYVal = registers:query_register(RY),
  NV = if
    RYVal <  Z -> utilities:minus_one();
    RYVal >  Z -> 1;
    RYVal == Z -> 0
  end,
  {Stmt, [{RX, NV}, next_command(PC)], []}.

neg(PC) ->
  {RX, Y, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("NEG $~.16B, ~B, $~.16B", [RX, Y, RZ])),
  negi(PC, Stmt, RX, Y, RZVal).
negi(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("NEGI $~.16B, ~.B, ~B", [RX, Y, Z])),
  negi(PC, Stmt, RX, Y, Z).
negi(PC, Stmt, RX, Y, Z) ->
  Diff = Y - Z,
  NV = if
         Diff < 0 -> utilities:minus_one() + Diff + 1;
         true     -> Diff
       end,
  {Stmt, [{RX, NV}, next_command(PC)], []}.

%% 40-4F

bn(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("BN $~.16B, ~B", [RX, Address])),
  branch:branch_forward(fun branch:bn/1, PC, RX, Address, Stmt).
bnb(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("BNB $~.16B, ~B", [RX, Address])),
  branch:branch_backward(fun branch:bn/1, PC, RX, Address, Stmt).

bz(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("BZ $~.16B, ~B", [RX, Address])),
  branch:branch_forward(fun branch:bz/1, PC, RX, Address, Stmt).
bzb(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("BZB $~.16B, ~B", [RX, Address])),
  branch:branch_backward(fun branch:bz/1, PC, RX, Address, Stmt).

bp(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("BP $~.16B, ~B", [RX, Address])),
  branch:branch_forward(fun branch:bp/1, PC, RX, Address, Stmt).
bpb(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("BPB $~.16B, ~B", [RX, Address])),
  branch:branch_backward(fun branch:bp/1, PC, RX, Address, Stmt).

bod(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("BOD $~.16B, ~B", [RX, Address])),
  branch:branch_forward(fun branch:bod/1, PC, RX, Address, Stmt).
bodb(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("BODB $~.16B, ~B", [RX, Address])),
  branch:branch_backward(fun branch:bod/1, PC, RX, Address, Stmt).

bnn(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("BNN $~.16B, ~B", [RX, Address])),
  branch:branch_forward(fun branch:bnn/1, PC, RX, Address, Stmt).
bnnb(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("BNNB $~.16B, ~B", [RX, Address])),
  branch:branch_backward(fun branch:bnn/1, PC, RX, Address, Stmt).
```

```
bnz(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("BNZ $~.16B, ~B", [RX, Address])),
  branch:branch_forward(fun branch:bnz/1, PC, RX, Address, Stmt).
bnzb(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("BNZB $~.16B, ~B", [RX, Address])),
  branch:branch_backward(fun branch:bnz/1, PC, RX, Address, Stmt).

bnp(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("BNP $~.16B, ~B", [RX, Address])),
  branch:branch_forward(fun branch:bnp/1, PC, RX, Address, Stmt).
bnpb(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("BNPB $~.16B, ~B", [RX, Address])),
  branch:branch_backward(fun branch:bnp/1, PC, RX, Address, Stmt).

bev(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("BEV $~.16B, ~B", [RX, Address])),
  branch:branch_forward(fun branch:bev/1, PC, RX, Address, Stmt).
bevb(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("BEVB $~.16B, ~B", [RX, Address])),
  branch:branch_backward(fun branch:bev/1, PC, RX, Address, Stmt).

%% 50-5F

pbn(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("PBN $~.16B, ~B", [RX, Address])),
  branch:branch_forward(fun branch:bn/1, PC, RX, Address, Stmt).
pbnb(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("PBNB $~.16B, ~B", [RX, Address])),
  branch:branch_backward(fun branch:bn/1, PC, RX, Address, Stmt).

pbz(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("PBZ $~.16B, ~B", [RX, Address])),
  branch:branch_forward(fun branch:bz/1, PC, RX, Address, Stmt).
pbzb(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("PBZB $~.16B, ~B", [RX, Address])),
  branch:branch_backward(fun branch:bz/1, PC, RX, Address, Stmt).

pbp(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("PBP $~.16B, ~B", [RX, Address])),
  branch:branch_forward(fun branch:bp/1, PC, RX, Address, Stmt).
pbpb(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("PBPB $~.16B, ~B", [RX, Address])),
  branch:branch_backward(fun branch:bp/1, PC, RX, Address, Stmt).

pbod(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("PBOD $~.16B, ~B", [RX, Address])),
  branch:branch_forward(fun branch:bod/1, PC, RX, Address, Stmt).
pbodb(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("PBODB $~.16B, ~B", [RX, Address])),
  branch:branch_backward(fun branch:bod/1, PC, RX, Address, Stmt).

pbnn(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("PBNN $~.16B, ~B", [RX, Address])),
  branch:branch_forward(fun branch:bnn/1, PC, RX, Address, Stmt).
```

```
pbnnb(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("PBNNB $~.16B, ~B", [RX, Address])),
  branch:branch_backward(fun branch:bnn/1, PC, RX, Address, Stmt).

pbnz(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("PBNZ $~.16B, ~B", [RX, Address])),
  branch:branch_forward(fun branch:bnz/1, PC, RX, Address, Stmt).
pbnzb(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("PBNZB $~.16B, ~B", [RX, Address])),
  branch:branch_backward(fun branch:bnz/1, PC, RX, Address, Stmt).

pbnp(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("PBNP $~.16B, ~B", [RX, Address])),
  branch:branch_forward(fun branch:bnp/1, PC, RX, Address, Stmt).
pbnpb(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("PBNPB $~.16B, ~B", [RX, Address])),
  branch:branch_backward(fun branch:bnp/1, PC, RX, Address, Stmt).

pbev(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("PBEV $~.16B, ~B", [RX, Address])),
  branch:branch_forward(fun branch:bev/1, PC, RX, Address, Stmt).
pbevb(PC) ->
  {RX, Y, Z} = three_operands(PC),
  Address = rval(Y, Z),
  Stmt = lists:flatten(io_lib:format("PBEVB $~.16B, ~B", [RX, Address])),
  branch:branch_backward(fun branch:bev/1, PC, RX, Address, Stmt).

