YAMS - Yet Another MIPS Simulator
Final Report

Adam Brinley Codd Ben Cotter James Milns
Qian Qiao Sridhar Sarnobat Simon West

December 19, 2003

Contents

List of Figures

List of Tables

Chapter 1

Project Brief

1.1 What is MIPS?

MIPS is a processor with a RISC (Reduced Instruction Set Chip) architecture. In the second year of Computing and JMC degrees at Imperial there is a laboratory exercise, as a part of the Compilers course, which involves translating a subset of Modula 2, called DEC(M2), into MIPS assembler code.

1.2 Program Requirements

Imperial College currently uses a MIPS simulator called SPIM to check that the code generated by the compiler is correct. The aim of this project is to write a more user friendly simulator which is partly customized for the needs of the course. There should be a text-only console version and a graphical version.

The program must output helpful error messages if the MIPS code given to it is not syntactically correct, and it must either give output produced by a run of correct MIPS code or output useful statistics about the program.

The graphical version must contain a help facility giving the MIPS instruction set with clear explanations, and other useful debugging tools such as the ability to set breakpoints and step through the code one instruction at a time.

Chapter 2

Specifications and Interfaces

The application is written in Java. The main reason for this is its cross-platform compatibility. Students can run it on any department-supported Operating System: Windows, Linux or Mac OS. The application has four main components:

A *Parser* validates the input assembler file's syntax, informing the user of any errors. It builds an *Abstract Syntax Tree* (AST) of sorts, which is passed to the Assembler.

An *Assembler* translates the MIPS instructions in this AST to machine code; the machine code is stored in the *Memory Manager*.

Next, a virtual *Processor* is activated and interprets the machine code. It simulates the actions, recording input or displaying output where necessary.

The final component is the *User Interface*. There are two versions:

- a *text only* interface which accepts input from and prints output to a command line (be it the UNIX console, DOS, ...).
- a *Graphical User Interface* (GUI).

Both have a central *Controller* class which communicates with the three components above. In the case of the *text only* version, this controller class is the *YAMSConsole*. It handles command line input and deals with the overall flow of the program. The equivalent for the GUI will be discussed in Chapter 3.

2.1 Lexer and Parser

The Lexer and Parser (hereafter simply referred to as the Parser), has the task of reading in assembler source files (**.asm* files) and building the AST. This AST is, in fact, linear due to the nature of assembler programs and has thus been modelled as a `LineList`. This class is described in figure 3.12.

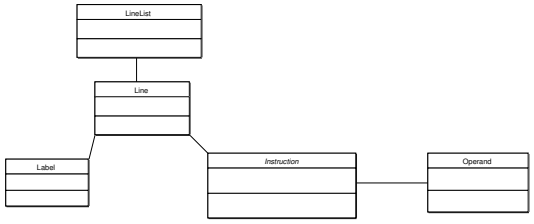

Figure 2.1: LineList

A `LineList` object contains a list of `Line` objects. Each `Line` object contains either a `Label`, an `Instruction`, or both.

Instructions are either R-type, I-type or J-type and, depending on their type, contain one, two or three operands. Operands can be one of six types, as illustrated in figure 2.2.

Alternatively, Instrucions can be of type `Directive` followed by a list of operands (possibly empty).

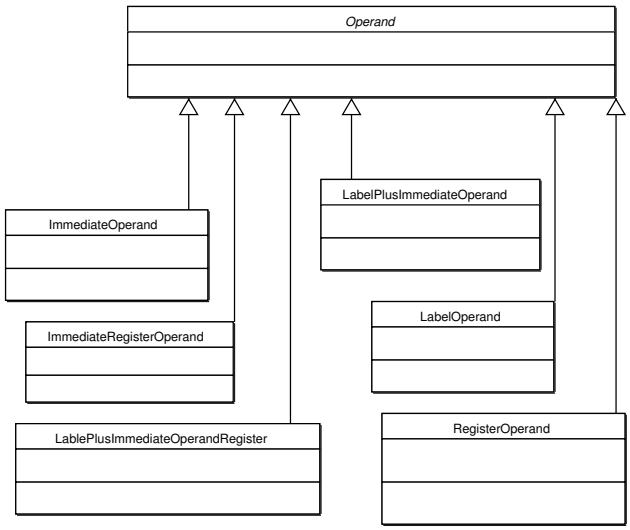

Figure 2.2: Operands

The `Lexer` tokenizes the input from an assembler file and control is passed to the `Parser`. The `Parser` attempts to translate a line of tokens into an object of type `Instruction`. Parse errors are immediately reported to the user and must be corrected before continuing to the next stage.

2.2 Assembler

When the Assembler object is created, the first thing is does is create an `AssemblerXMLHandler` which is used to access the XML file where all of the MIPS Instructions' behaviour is specified. The `Assembler` builds the `InstructionTable`, a lookup table for instructions and their machine code representation.

The `LineList` object returned by the Parser is passed to the `Assembler` which iterates through it line-by-line.

If the line contains a label, a reference to it is stored in the `SymbolTable`.

If the line contains an instruction, it is assembled to machine code using the template given by the `InstructionTable`. If an instruction contains a label, this is looked up in the `SymbolTable`. Sometimes, however, a label may be encountered before being defined in the assembler file. In such a case, the assembler adds a note to the `ToBeDoneList` and continues. After completing one pass of the list, all labels should have been defined. At this point the assembler verifies that all symbols in the `ToBeDoneList` do indeed refer to an entry in the `SymbolTable`.

2.3 Processor

The `Processor` is created in the controller (either `YAMSConsole` or `YAMSGui`. The classes' main job is to create and co-ordinate activity between the Processor's constituent components. These include:

- Cycle Manager
- Register Manager
- Memory Manager
- System Call Handler
- Instruction Handler
- Instruction Decoder

The Cycle Manager, whilst running, executes machine code instructions which have been stored in the Memory Manager. It looks up the current instruction using the Program Counter register, with the aid of the Register Manager, and tries to execute it using the Instruction Decoder.