%% 60-6F

csn(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("CSN $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  cs(fun branch:bn/1, PC, Stmt, RX, RY, RZVal).
csni(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("CSNI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  cs(fun branch:bn/1, PC, Stmt, RX, RY, Z).

csz(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("CSZ $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  cs(fun branch:bz/1, PC, Stmt, RX, RY, RZVal).
cszi(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("CSZI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  cs(fun branch:bz/1, PC, Stmt, RX, RY, Z).

csp(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("CSP $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  cs(fun branch:bp/1, PC, Stmt, RX, RY, RZVal).
cspi(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("CSPI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  cs(fun branch:bp/1, PC, Stmt, RX, RY, Z).

csod(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("CSOD $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  cs(fun branch:bod/1, PC, Stmt, RX, RY, RZVal).
csodi(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("CSODI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  cs(fun branch:bod/1, PC, Stmt, RX, RY, Z).

csnn(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
```

```
  Stmt = lists:flatten(io_lib:format("CSNN $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  cs(fun branch:bnn/1, PC, Stmt, RX, RY, RZVal).
csnni(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("CSNNI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  cs(fun branch:bnn/1, PC, Stmt, RX, RY, Z).

csnz(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("CSNZ $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  cs(fun branch:bnz/1, PC, Stmt, RX, RY, RZVal).
csnzi(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("CSNZI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  cs(fun branch:bnz/1, PC, Stmt, RX, RY, Z).

csnp(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("CSNP $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  cs(fun branch:bnp/1, PC, Stmt, RX, RY, RZVal).
csnpi(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("CSNPI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  cs(fun branch:bnp/1, PC, Stmt, RX, RY, Z).

csev(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("CSEV $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  cs(fun branch:bev/1, PC, Stmt, RX, RY, RZVal).
csevi(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("CSEVI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  cs(fun branch:bev/1, PC, Stmt, RX, RY, Z).

cs(Fun, PC, Stmt, RX, RY, Z) ->
  Updates = case Fun(RY) of
              true  -> [{RX, Z}, next_command(PC)];
              false -> [next_command(PC)]
            end,
  {Stmt, Updates, []}.

%% 70-7F

zsn(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("CSN $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  zs(fun branch:bn/1, PC, Stmt, RX, RY, RZVal).
zsni(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("CSNI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  zs(fun branch:bn/1, PC, Stmt, RX, RY, Z).

zsz(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("CSZ $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  zs(fun branch:bz/1, PC, Stmt, RX, RY, RZVal).
zszi(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("CSZI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  zs(fun branch:bz/1, PC, Stmt, RX, RY, Z).

zsp(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("CSP $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  zs(fun branch:bp/1, PC, Stmt, RX, RY, RZVal).
zspi(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("CSPI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  zs(fun branch:bp/1, PC, Stmt, RX, RY, Z).

zsod(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("CSOD $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  zs(fun branch:bod/1, PC, Stmt, RX, RY, RZVal).
zsodi(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("CSODI $~.16B, $~.16B, ~B", [RX, RY, Z])),
```

```
  zs(fun branch:bod/1, PC, Stmt, RX, RY, Z).

zsnn(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("CSNN $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  zs(fun branch:bnn/1, PC, Stmt, RX, RY, RZVal).
zsnni(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("CSNNI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  zs(fun branch:bnn/1, PC, Stmt, RX, RY, Z).

zsnz(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("CSNZ $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  zs(fun branch:bnz/1, PC, Stmt, RX, RY, RZVal).
zsnzi(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("CSNZI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  zs(fun branch:bnz/1, PC, Stmt, RX, RY, Z).

zsnp(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("CSNP $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  zs(fun branch:bnp/1, PC, Stmt, RX, RY, RZVal).
zsnpi(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("CSNPI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  zs(fun branch:bnp/1, PC, Stmt, RX, RY, Z).

zsev(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("CSEV $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  zs(fun branch:bev/1, PC, Stmt, RX, RY, RZVal).
zsevi(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("CSEVI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  zs(fun branch:bev/1, PC, Stmt, RX, RY, Z).

zs(Fun, PC, Stmt, RX, RY, Z) ->
  Updates = case Fun(RY) of
              true  -> [{RX, Z}, next_command(PC)];
              false -> [{RX, 0}, next_command(PC)]
            end,
  {Stmt, Updates, []}.

%% 80-8F

ldb(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("LDB $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  ldbi(PC, Stmt, RX, RY, RZVal).
ldbi(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("LDBI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  ldbi(PC, Stmt, RX, RY, RZ).
ldbi(PC, Stmt, RX, RY, Z) ->
  {_Overflow, Address} = immediate_address(RY, Z),
  Value = memory:get_byte(Address),
  SignedValue = if
                  Value > 127 -> Value - 256;
                  true        -> Value
                end,
  {Stmt, [{RX, SignedValue}, next_command(PC)], []}.

ldbu(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("LDBU $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  ldbui(PC, Stmt, RX, RY, RZVal).
ldbui(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("LDBUI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  ldbui(PC, Stmt, RX, RY, RZ).
ldbui(PC, Stmt, RX, RY, Z) ->
  {_Overflow, Address} = immediate_address(RY, Z),
  Value = memory:get_byte(Address),
  {Stmt, [{RX, Value}, next_command(PC)], []}.

ldw(PC) ->
```

```
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("LDW $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  ldwi(PC, Stmt, RX, RY, RZVal).
ldwi(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("LDWI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  ldwi(PC, Stmt, RX, RY, RZ).
ldwi(PC, Stmt, RX, RY, Z) ->
  {_Overflow, Address} = immediate_address(RY, Z),
  Value = memory:get_wyde(Address),
  SignedValue = if
                  Value > 32767 -> Value - 32768;
                  true          -> Value
                end,
  {Stmt, [{RX, SignedValue}, next_command(PC)], []}.

ldwu(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("LDWU $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  ldwui(PC, Stmt, RX, RY, RZVal).
ldwui(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("LDWUI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  ldwui(PC, Stmt, RX, RY, RZ).
ldwui(PC, Stmt, RX, RY, Z) ->
  {_Overflow, Address} = immediate_address(RY, Z),
  Value = memory:get_wyde(Address),
  {Stmt, [{RX, Value}, next_command(PC)], []}.

ldt(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("LDT $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  ldti(PC, Stmt, RX, RY, RZVal).
ldti(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("LDTI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  ldti(PC, Stmt, RX, RY, RZ).
ldti(PC, Stmt, RX, RY, Z) ->
  {_Overflow, Address} = immediate_address(RY, Z),
  Value = memory:get_tetrabyte(Address),
  SignedValue = if
                  Value > 2147483647 -> Value - 2147483648;
                  true               -> Value
                end,
  {Stmt, [{RX, SignedValue}, next_command(PC)], []}.

ldtu(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("LDTU $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  ldtui(PC, Stmt, RX, RY, RZVal).
ldtui(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("LDTUI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  ldtui(PC, Stmt, RX, RY, RZ).
ldtui(PC, Stmt, RX, RY, Z) ->
  {_Overflow, Address} = immediate_address(RY, Z),
  Value = memory:get_tetrabyte(Address),
  {Stmt, [{RX, Value}, next_command(PC)], []}.

ldo(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("LDO $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  RZVal = registers:query_register(RZ),
  ldoui(PC, Stmt, RX, RY, RZVal).
ldoi(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("LDOI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  ldoui(PC, Stmt, RX, RY, Z).

ldou(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("LDOU $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  RZVal = registers:query_register(RZ),
  ldoui(PC, Stmt, RX, RY, RZVal).
ldoui(PC) ->
  {RX, RY, Z} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("LDOUI $~.16B, $~.16B, ~B", [RX, RY, Z])),
  ldoui(PC, Stmt, RX, RY, Z).
ldoui(PC, Stmt, RX, RY, Z) ->
  {_Overflow, Address} = immediate_address(RY, Z),
```

```
  Value = memory:get_octabyte(Address),
  io:format("Set the register ~w to ~w~n", [RX, Value]),
  {Stmt, [{RX, Value}, next_command(PC)], []}.

%% 90-9F
%% A0-AF

stb(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("STB $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  stbui(PC, Stmt, RX, RY, RZVal).
stbi(PC) ->
  {RX, RY, RZVal} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("STBI $~.16B, $~.16B, ~B", [RX, RY, RZVal])),
  stbui(PC, Stmt, RX, RY, RZVal).
stbu(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("STBU $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  Result = stbui(PC, Stmt, RX, RY, RZVal),
  register_ra ! {remove, overflow},
  Result.
stbui(PC) ->
  {RX, RY, RZVal} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("STBUI $~.16B, $~.16B, ~B", [RX, RY, RZVal])),
  Result = stbui(PC, Stmt, RX, RY, RZVal),
  register_ra ! {remove, overflow},
  Result.
stbui(PC, Stmt, RX, RY, Z) ->
  IA = immediate_address(RY, Z),
  case IA of
    {_, Location} ->
      RXVal = registers:query_register(RX),
      LSB = utilities:get_0_byte(RXVal),
      MemoryChanges = memory:set_byte(Location, LSB),
      NewMessages = [{memory_change, MemoryChanges}],
      {Stmt, [next_command(PC)], NewMessages}
  end.

stwu(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("STWU $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  stwui(PC, Stmt, RX, RY, RZVal).

stwui(PC) ->
  {RX, RY, RZVal} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("STWUI $~.16B, $~.16B, ~B", [RX, RY, RZVal])),
  stwui(PC, Stmt, RX, RY, RZVal).

stwui(PC, Stmt, RX, RY, Z) ->
  IA = immediate_address(RY, Z),
  case IA of
    {_, Location} ->
      RXVal = registers:query_register(RX),
      LSB = utilities:get_0_wyde(RXVal),
      MemoryChanges = memory:set_wyde(Location, LSB),
      NewMessages = [{memory_change, MemoryChanges}],
      {Stmt, [next_command(PC)], NewMessages}
  end.

stou(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("STOU $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  stoui(PC, Stmt, RX, RY, RZVal).
stoui(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("STOUI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  stoui(PC, Stmt, RX, RY, RZ).
stoui(PC, Stmt, RX, RY, Z) ->
  IA = immediate_address(RY, Z),
  case IA of
    {_, Location} ->
      RXVal = registers:query_register(RX),
      MemoryChanges = memory:set_octabyte(Location, RXVal),
      NewMessages = [{memory_change, MemoryChanges}],
      {Stmt, [next_command(PC)], NewMessages};
    _ -> {"STOUI Address Error", [], []}
  end.

%% B0-BF
%% C0-CF
```

```
mmix_or(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RZVal = registers:query_register(RZ),
  Stmt = lists:flatten(io_lib:format("OR $~.16B, $~.16B, $~.16B", [RX, RY, RZ])),
  ori(PC, Stmt, RX, RY, RZVal).

ori(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  Stmt = lists:flatten(io_lib:format("ORI $~.16B, $~.16B, ~B", [RX, RY, RZ])),
  ori(PC, Stmt, RX, RY, RZ).

ori(PC, Stmt, RX, RY, Z) ->
  RYVal = registers:query_register(RY),
  NVal = RYVal bor Z,
  {Stmt, [{RX, NVal}, next_command(PC)], []}.