The Instruction Decoder converts the instruction to a string of 32 bits in order to make it easier to manipulate. It then attempts to identify which instruction it is and calls the corresponding code from the Instruction Handler or, in the case of a system call, the System Call Handler.

The Instruction Handler and the System Call Handler were automatically generated form the XML file, so that the instruction set can be extended without changing any of the source code files. The XML file includes a small amount

of Java code for each instruction describing what it does and, of course, it has access to the Register Manager and Memory Manager.

The Register Manager and Memory Manager provide access to the Registers and Memory respectively. In order to make the simulation reflect reality as far as possible, the Memory Manager only has public methods, for reading from and writing to memory. Similarly, the Register Manager can only read from and write to registers. We could have included more complicated access methods but felt that this was giving too much responsibility to an otherwise simple part of the processor.

Chapter 3

Detailed Descriptions

3.1 Lexical Tokenizer and Parser

3.1.1 Overview

The parser package is quite self-explanatory. It parses the user's program input, generates a `LineList` (a AST[1] derivative) object which will be used by the assembler. Figure 3.1 is a UML class diagram of the major components of the parser package.

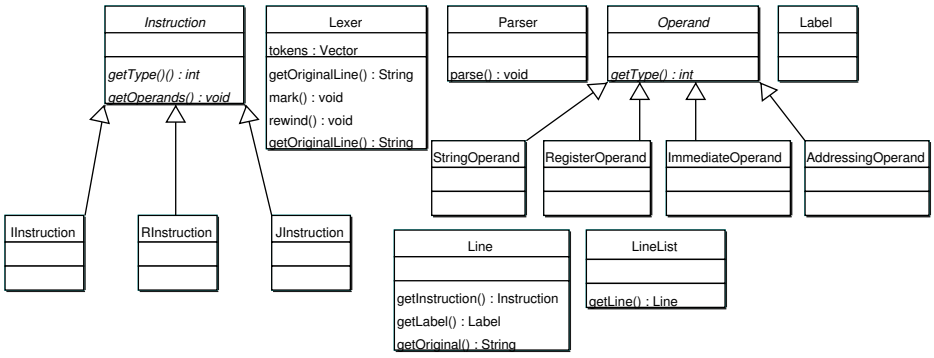

Figure 3.1: UML diagram for the parser package

3.1.2 Tokenizing

A Lexical tokenizer (referred to as 'lexer' in later part of this report) breaks up the original program code into a sequence of tokens, ignores all comments and whitespaces. Each token is associated with a type and a value.

[1] Abstract Syntax Tree

Example: `yams(5, "isGreat")` will be broken into:
`IDENTIFIER` token `"yams"`
`LEFT_PAR` token `"("`
`INT` token `"5"`
`COMMA` token `","`
`QUOTE` token `"\""`
`IDENTIFIER` token `"isGreat"`
`QUOTE` token `"\""`
`RIGHT_PAR` token `")"`
The lexer build for the YAMS project has the following additional features:

- We need to provide a tool for the user to debug their program, hence the lexer needs to be able to output user's program in it's original form as well as the tokenized form, so a `getOriginalLine()` method is implemented.

  An example:

  user's input: `add $a1, $a2, $a3`

  `nextToken()` returns: `"add"` token, then `"$"` token, followed by `"a1"` token, etc.

  `getOriginalLine()` returns: a `String` containing: "`add $a1, $a2, $a3`"

- Unlike normal lexers which construct only one token each time, when the `nextToken()` method is invoked it loses the previous one. The lexer for YAMS tokenizes the whole program at its initializing stage. The lexer then stores all the tokens in a `Vector`. This feature is required by the `mark()` and `rewind()` feature.

- There are two methods `mark()` and `rewind()` which enables the parser to look a arbitrary number of tokens ahead. Here is an example of the advantage of using `mark()` and `rewind()`:

  Suppose we have a language gramma that has 2 rules: A, B, C, D, E and A, B, C, D, F. Traditionally, when a parser sees an A, it'll have to store all A, B, C, D before it can make a deterministic choise of which rule to follow. Then it needs to retrieve A, B, C, D, and start the parsing. And also, a parser build to look 5 tokens ahead will normally have a storage mechanism which stores exactly 5 tokens.

  With the help of the `mark()` and `rewind()`, things can be done in a easier way. When the parser sees A, it set a mark at A by invoking the `mark` method, it then go on checking each tokens until it can make a deterministic choice, then it invokes the `rewind()` method, the token pointer will be set back to A, and the parser can start parsing.

  Suppose the language specification is then changed, the parser now needs to look 50 tokens ahead to make it choice. The parsers written in tridi-tional ways will have to be re-written, whereas the parser build usin the lexer's `mark()` and `rewind()` method won't have to change at all.

Hence this lexer is suitable for any parser with any number of look-ahead tokens.

Normally, for MIPS instructions, an LL(1) parser will be sufficient. However, the parser in the YAMS project works in a slightly different way to normal parsers, which is another reason the lexer is constructed to support `mark()` and `rewind()`. Detailed reason will be given in section 3.1.3.

3.1.3 Parsing

The parser analyzes the phrase structure of the program, and generate a AST representing this structure.

As the parser will be generated from the XML file. It would be better if we could separate the functionality of parsing each operand into the `Instruction` class, and hence reduce the complexity of the XSLT stylesheet[2].