%% D0-DF
%% E0-EF

setl(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RVal = rval(RY, RZ),
  Update = registers:set_register_lowwyde(RX, RVal),
  Stmt = lists:flatten(io_lib:format("SETL $~.16B, ~B", [RX, RVal])),
  {Stmt, [Update, next_command(PC)], []}.

incl(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RVal = rval(RY, RZ),
  Stmt = lists:flatten(io_lib:format("INCL $~.16B, ~B", [RX, RVal])),
  RXVal = registers:query_register(RX),
  {_Overflow, NV} = add_values(RXVal, RVal),
  {Stmt, [{RX, NV}, next_command(PC)], []}.

orh(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RVal = rval(RY, RZ),
  RValA = RVal bsl 48,
  RXVal = registers:query_register(RX),
  NV = RXVal bor RValA,
  Stmt = lists:flatten(io_lib:format("ORH $~.16B, ~B", [RX, RVal])),
  {Stmt, [{RX, NV}, next_command(PC)], []}.

ormh(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RVal = rval(RY, RZ),
  RValA = RVal bsl 32,
  RXVal = registers:query_register(RX),
  NV = RXVal bor RValA,
  Stmt = lists:flatten(io_lib:format("ORMH $~.16B, ~B", [RX, RVal])),
  {Stmt, [{RX, NV}, next_command(PC)], []}.

orml(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RVal = rval(RY, RZ),
  RValA = RVal bsl 16,
  RXVal = registers:query_register(RX),
  NV = RXVal bor RValA,
  Stmt = lists:flatten(io_lib:format("ORML $~.16B, ~B", [RX, RVal])),
  {Stmt, [{RX, NV}, next_command(PC)], []}.

orl(PC) ->
  {RX, RY, RZ} = three_operands(PC),
  RVal = rval(RY, RZ),
  RXVal = registers:query_register(RX),
  NV = RXVal bor RVal,
  Stmt = lists:flatten(io_lib:format("ORL $~.16B, ~B", [RX, RVal])),
  {Stmt, [{RX, NV}, next_command(PC)], []}.

%% F0-FF

jmp(PC) ->
  {X, Y, Z} = three_operands(PC),
  Address = rval(rval(X, Y), Z),
  Stmt = lists:flatten(io_lib:format("JMP ~B", [Address])),
  branch:branch_forward(fun branch:jmp/1, PC, X, Address, Stmt).

jmpb(PC) ->
  {X, Y, Z} = three_operands(PC),
  Address = rval(rval(X, Y), Z),
  Stmt = lists:flatten(io_lib:format("JMPB ~B", [Address])),
  branch:branch_backward(fun branch:jmp/1, PC, X, Address, Stmt).
```

```
mmix_get(PC) ->
  {RX, _RY, RZ} = three_operands(PC),
  SR = operand_to_special_register(RZ),
  RegVal = registers:query_register(SR),
  Stmt = lists:flatten(io_lib:format("GET $~.16B, ~w", [RX, SR])),
  {Stmt, [{RX, RegVal}, next_command(PC)], []}.

%% Utilities

operand_to_special_register(0)  -> rB;
operand_to_special_register(1)  -> rD;
operand_to_special_register(2)  -> rE;
operand_to_special_register(3)  -> rH;
operand_to_special_register(4)  -> rJ;
operand_to_special_register(5)  -> rM;
operand_to_special_register(6)  -> rR;
operand_to_special_register(7)  -> rBB;
operand_to_special_register(8)  -> rC;
operand_to_special_register(9)  -> rN;
operand_to_special_register(10) -> rO;
operand_to_special_register(11) -> rS;
operand_to_special_register(12) -> rI;
operand_to_special_register(13) -> rT;
operand_to_special_register(14) -> rTT;
operand_to_special_register(15) -> rK;
operand_to_special_register(16) -> rQ;
operand_to_special_register(17) -> rU;
operand_to_special_register(18) -> rW;
operand_to_special_register(19) -> rG;
operand_to_special_register(20) -> rL;
operand_to_special_register(21) -> rA;
operand_to_special_register(22) -> rF;
operand_to_special_register(23) -> rP;
operand_to_special_register(24) -> rW;
operand_to_special_register(25) -> rX;
operand_to_special_register(26) -> rY;
operand_to_special_register(27) -> rZ;
operand_to_special_register(28) -> rWW;
operand_to_special_register(29) -> rXX;
operand_to_special_register(30) -> rYY;
operand_to_special_register(31) -> rZZ.

rval(RY, RZ) ->
  (RY * 256) + RZ.

three_operands(PC) ->
  First = operand(PC+1),
  Second = operand(PC+2),
  Third = operand(PC+3),
  {First, Second, Third}.

operand(Location) ->
  memory:get_byte(Location).

immediate_address(RY, RZ) ->
  R1 = registers:query_register(RY),
  add_values(R1, RZ).

add_values(V1, V2) ->
  A = (V1 + V2),
  MaxMemory = utilities:minus_one(),
  if
    A > MaxMemory
      ->
        register_ra ! {event, overflow},
        {overflow,(A - (MaxMemory + 1))};
    true
      -> {no_overflow, A}
  end.
```

B.3.6 Memory

```
%%% File      : memory.erl
%%% Description : API and gen_server code to simulate the memory for our virtual machine
-module(memory).
-author("Steve Edmans").

-behaviour(gen_server).

-define(MEMORY_TABLE, memory_table).
-define(MEMORY_SERVER, memory_server).
```

```
%% API
-export([start_link/0,
  stop/0,
  store_program/1,
  get_byte/1,
  get_wyde/1,
  get_tetrabyte/1,
  get_octabyte/1,
  get_nstring/1,
  set_byte/2,
  set_wyde/2,
  set_tetrabyte/2,
  set_octabyte/2,
  contents/0]).

%% gen_server callbacks
-export([init/1,
  handle_call/3,
  handle_cast/2,
  handle_info/2,
  terminate/2,
  code_change/3]).

%%%===================================================================
%%% API
%%%===================================================================

%%--------------------------------------------------------------------
%% @doc
%% Starts the server
%%
%% @end
%%--------------------------------------------------------------------
-spec(start_link() ->
  {ok, Pid :: pid()} | ignore | {error, Reason :: term()}).
start_link() ->
  gen_server:start_link({local, ?MEMORY_SERVER}, ?MODULE, [], []).