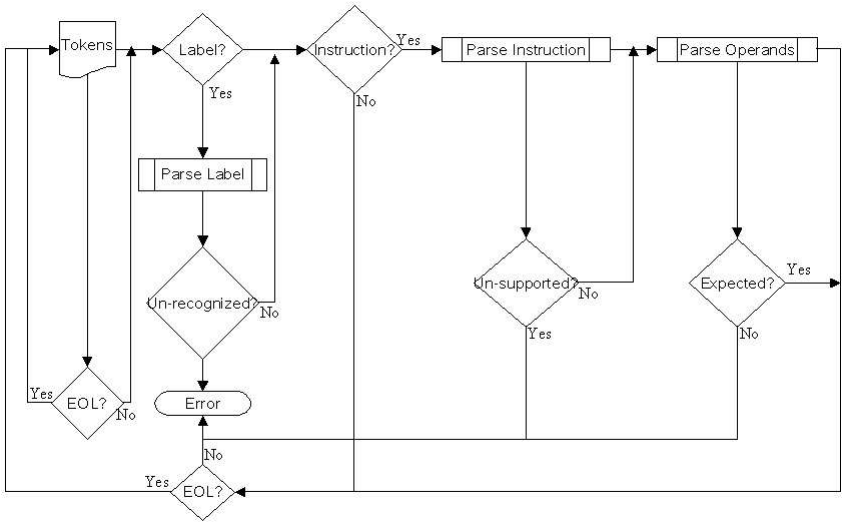

Figure 3.2: How the parser works

Figure 3.2 is a diagram illustrating how the parser parses a instruction. The actual parsing is slightly more complicated, but this is it's main idea:

1. If the current token is end of file (EOF), finish the parsing, otherwise go on.

2. If the current token is '\n' (NEWLINE), create an empty `Line` object, advance the token pointer, start over again.

[2]Details of how the auto-generation works will be explained in section 3.1.4

3. If the current token matches an IDENTIFIER, and has a COLON following it, create a new label, advance the token pointer, then go on, otherwise go straight to rule 5.

4. If the current token matches a '\n' (NEWLINE), create a new `Line` with only the label in it, advance the token pointer, start over again, otherwise go on.

5. If the current token(s) matches an instruction or directive, create the corresponding object, specify which operands the instruction/directive takes, wait for the object to parse it's operands. If parsed successfully, advance the token pointer, go on, otherwise raise an error.

6. Test the current token against '\n', if true, advance the token pointer, return new `Line` with what was parsed in this line so far, otherwise, raise error.

3.1.4 XML Auto-generation of the Parser

Why We Want Auto-generation?

Traditionally, all instructions and keywords support by a parser are hard-coded into the parser or as class in the parser package.

This could be cumbersome if a new instruction is needed to be supported by the parser, or the specification of a instruction has changed and it takes different operands.

Knowledge of the language the parser is written in will be required, time must be spent in order to understand how the parser works and to figure out where to edit (there could be several places!). Moreover, the new parser has to be tested again, and this whole process can be error prone.

One way of resolving this problem, is to store the details of the instructions in a file, and have some tool to generate the parser automatically, and this is how the parser in YAMS project is constructed.

Why Use XML?

There are several reasons:

1. XML is semi-structured. Unlike files written in plain text, XML *tags* data. Tags can also be nested, so that it reflects the hierarchy of the data.

   With the help of a clear documentation, and a well structured XML file. Editing the XML file can be straight forward, hence, significantly speed up the process of the re-configuration of the software.

   Figure 4.6 is an example the XML instruction file in the YAMS.

2. XML is easy to handle. Java provides powerful packages to read, parse and transform XML files. Less effort will be required to process the configuration file. Without XML, we need to write a parser to parse the configuration file. ergh...

An alternative to XML is to write a BNF grammar file, and have a parser generater (e.g., ANTLR[3]) to generate the parser.

The reason we didn't use ANTLR:

1. Constructing a BNF is time consuming, and for the end users to understand BNF, and to be able to modify it, a lot of knowledge is required.
2. The ANTLR package must be included in our software.
3. We had problems with ANTLR. Details will be discussed in section 3.1.5.

How Does Things Work?

Things will be straight forward if we illustrate the process of producing a parser from XML by a diagram.

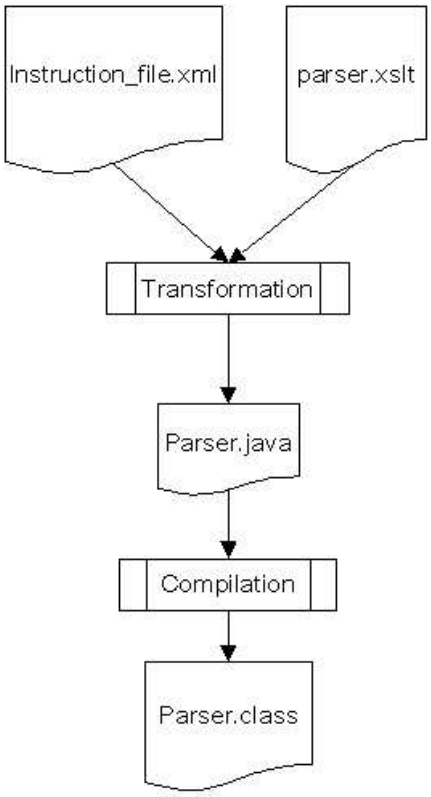

Figure 3.3: Process of producing a parser from a XML file

[3]A open source parser generater, can be found at http://www.antlr.org

The XML parser builds a internal model of the XML file and the XSLT file[4].

It then performs the transformation according to the rules specified by the XSLT, in our case, generate a plain text file containing Java code and ends with .java extension.

Then the java compiler compiles the java code, and a customized parser class is then created.

3.1.5 Problems

ANTLR and BNF Grammar File

Our original intention was to use ANTLR as our parser generator, so if we can find a BNF grammar of the full MIPS instruction set, we'll be able to generate a parser that supports the full MIPS instruction set.

It'll take less time and less work will need to be done, however, we wasn't able to find a BNF with deterministic rules, i.e., a correct BNF, off the internet.

[4]The XSLT file is just another XML file specifying how to transform the XML file from one form to another

3.2 Assembler

The following section of the report will describe the purpose of the Assembler Software component within YAMS, and provide in-depth details regarding its implementation and how this relates to YAMS as a whole.

3.2.1 Introduction: Purpose of assembler

The Assembler is the intermediate stage between the Parser and the Processor. The parser will have already taken the MIPS code provided to YAMS as input, checked its syntactic validity and transformed it into an internal representation that is simple to retrieve and manipulate.

Within the YAMS implementation we have closely modelled the real structure of a MIPS R3000 processor, thus internally the processor will retrieve instructions from its Memory Text Segment and execute them as in the real case. Therefore, logically the next phase in the process is to turn the internal representation of the parsed MIPS program into a binary representation that can be written to memory for the processor to execute.

The assembler carries out this task of assembly, while checking the semantics of the program as a whole e.g. checking whether the user refers to labels that aren't contained within the program, or whether all immediate values provided in decimal can safely be handled by the processor (i.e. no arithmetic overflow problems) once translated into their correct binary representation.

3.2.2 XML Interaction

As has already been mentioned within this document, XML has been extensively used within the project and is in fact at the core of the flexibility YAMS provides. All of the instructions supported by YAMS are stored within "Instruction_file.xml," which is the Instruction Repository for the software. Intuitively, the Assembler needed to have a flexible design, handling instructions in a generic way such that it could cope with any new instructions that advanced users wished to implement.

Unlike both the Parser and the Processor, the Assembler would not have automatically generated handlers for each instruction it supports. Instead, it reads the required information from the XML at runtime and places this in tables which it can read and manipulate at will.

3.2.3 Overview

COMPONENT DESIGN

The following class diagram indicates the overall structural design of the Assembler, indicating which objects would maintain permanent links during YAMS runtime.

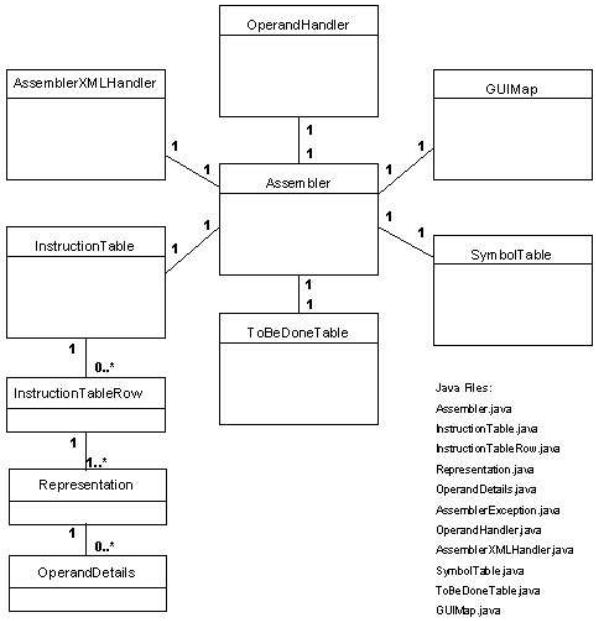

Figure 3.4: Class Diagram

BRIEF COMPONENT DESCRIPTIONS

As can be seen in the above diagram, there are a number of components within YAMS Assembler. They can be classified conceptually in two main areas. The assembler stores a lot of information (not just from the XML file, but also during assembly itself and for later stages within YAMS). Therefore there are isolated tables that simply store data and carry out operations limited to their data. The other components carry out the required tasks related to assembly utilising these tables accordingly. An overall summary of the components is outlined here, implementation details are provided later.

Tables used by the assembler:

InstructionTable The InstructionTable is designed to store the data read from the "Instruction_file.xml" by the AssemblerXMLHandler in a predefined object structure which can be easily read by the Assembler.

SymbolTable The SymbolTable is designed to store maps of symbols (e.g. a label within the Text segment OR a variable name in the Data segment) to their absolute addresses within memory.

ToBeDoneTable The assembler will be reading and assembling the provided user code "line by line," therefore there will be occasions where an instruction cannot be immediately translated into machine code. For example,

if we encounter a 'j _location' (JUMP to _location) MIPS instruction and _location does not appear till some later line in the program, the assembler will not be able to work out the offset to assign the corresponding target jump instruction in binary. Therefore the translation of this instruction must be left till a later time, and the ToBeDone table contains a list of these instructions that need implementing along with the addresses to which they need assembling in memory.

GUIMap The GUIMap component is a table whose existence is pertinent from the GUI point of view. It maps the line numbers of the original MIPS file to their assembled locations within memory, and can be used in the GUI to keep track of which line is being executed during a run of the program.

Task handlers used by the assembler:

Assembler This is the core component of the overall Assembler structure, and it coordinates the activities of the other components. The public methods it provides will allow one entire program to be assembled into machine code, using the other components which it creates and maintains.

OperandHandler This component is designed to take a provided set of operands for a given instruction, along with its core machine code representation (without any operand values filled in). It then performs the correct translation of the operands as required for the specific instruction and substitutes them into the machine code, leaving us with a machine word that can be written to memory ready for execution.

AssemblerXMLHandler This component is designed to read from the "Instruction_file.xml" instruction repository, by parsing through an internal representation of the file at runtime. It searches for expected tags and builds up a table of data on the instructions stored within the repository (table is InstructionTable described later).

3.2.4 Assembler

As can be seen in the class diagram already seen, the Assembler is the central object that coordinates the activities of the overall Assembly process. It therefore has control over the flow of all data, making critical decisions and calling other components in to carry out required tasks.