%%--------------------------------------------------------------------
%% @doc
%% Store a new program in memory
%%
%% @end
%%--------------------------------------------------------------------
store_program(Code) ->
  gen_server:call(?MEMORY_SERVER, {store_program, Code}).

stop() ->
  gen_server:call(?MEMORY_SERVER, stop_program).

contents() ->
  gen_server:call(?MEMORY_SERVER, get_contents).

get_byte(Location) ->
  gen_server:call(?MEMORY_SERVER, {get_byte, Location}).

get_wyde(Location) ->
  gen_server:call(?MEMORY_SERVER, {get_wyde, Location}).

get_tetrabyte(Location) ->
  gen_server:call(?MEMORY_SERVER, {get_tetrabyte, Location}).

get_octabyte(Location) ->
  gen_server:call(?MEMORY_SERVER, {get_octabyte, Location}).

get_nstring(Location) ->
  gen_server:call(?MEMORY_SERVER, {get_nstring, Location}).

set_byte(Location, Value) ->
  gen_server:call(?MEMORY_SERVER, {set_byte, Location, Value}).

set_wyde(Location, Value) ->
  gen_server:call(?MEMORY_SERVER, {set_wyde, Location, Value}).

set_tetrabyte(Location, Value) ->
  gen_server:call(?MEMORY_SERVER, {set_tetrabyte, Location, Value}).

set_octabyte(Location, Value) ->
  gen_server:call(?MEMORY_SERVER, {set_octabyte, Location, Value}).

%%%===================================================================
%%% gen_server callbacks
%%%===================================================================
```

```
%%--------------------------------------------------------------------
%% @private
%% @doc
%% Initializes the server
%%
%% @spec init(Args) -> {ok, State} |
%%                     {ok, State, Timeout} |
%%                     ignore |
%%                     {stop, Reason}
%% @end
%%--------------------------------------------------------------------
-spec(init(Args :: term()) ->
  {ok, State :: term()} | {ok, State :: term(), timeout() | hibernate} |
  {stop, Reason :: term()} | ignore).
init([]) ->
  TableId = ets:new(?MEMORY_TABLE, [set, public]),
  {ok, TableId}.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% Handling call messages
%%
%% @end
%%--------------------------------------------------------------------
-spec(handle_call(Request :: term(), From :: {pid(), Tag :: term()},
    State :: term()) ->
  {reply, Reply :: term(), NewState :: term()} |
  {reply, Reply :: term(), NewState :: term(), timeout() | hibernate} |
  {noreply, NewState :: term()} |
  {noreply, NewState :: term(), timeout() | hibernate} |
  {stop, Reason :: term(), Reply :: term(), NewState :: term()} |
  {stop, Reason :: term(), NewState :: term()}).
handle_call(reset_memory, _From, TableId) ->
  clear_memory(TableId);
handle_call({store_program, Code}, _From, TableId) ->
  clear_memory(TableId),
  store_program_int(Code, TableId),
  {reply, ok, TableId};
handle_call({get_byte, Location}, _From, TableId) ->
  {reply, get_memory_location_byte(Location, TableId), TableId};
handle_call({get_wyde, Location}, _From, TableId) ->
  {reply, get_memory_location_wyde(Location, TableId), TableId};
handle_call({get_tetrabyte, Location}, _From, TableId) ->
  {reply, get_memory_location_tetrabyte(Location, TableId), TableId};
handle_call({get_octabyte, Location}, _From, TableId) ->
  {reply, get_memory_location_octabyte(Location, TableId), TableId};
handle_call({get_nstring, Location}, _From, TableId) ->
  {reply, get_memory_location_nstring(Location, TableId), TableId};
handle_call({set_byte, Location, Value}, _From, TableId) ->
  {reply, [set_byte(Location, Value, TableId)], TableId};
handle_call({set_wyde, Location, Value}, _From, TableId) ->
  {reply, set_wyde(Location, Value, TableId), TableId};
handle_call({set_octabyte, Location, Value}, _From, TableId) ->
  {reply, set_octabyte(Location, Value, TableId), TableId};
handle_call(stop_program, _From, _TableId) ->
  {stop, normal};
handle_call(get_contents, _From, TableId) ->
  erlang:display(contents(TableId)),
  {reply, ok, TableId};
handle_call(_Request, _From, State) ->
  {reply, ok, State}.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% Handling cast messages
%%
%% @end
%%--------------------------------------------------------------------
-spec(handle_cast(Request :: term(), State :: term()) ->
  {noreply, NewState :: term()} |
  {noreply, NewState :: term(), timeout() | hibernate} |
  {stop, Reason :: term(), NewState :: term()}).
handle_cast(_Request, State) ->
  {noreply, State}.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% Handling all non call/cast messages
%%
%% @spec handle_info(Info, State) -> {noreply, State} |
%%                                   {noreply, State, Timeout} |
```

```
%%                                    {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
-spec(handle_info(Info :: timeout() | term(), State :: term()) ->
  {noreply, NewState :: term()} |
  {noreply, NewState :: term(), timeout() | hibernate} |
  {stop, Reason :: term(), NewState :: term()}).
handle_info(_Info, State) ->
  {noreply, State}.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% This function is called by a gen_server when it is about to
%% terminate. It should be the opposite of Module:init/1 and do any
%% necessary cleaning up. When it returns, the gen_server terminates
%% with Reason. The return value is ignored.
%%
%% @spec terminate(Reason, State) -> void()
%% @end
%%--------------------------------------------------------------------
-spec(terminate(Reason :: (normal | shutdown | {shutdown, term()} | term()),
    State :: term()) -> term()).
terminate(_Reason, _State) ->
  ok.

%%--------------------------------------------------------------------
%% @private
%% @doc
%% Convert process state when code is changed
%%
%% @spec code_change(OldVsn, State, Extra) -> {ok, NewState}
%% @end
%%--------------------------------------------------------------------
-spec(code_change(OldVsn :: term() | {down, term()}, State :: term(),
    Extra :: term()) ->
  {ok, NewState :: term()} | {error, Reason :: term()}).
code_change(_OldVsn, State, _Extra) ->
  {ok, State}.