External Interactions

Within the Assembler framework, the Assembler class is the only class with any external links to other YAMS components.

CREATION An instance of the Assembler class is created by the YAMSConsole/YAMSGui objects (choice of console/Gui versions).

CALLED BY The YAMSConsole/YAMSGui calls the public methods within Assembler class to assembler one MIPS Program (already Parsed to an AST).

NOTE: The Assembler is designed to ONLY assemble ONE MIPS Program at a time, and has not been enabled with features that can cope with linking specific programs together. Therefore, the interface methods it provides allow only a single program to be submitted for assembly.

```
public void resetAssembler()
public void assembleCode(LineList MIPSProgram)
```

The resetAssembler() method resets the assembler ready to receive a new MIPS program, while the assembleCode method instructs the assembler to begin assembly on a new MIPS Program.

CALLS The Assembler class maintains a reference to the Memory Manager within the Processor package to be able to write machine code words to memory directly as they are assembled.

It also maintains a link to the StatisticsManager class (also within processor package) so that while it reads information from the XML file, it can build any tables that it needs to maintain statistics on individual instruction usage during program running etc.

Requirements

INPUT The assembler receives a LineList object (AST) from the Parser

NOTE: The LineList object is described within the Parser documentation, but essentially provides a mapping from line-numbers to their contents:

```
Label                   _location:
Instruction                         lw $a1,_x
Label and Instruction   _start:     add $a1, $a2, $a3
```

The contents of the Labels and the Instructions (including the values of the operands) are accessible through a variety of methods within this AST.

ASSEMBLY Work through the LineList object "line-by-line," translate to machine code, encoding each instruction and associated operands into their machine code representation(s).

OUTPUT 32 Bit Words written to memory (Memory Manager) correctly representing the MIPS source code in binary (both TEXT and DATA segments), ready for the processor to begin its operation.

Operation

There is one pre-operation phase that must be completed when YAMS is first created:

0. Assembler Instantiation

There are three main stages of data flow for the Assembler class:

1. Table Initialisation
2. 1st Pass Assembly
3. 2nd Pass Assembly

The implementation of these stages is outlined below:

0. ASSEMBLER INITIALISATION

   When an instance of YAMS is first created, the YAMSConsole or the YAMSGui classes (choice depending on required mode: console/gui) will create the core components of the software: Parser / Assembler / Processor. Therefore there are a number of stages the Assembler class must go through before any programs can be supplied for assembly.

   (a) Table Creation

   Must create tables it will need during assembly:

   - InstructionTable
   - SymbolTable
   - ToBeDoneTable
   - GUIMap

   (b) Operand Handler Creation

   (c) Set internal MemoryManager reference

   The Assembler creation call will also contain an external reference to an already existing MemoryManager. Since the writeToLocation() method contained within this class will be used frequently throughout various stages of the Assembler, it is important to set a global reference to this object.

   (d) XML File Reading

   It is necessary to fill the InstructionTable with information on each of the instructions within the `Instruction_file.xml` repository, since this will give us our supported instruction set. It would be possible to do this every time we reset the assembler, and capture any changes to the XML file (say if an instruction is added). However any changes to the instruction set require recompilation of YAMS to autogenerate the Parser and Processor handlers, so it is only necessary to read the XML once and copy the data to our tables.

This is achieved by utilising the LoadTableFromXML() method within the AssemblerXMLHandler, which when provided with the XML File Path and the empty instruction table (just created in phase I), will fill the table.

1. TABLE INITIALISATION

   The Assembler must initialise all of the tables that it will require during its later phases. References to these tables will already exist within the Assembler class (see the previous section), they must now be reset (to clear the assembler of any old information on other programs), and the assembler be made ready to receive a new file. The following tasks are required:

   (a) Reset Tables

   SymbolTable Reset any mappings from symbols to addresses. ToBeDoneTable Reset any previous instructions that needed to be dealt with in the second phase. GUIMap Reset any mappings from lines to addresses.

   (b) Reset TEXT Address value

   During assembly, the global variable `NEXT_TEXT_ADDRESS` will be used as the next free address in the TEXT segment where assembled instructions can be written to. Therefore, we must reset this when we are about to encounter a new program. Just like SPIM, and as set out in various definitions of the MIPS architecture, we define the TEXT segment to start at address 0x400000, and thus this is the required reset value.

   (c) Reset DATA Address value

   A similar point to above, we must reset the `NEXT_DATA_ADDRESS` to address 0x10000000, so that we effectively empty the DATA segment (from assembler's perspective).

   (d) Reset Fixed Block Address Offset

   YAMS can support fixed block addressing for certain instructions, for example lw. Please see the later OperandHandler section for further information on possible addressing modes. However, just in case this value has been changed at all during previous assemblies, we reset the value to 0x10008000.

2. FIRST PASS ASSEMBLY

   Purpose?

   The assembler must now iterate through the LineList (AST) object that has been passed from the Parser and attempt to assemble the program.

   The following data-flow diagram will explain this process in further detail:

   The following are considered global variables for this data flow:

   `NEXT_TEXT_ADDRESS`, `NEXT_DATA_ADDRESS`,

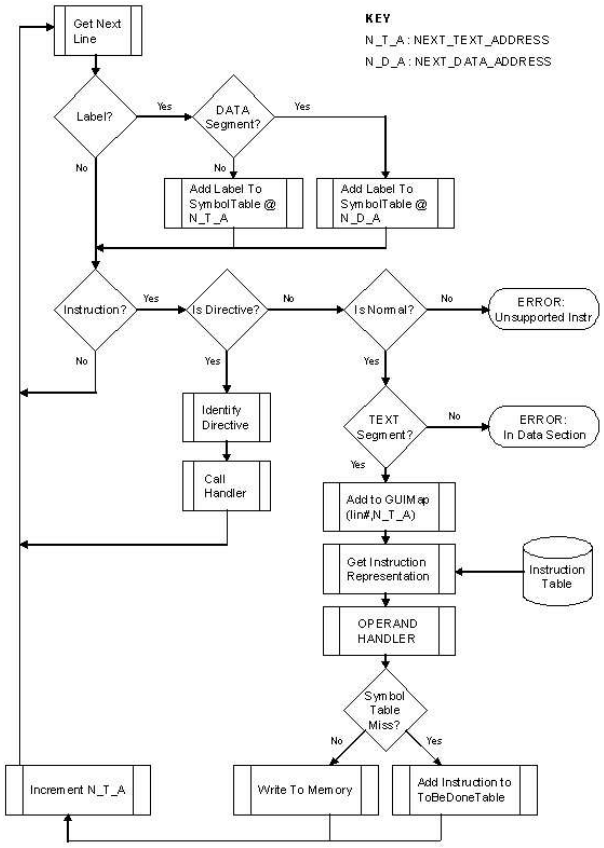

Figure 3.5: DFD For 1st Pass

Data/Text Segment (position of Assembler through file TEXT/DATA segment)

And for simplicity we are omitting graphically the GUIMap and Symbol Tables.

When the Assembler encounters a new line it must first check if it contains a label or not e.g. `_start:` If it does contain a label, then an entry needs to be added to the SymbolTable such that instructions can make references to the location and the assembler is able to resolve these addresses. If we are in the data segment then we add the label with the `NEXT_DATA_ADDRESS`, if the text segment then we add it with `NEXT_TEXT_ADDRESS`. Clearly, if we receive a label in the data segment then we MUST expect some variable definition as the next instruction, and this should have been already checked syntactically by the parser so no double-checks are performed here. Labels are allowed to be on their own lines within the TEXT segment, so

there are no conflict problems there.

The Assembler next checks whether the line contains an Instruction (which can be an Assembler Directive or a Normal Instruction). Logically, if we are dealing with an Instruction, the Directive/Normal check is carried out subsequently.

If we encounter a directive (identified as a normal instruction with a "." prefix, we identify the type of directive (hard-coded into specific handlers) and excecute the code. Supported directives are as follows:

.asciiz Writes a subsequently defined string to the DATA segment in memory using Ascii encoding.

.space Sets aside a specified number of bytes within the DATA segment, achieved by incrementing `NEXT_DATA_ADDRESS` accordingly.

.data Indicates the beginning of the data segment, changes global state variable in assembler (knows to expect variable definitions).

.text Indicates the beginning of the text segment, assembler will expect instructions).

.word Encodes the specified word into binary and places it in the DATA segment.

Other MIPS directives currently have their handlers constructed within the source code, but are NOT implemented. This can be left as an extension to the project, to completed at a later date.

NOTE: XML Problems: It was considered that the Directives be added to the XML file, instead of being hard-coded within the Assembler class. However, the sheer volume of code required for each handler implied that a fixed directive set should be kept. However, it would be perfectly possible to implement similar XSLT transformations to those used on the Parser / Processor to autogenerate these handlers if it were really necessary and this can be left as an Extension to YAMS.

If we encounter a normal instruction that is in the TEXT segment (instructions in DATA segment are NOT allowed), then we add a mapping of the current line to the `NEXT_TEXT_ADDRESS` to the GUIMap. Next we retrieve a Representation object for this current instruction name and operand types from the InstructionTable and pass these to the Operand Handler (described later) which attempts to correctly encode the operands and return the full machine code for the instruction. We now check if there has been an error in encoding operands in terms of a SymbolTable "miss."

These occur where we attempt to resolve an address of a symbol we have not yet encountered in the program (could occur later, for example a jump instruction to a later address). If we do have a "miss" then we must add the current instruction to the ToBeDoneTable (along with the start TEXT address where it should be written to) so we can attempt to encode it later,

once we have passed through all of the label references in the program. We DO NOT write ANY of the machine code for this instruction to memory. If we have no symbol table problems then we go ahead and write all of the code for this instruction to memory. In both cases however, we increment the `NEXT_TEXT_ADDRESS` variable so that space is left for when the second pass takes place.

The Assembler then moves back to the next line in the AST.

3. SECOND PASS ASSEMBLY

   Purpose? Why is a second pass required?

   As has already been detailed, a second assembly pass is required at this stage, as there will be some instructions which had unresolvable addresses at that particular point in assembly. The second pass is NOT through the whole program again, but rather only on those instructions that were added to the ToBeDoneTable.

   During this phase the Assembler iterates through the Instructions stored in the ToBeDoneTable. It then handles the instructions in a very similar way to that already described within the previous 1st pass section. It retrieves the Representation and invokes the OperandHandler. This time however, if we get a SymbolTable "miss" then we must throw an error. By this point, our symbol table should contain all of the label-¿address mappings for the entire MIPS program. Therefore if the label is not in the SymbolTable, then it cannot be in the program and we must indicate to the user that semantically speaking this program is erroneous. This semantic check is something that the parser cannot identify and is vital to the checking validity of a program.

3.2.5 OperandHandler

XML

One point that must be again highlighted regarding the structure and operation of the OperandHandler refers to the requirement for flexibility within YAMS. Since all the instructions are found in this repository and there is no hard-coding of instructions, then the OperandHanlder must be constructed to deal in the most generic way with the data it retrieves from the `Instruction_file.xml.`

Requirements

INPUT Receives a combination of operands:

`String machineWord` A single 32 bit bitstring, which is an "uncoded" machine code representation of some of the instruction which is required to be translated. `int address` An integer representing the current text address (`NEXT_TEXT_ADDRESS` from Assembler). `List opList` A list structure containing the Operand objects retrieved from the current Instruction

representation (passed from parser). `String opsCoding` A coded representation of the types of the operands in the list above. `int line` The current line number of the Instruction `Instruction current` The full instruction itself from the original LineList AST.

The final two operands imply redundant reference to certain objects, but note that they are used to highlight any errors encountered with the whole line.

NOTE: "uncoded" refers to machine code that hasn't had the correct values for its operands substituted yet e.g. 001000bbbbbaaaaaccccccccccccccc

NOTE: Some instructions within the supported set can be multiple words in length, and the Assembler essentially will pass the OperandHandler a SINGLE word from this representation, and the operands themselves ready to be substituted. The other variables are used to setup the handler to translate these operands correctly pre-substitution e.g. if one operand is a label, then the OperandHandler must establish whether the offset should be calculated from the current address or from the `FIXED_LINE_ADDRESS` (see addressing modes section later in this report).

HANDLING The handler must encode the provided operands, and substitute their translated values into the provided "uncoded" machine word.

OUTPUT Returns a machine word which is fully coded, with all operands successfully substituted into their correct positions, ready to be written to memory. Must also provide an indication of whether there has been some error in referencing the SymbolTable (referencing label not yet encountered in the MIPS program).

Operation

The OperandHandler has been created (by the Assembler class) with references to:

SymbolTable, ToBeDoneTable, InstructionTable

As explained in the Requirements section above, it must now attempt to encode a SINGLE machine word using the provided operands.

Diagrams

The following data flow diagram shows the main phases involved in encoding a single machine word using the OperandHandler when the encodeOperands() call is made from the Assembler.

Global variable used throughout the diagram:

List translatedList List of encoded operands, in machine code.

Firstly, the handler must clear the translatedList, to flush out any previously encoded operand values. It then recalculates the values of the operands in the operandList provided to the OperandHandler, adding them to the translatedList (process described subsequently). With these newly calculated operands,

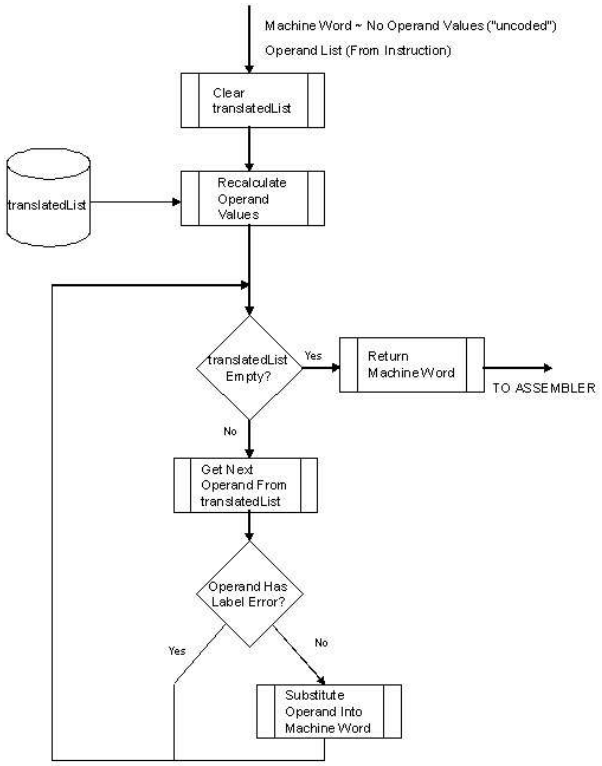

Figure 3.6: DFD For OperandHandler encodeOperands() call

we iterate through this list and replace their corresponding entries in the provided "uncoded" machine word in order. For example any substrings of "a" contained within the machine word are replaced by the FIRST entry in the translatedList. During this process if an operand has encountered a SymbolTable "miss" (referred to in Assembler-1st/2nd pass documentation) then its value is set to a recognised "ERROR" value. If this value is found, clearly this value isn't substituted into our output machine word. When the translatedList is exhausted, the substituted machine word is returned.

Such exhaustive methods for replacing the operands must be employed since the structure of any instruction is not known to the OperandHandler due to the XML requirements. Thus, it must attempt to anticipate any combination / location of operands within each machine word.

An example substitution:

```
addi rd, rs, imm 001000bbbbbaaaaaccccccccccccccc
                            rs    rd    immm

Operand Encoding
 $a1 = 00101
  20 = 0000000000010100

Substitution 00100000101001010000000000010100
```

The operands are recalculated for every word that is required to be encoded. Essentially, the list of provided operands is iterated through, and its type identified (e.g. Register, Immediate).

The typing system for operands within the OperandHandler has to be necessarily more complex than those used within the Parser, since for certain operands the assembler must know more detailed information. The types supported are as follows:

Therefore, the OperandHandler will identify a code for each of the Operands, and form an "OpCode" for this particular Operand Combination e.g. sub REGISTER REGISTER REGISTER has "OpCode" = 111. For each instruction stored in the XML there will be a unique machine code and associated operand information for each supported "OpCode." Therefore there will be different machine code representations for 111 than for 110 within the machine code, since it may be required that other instructions will be needed for the latter "OpCode" (say moving the immediate into a register before performaing the sub).

Further details on the operands are stored within the instruction repository, including the accuracy with which to encode in binary and which bits to include in your final substitution. These details are retrieved from the InstructionTable and set the handler to a specific mode defined by a variety of variables.

The operand value provided with the MIPS instruction is then evaluated within the bounds of this mode, and added to the translatedList as described before. This method of setting the handler to a specific mode and then starting evaluation is necessary to provide the flexibility of being able to create a variety of instructions.