%%%===================================================================
%%% Internal functions
%%%===================================================================


store_program_int([], _) ->
  erlang:display("STORED PROGRAM");
store_program_int([{StartLocation, Program}|Rest], TableId) ->
  store_program(Program, StartLocation, TableId),
  store_program_int(Rest, TableId).

contents(TableId) ->
  ets:tab2list(TableId).

clear_memory(TableId) ->
  ets:delete_all_objects(TableId).

store_program([], _Location, State) ->
  {ok, State};
store_program([Entry|Rest], Location, TableId) ->
  ets:insert(TableId, {Location, Entry}),
  store_program(Rest, (Location + 1), TableId).

get_memory_location_byte(Location, TableId) ->
  case ets:lookup(TableId, Location) of
    [{_, Byte}] -> Byte;
    _ -> 0
  end.

get_memory_location_wyde(Location, TableId) ->
  AdjustedLocation = utilities:adjust_location(Location, 2),
  Byte0 = case ets:lookup(TableId, AdjustedLocation) of
            [{_, B0}] -> B0;
            _ -> 0
          end,
  Byte1 = case ets:lookup(TableId, AdjustedLocation + 1) of
            [{_, B1}] -> B1;
            _ -> 0
          end,
  Value = (Byte0 * 256) + Byte1,
  Value.

get_memory_location_tetrabyte(Location, TableId) ->
  AdjustedLocation = utilities:adjust_location(Location, 4),
```

```
  Byte0 = case ets:lookup(TableId, AdjustedLocation) of
            [{_, B0}] -> B0;
            _ -> 0
          end,
  Byte1 = case ets:lookup(TableId, AdjustedLocation + 1) of
            [{_, B1}] -> B1;
            _ -> 0
          end,
  Byte2 = case ets:lookup(TableId, AdjustedLocation + 2) of
            [{_, B2}] -> B2;
            _ -> 0
          end,
  Byte3 = case ets:lookup(TableId, AdjustedLocation + 3) of
            [{_, B3}] -> B3;
            _ -> 0
          end,
  Value = (((((Byte0 * 256) + Byte1) * 256) + Byte2) * 256) + Byte3,
  Value.

get_memory_location_octabyte(Location, TableId) ->
  AdjustedLocation = utilities:adjust_location(Location, 8),
  Byte0 = case ets:lookup(TableId, AdjustedLocation) of
            [{_, B0}] -> B0;
            _ -> 0
          end,
  Byte1 = case ets:lookup(TableId, AdjustedLocation + 1) of
            [{_, B1}] -> B1;
            _ -> 0
          end,
  Byte2 = case ets:lookup(TableId, AdjustedLocation + 2) of
            [{_, B2}] -> B2;
            _ -> 0
          end,
  Byte3 = case ets:lookup(TableId, AdjustedLocation + 3) of
            [{_, B3}] -> B3;
            _ -> 0
          end,
  Byte4 = case ets:lookup(TableId, AdjustedLocation) of
            [{_, B4}] -> B4;
            _ -> 0
          end,
  Byte5 = case ets:lookup(TableId, AdjustedLocation + 1) of
            [{_, B5}] -> B5;
            _ -> 0
          end,
  Byte6 = case ets:lookup(TableId, AdjustedLocation + 2) of
            [{_, B6}] -> B6;
            _ -> 0
          end,
  Byte7 = case ets:lookup(TableId, AdjustedLocation + 3) of
            [{_, B7}] -> B7;
            _ -> 0
          end,
  Value = (((((((((((((Byte0 * 256) + Byte1) * 256) + Byte2) * 256) + Byte3) * 256) + Byte4) *
        256) + Byte5) * 256) + Byte6) * 256) + Byte7,
  Value.

get_memory_location_nstring(Location, TableId) ->
  get_memory_location_nstring(Location, TableId, []).

get_memory_location_nstring(Location, TableId, Accumulator) ->
  CurrentByte = get_memory_location_byte(Location, TableId),
  case CurrentByte of
    0 -> lists:reverse(Accumulator);
    _ -> get_memory_location_nstring((Location + 1), TableId, [CurrentByte | Accumulator])
  end.

set_byte(Location, Value, TableId) ->
  AdjustedValue = Value rem 256,  %% Make sure we only store byte values
  ets:insert(TableId, {Location, AdjustedValue}),
  {Location, Value}.

set_wyde(Location, Value, TableId) ->
  Adjusted_Location = utilities:adjust_location(Location, 2),
  B0 = utilities:get_0_byte(Value),
  B1 = utilities:get_1_byte(Value),
  C0 = set_byte(Adjusted_Location, B1, TableId),
  C1 = set_byte((Adjusted_Location + 1), B0, TableId),
  [C0, C1].

set_octabyte(Location, Value, TableId) ->
  Adjusted_Location = utilities:adjust_location(Location, 8),
  B0 = utilities:get_0_byte(Value),
  B1 = utilities:get_1_byte(Value),
```

```
  B2 = utilities:get_2_byte(Value),
  B3 = utilities:get_3_byte(Value),
  B4 = utilities:get_4_byte(Value),
  B5 = utilities:get_5_byte(Value),
  B6 = utilities:get_6_byte(Value),
  B7 = utilities:get_7_byte(Value),
  C0 = set_byte(Adjusted_Location, B7, TableId),
  C1 = set_byte((Adjusted_Location + 1), B6, TableId),
  C2 = set_byte((Adjusted_Location + 2), B5, TableId),
  C3 = set_byte((Adjusted_Location + 3), B4, TableId),
  C4 = set_byte((Adjusted_Location + 4), B3, TableId),
  C5 = set_byte((Adjusted_Location + 5), B2, TableId),
  C6 = set_byte((Adjusted_Location + 6), B1, TableId),
  C7 = set_byte((Adjusted_Location + 7), B0, TableId),
  [C0, C1, C2, C3, C4, C5, C6, C7].
```

B.3.7 rA Register

```
%%%-------------------------------------------------------------------
%%% @author steve edmans
%%% @copyright (C) 2015, <COMPANY>
%%% @doc
%%% The module creates an actor that we use to handle the Arithmetic Status Register (rA).
%%% @end
%%% Created : 13. Sep 2015 11:33
%%%-------------------------------------------------------------------
-module(register_ra).
-author("steveedmans").

-define(ROUND_TO_NEAREST, 0).
-define(ROUND_TO_OFF,     1).
-define(ROUND_TO_UP,      2).
-define(ROUND_TO_DOWN,    3).

%% API
-export([start/0, loop/3, calculate_byte/2]).

start() ->
  register(register_ra, spawn(register_ra, loop, [?ROUND_TO_NEAREST, sets:new(), sets:new()]))
      .

loop(RoundingMode, EnableBits, EventBits) ->
  receive
    stop ->
      true;
    {From, value} ->
      return_state(From, RoundingMode, EnableBits, EventBits),
      loop(RoundingMode, EnableBits, sets:new());
    {From, rounding_mode} ->
      From ! {self(), RoundingMode},
      loop(RoundingMode, EnableBits, EventBits);
    {event, Flag} ->
      loop(RoundingMode, EnableBits, set_flag(EnableBits, EventBits, Flag));
    {remove, Flag} ->
      loop(RoundingMode, EnableBits, remove_flag(EventBits, Flag));
    Msg ->
      io:format("We received this message ~w~n", [Msg]),
      loop(RoundingMode, EnableBits, EventBits)
  end.

return_state(From, RoundingMode, EnableBits, EventBits) ->
  EventValue = calculate_byte(sets:to_list(EventBits), 0),
  EnableValue = calculate_byte(sets:to_list(EnableBits), 0),
  Value = ((RoundingMode * 256) + EnableValue) * 256 + EventValue,
  From ! Value.

calculate_byte([floating_inexact|Rest], Total) -> calculate_byte(Rest, Total + 1);
calculate_byte([division_by_zero|Rest], Total) -> calculate_byte(Rest, Total + 2);
calculate_byte([floating_underflow|Rest], Total) -> calculate_byte(Rest, Total + 4);
calculate_byte([floating_overflow|Rest], Total) -> calculate_byte(Rest, Total + 8);
calculate_byte([invalid_operation|Rest], Total) -> calculate_byte(Rest, Total + 16);
calculate_byte([float_to_fix|Rest], Total) -> calculate_byte(Rest, Total + 32);
calculate_byte([overflow|Rest], Total) -> calculate_byte(Rest, Total + 64);
calculate_byte([divide_check|Rest], Total) -> calculate_byte(Rest, Total + 128);
calculate_byte(_, Total) -> Total.

remove_flag(EventBits, Flag) ->
  case sets:is_element(Flag, EventBits) of
    true ->
      sets:del_element(Flag, EventBits);
    false ->
      EventBits
```

```
  end.

set_flag(EnableBits, EventBits, divide_check) ->
  case sets:is_element(divide_check, EnableBits) of
    true ->
      EventBits;
    false ->
      sets:add_element(divide_check, EventBits)
  end;
set_flag(EnableBits, EventBits, overflow) ->
  case sets:is_element(overflow, EnableBits) of
    true ->
      EventBits;
    false ->
      sets:add_element(overflow, EventBits)
  end;
set_flag(EnableBits, EventBits, float_to_fix) ->
  case sets:is_element(float_to_fix, EnableBits) of
    true ->
      EventBits;
    false ->
      sets:add_element(float_to_fix, EventBits)
  end;
set_flag(EnableBits, EventBits, invalid_operation) ->
  case sets:is_element(invalid_operation, EnableBits) of
    true ->
      EventBits;
    false ->
      sets:add_element(invalid_operation, EventBits)
  end;
set_flag(EnableBits, EventBits, floating_overflow) ->
  case sets:is_element(floating_overflow, EnableBits) of
    true ->
      EventBits;
    false ->
      sets:add_element(floating_overflow, EventBits)
  end;
set_flag(EnableBits, EventBits, floating_underflow) ->
  case sets:is_element(floating_underflow, EnableBits) of
    true ->
      EventBits;
    false ->
      sets:add_element(floating_underflow, EventBits)
  end;
set_flag(EnableBits, EventBits, division_by_zero) ->
  case sets:is_element(division_by_zero, EnableBits) of
    true ->
      EventBits;
    false ->
      sets:add_element(division_by_zero, EventBits)
  end;
set_flag(EnableBits, EventBits, floating_inexact) ->
  case sets:is_element(floating_inexact, EnableBits) of
    true ->
      EventBits;
    false ->
      sets:add_element(floating_inexact, EventBits)
  end;
set_flag(_EnableBits, EventBits, _Flag) ->
  EventBits.
```

B.3.8 Registers

```
%%%------------------------------------------------------------------- -
%%% @author Steve Edmans
%%% @copyright (C) 2014, Steve Edmans
%%% @doc
%%% This module contains the functionality for setting and querying
%%% the registers.
%%% @end
%%% Created : 17. Mar 2014 13:09
%%%-------------------------------------------------------------------
-module(registers).
-author("Steve Edmans").

%% API
-export([init/0, contents/0, set_register/2, query_register/1, query_adjusted_register/1, stop
     /0, set_register_lowwyde/2]).

%% Create a new ETS table and populate it with all of the available
%% registers.
init() ->
```

```
  Registers_Table = create_table(),
  create_user_defined_registers(Registers_Table),
  create_mmix_specific_registers(Registers_Table).

stop() ->
  ets:delete(registers).

contents() ->
  FL = ets:tab2list(registers),
  FL.

set_register_lowwyde(RX, RVal) ->
  CVal = query_register(RX),
  CQuot = CVal div 16#10000,
  NVal = CQuot * 16#10000 + RVal,
  {RX, NVal}.

set_register(Register, Value) ->
  ets:update_element(registers, Register, {2, Value}).

query_register(Register) ->
  [{Register, Value}] = ets:lookup(registers,Register),
  Value.

query_adjusted_register(Register) ->
  Unadjusted_Value = query_register(Register),
  utilities:signed_integer16(Unadjusted_Value).

create_table() ->
  ets:new(registers, [set, named_table]).

create_user_defined_registers(Registers) ->
  lists:map(fun(X) -> {ets:insert(Registers,{X, 0})} end, lists:seq(0, 255)).

create_mmix_specific_registers(Registers) ->
  Mmix_registers = [pc,rA, rB, rC, rD, rE, rF, rG, rH, rI, rJ, rK, rL, rM, rN, rO, rP, rQ, rR,
        rS, rT, rU, rV, rW, rX, rY, rZ, rBB, rTT, rWW, rXX, rYY, rZZ],
  lists:map(fun(X) -> {ets:insert(Registers,{X, 0})} end, Mmix_registers).
```

B.3.9 Trap

```
%%%-------------------------------------------------------------------
%%% @author steveedmans
%%% @copyright (C) 2015, <COMPANY>
%%% @doc
%%%
%%% @end
%%% Created : 20. Aug 2015 11:12
%%%-------------------------------------------------------------------
-module(trap).
-author("steveedmans").

-define(FPUTS, 7).
-define(STDOUT, 1).
-define(HALT, 0).

%% API
-export([process_trap/3]).

process_trap(0, ?FPUTS, ?STDOUT) ->
  R = registers:query_register(255),
  Txt = memory:get_nstring(R),
  [{display, Txt}];
process_trap(0, ?HALT, 0) ->
  [halt];
process_trap(RX, RY, RZ) ->
  io:format("Process an unknown trap ~w ~w ~w~n", [RX, RY, RZ]),
  [unknown].
```

B.3.10 Utilities

```
%%%-------------------------------------------------------------------
%%% @author steveedmans
%%% @copyright (C) 2015, <COMPANY>
%%% @doc
%%%
%%% @end
%%% Created : 08. Sep 2015 20:04
%%%-------------------------------------------------------------------
```

```
-module(utilities).
-author("Steve Edmans").

%% API
-export([signed_integer16/1, unsigned_integer16/1, hex2int/1, hex2uint/1, get_8_bytes/1]).
-export([get_0_wyde/1,get_1_wyde/1,get_2_wyde/1,get_3_wyde/1]).
-export([get_0_byte/1,get_1_byte/1,get_2_byte/1,get_3_byte/1,get_4_byte/1,get_5_byte/1,
     get_6_byte/1,get_7_byte/1]).
-export([adjust_location/2, twos_complement/1, minus_one/0, max_value/0, min_value/0]).

hex2uint(L) ->
  << I:64/unsigned-integer >> = hex_to_bin(L),
  I.

hex2int(L) ->
  << I:64/signed-integer >> = hex_to_bin(L),
  I.

hex_to_bin(L) -> << <<(h2i(X)):4>> || X<-L >>.

h2i(X) ->
  case X band 64 of
    64 -> X band 7 + 9;
    _  -> X band 15
  end.

signed_integer16(V) ->
  FV = unsigned_integer16(V),
  case FV >= min_value() of
    false -> FV;
    true  -> -1 * ((minus_one() - FV) + 1)
  end.

unsigned_integer16(V) ->
  twos_complement(twos_complement(V)).

get_8_bytes(V) ->
  A = integer_to_list(V, 16),
  B = lists:reverse(A),
  C = split_bytes(B, []),
  pad_with_8_zero(C).

split_bytes([], Acc) -> Acc;
split_bytes([B|[]], Acc) -> [{lists:reverse([B | "0"])} | Acc];
split_bytes(X, Acc) ->
  {B, R} = lists:split(2, X),
  NewAcc = [{lists:reverse(B)} | Acc],
  split_bytes(R, NewAcc).

pad_with_8_zero([_,_,_,_,_,_,_,_|[]] = L) -> L;
pad_with_8_zero([_,_,_,_,_,_,_,_|_] = L) ->
  {_Lose, RL} = lists:split(1, L),
  pad_with_8_zero(RL);
pad_with_8_zero(L) ->
  NL = lists:append([{"00"}], L),
  pad_with_8_zero(NL).

get_0_byte(V) ->
  VB = get_8_bytes(V),
  {_, [{B}|_]} = lists:split(7, VB),
  byte_to_int(B).

get_1_byte(V) ->
  VB = get_8_bytes(V),
  {_, [{B}|_]} = lists:split(6, VB),
  byte_to_int(B).

get_2_byte(V) ->
  VB = get_8_bytes(V),
  {_, [{B}|_]} = lists:split(5, VB),
  byte_to_int(B).

get_3_byte(V) ->
  VB = get_8_bytes(V),
  {_, [{B}|_]} = lists:split(4, VB),
  byte_to_int(B).

get_4_byte(V) ->
  VB = get_8_bytes(V),
  {_, [{B}|_]} = lists:split(3, VB),
  byte_to_int(B).

get_5_byte(V) ->
  VB = get_8_bytes(V),
```

```
  {_, [{B}|_]} = lists:split(2, VB),
  byte_to_int(B).

get_6_byte(V) ->
  VB = get_8_bytes(V),
  {_, [{B}|_]} = lists:split(1, VB),
  byte_to_int(B).

get_7_byte(V) ->
  VB = get_8_bytes(V),
  [{B}|_] = VB,
  byte_to_int(B).

get_0_wyde(V) ->
  VB = get_8_bytes(V),
  {_, B} = lists:split(6, VB),
  wyde_to_int(B).

get_1_wyde(V) ->
  VB = get_8_bytes(V),
  {_, B} = lists:split(4, VB),
  {C, _} = lists:split(2, B),
  wyde_to_int(C).

get_2_wyde(V) ->
  VB = get_8_bytes(V),
  {B, _} = lists:split(4, VB),
  {_, C} = lists:split(2, B),
  wyde_to_int(C).

get_3_wyde(V) ->
  VB = get_8_bytes(V),
  {B, _} = lists:split(4, VB),
  {C, _} = lists:split(2, B),
  wyde_to_int(C).

byte_to_int(B) ->
  FullByte = lists:append("00000000000000", B),
  hex2int(FullByte).

wyde_to_int(B) ->
  [{M0},{M1}|_] = B,
  Wyde = lists:append(M0,M1),
  FullWyde = lists:append("000000000000", Wyde),
  hex2int(FullWyde).

adjust_location(Location, Scale) ->
  (Location div Scale) * Scale.

min_value() -> utilities:hex2uint("8000000000000000").
max_value() -> utilities:hex2uint("7FFFFFFFFFFFFFFF").
minus_one() -> utilities:hex2uint("FFFFFFFFFFFFFFFF").

twos_complement(Value) ->
  Step3 = case Value < 0 of
    false ->
      Step1 = minus_one(),
      Step2 = Value - 1,
      Step1 - Step2;
    true ->
      -1 * Value
  end,
  Step3.
```