Character Code	Meaning
0	<=16 Bit Twos Complement Immediate
a	32 Bit Twos Complement Immediate
1	Register
2	Label
3	Address
4	Add: Immediate
5	Add: Immediate(Register)
6	Add: Label
7	Add: Label_Plus_Immediate
8	Add: Label_Plus_Immediate_Register
c	Add: Label_Minus_Immediate_Register
9	Add: Register
b	no operands

Table 3.1: Operand Types

NOTE: For more information on the structure of the XML Representation of instructions please refer to the "Guide To Instruction Addition" within this documentation. This will also indicate the various addressing modes that are available to any instructions.

3.2.6 AssemblerXMLHandler

Requirements

INPUT Provided with a file path to the `Instruction_file` XML Instruction Repository and a reference to the empty InstructionTable.

HANDLES Must read the XML File, locate the `<Instruction>` tags containing data on each instruction, and add this data to the InstructionTable

OUTPUT Create an object based representation of the information stored in the file-based Instruction Repository.

Operation

The AssemblerXMLHandler achieves its objectives by using the already existing XML parsers included in the Java release. In this case, a DOM parser has been used.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
try {
  DocumentBuilder builder = factory.newDocumentBuilder();
  yams.util.FileReader fr = new yams.util.FileReader();
  Document document = builder.parse(fr.readFile(xmlFilePath));
```

```
  NodeList children = document.getChildNodes();
}
```

The DOM Parser automatically reads the XML file and generates an internal tree representation of the structured XML. This tree is then navigated through, and a series of hard-coded handlers attempt to match the values stored in these nodes to predefined tags, specified below.

```
<Instructions>

  <Instruction>
    <Name></Name>
    <OperandTypes></OperandTypes>
    <Javacode></Javacode>
    <Type fixedRt="true"></Type>
    <MachineCodeRepresentations>
      <Representation>
        <OperandsCoding></OperandsCoding>
        <MachineCode></MachineCode>
        <Operands>

        </Operands>
      </Representation>

    </MachineCodeRepresentations>
    <Help>
      <FullName></FullName>
      <Format></Format>
      <Description></Description>
    </Help>
  </Instruction>

<Instructions>
```

As soon as certain tags are encountered, the handler adds the required information to the InstructionTable. For example when the `INSTRUCTION_TAG` is encountered, then a new InstructionTableRow is created and added to the InstructionTable to contain all the information regarding the new instruction. As the navigation continues deeper into the tree beneath the `INSTRUCTION_TAG` further objects are created and then added appropriately to the correct entry in the InstructionTable.

The following indicates the events that occur at each tag encounter:

```
\\ NAVIGATE THROUGH TO LOWER LEVEL
private static String INSTRUCTIONS_TAG = "Instructions";

\\create new InstructionTableRow
```

```
private static String INSTRUCTION_TAG = "Instruction";

\\subsequent information appears in InstructionTableRow
private static String INSTRUCTION_TYPE_TAG = "Type";
private static String CORE_TAG = "CoreMachineCode";
private static String NAME_TAG = "Name";

\\create new Representation
private static String REPRESENTATION_TAG = "Representation";

\\subsequent information appears in Representation
private static String OPERANDS_CODING_TAG = "OperandsCoding";
private static String MACHINE_CODE_TAG = "MachineCode";

\\ NAVIGATE THROUGH TO LOWER LEVEL
private static String OPERANDS_TAG = "Operands";

\\create new OperandDetails
private static String OPERAND_TAG = "Op";

\\subsequent information appears in OperandDetails
private static String OPERANDS_TYPE_TAG = "Type";
private static String OPERANDS_NUMBER_TAG = "Number";
private static String OPERANDS_ENCODE_BITS_TAG = "EncodeBits";
private static String OPERANDS_OUTPUT_BITS_TAG = "OutputBits";
private static String OPERANDS_MASK_TAG = "Mask";
private static String OPERANDS_OFFSET_MODE_TAG = "OffSetMode";
private static String OPERANDS_TAG = "Operands";
```

NOTE: A further implementation point to note is that the handler has also been used to build a StatisticsManager table for keeping count of how many times a particular instruction has been used within any given execution. Therefore every time the `NAME_TAG` is encountered for an instruction, a method in the StatisticsManager is called to add this name to its table as a new entry.

3.2.7 Tables

The remaining components within the Assembler are simply table structures that are utilised by the Assembler, the OperandHandler and the AssemblerXMLHandler objects. Their definitions have already been discussed in the initial section of this chapter of the report, thus subsequent sections will only contain some specific implementation points:

Some general implementation points are common to all tables used within the assembler package:

Wherever possible, TreeMap structures have been used to store all mappings / required indexings. A TreeMap is a memory structure which stores

(Key,Value) pairs where both are Objects. It is memory efficient, and very effective in terms of both in storage and retrieval times. The other advantage of using this structure is that almost all of the table accesses used within the assembler are random index accesses and do not need iteration across the whole structure. For example, queries to the InstructionTable consist of asking for a specific entry where the name = "queryString."

Another common feature across all the tables is the use of multiple `get____()` and `set____()` methods. For structures that have nested TreeMaps, there are `add___()` methods to enable easy addition of objects to existing TreeMaps.

Finally, all the tables have reset features, allowing their contents to be wiped clean, while not deleting the object.

INSTRUCTIONTABLE

Structure

```
TreeMap instructionMap
  KEY = Instruction Name (String)
  VALUE = Instruction Info (InstructionTableRow)
```

Description

The InstructionTable stores data from the `Instruction_file` on each instruction to be supported by the assembler. In actuality, the InstructionTable uses a TreeMap to store objects of the type InstructionTableRow, which contains more information on the instruction and further links to other objects containing other details.

Hierarchically, when considering all levels of linking (please see Figure 1) the InstructionTable contains references to the following objects:

- InstructionTableRow
- Representation
- OperandDetails

The definitions of the structure of these objects is included within this section of the documentation.

NOTE: The implementation of InstructionTable also contains a mapping in the reverse direction from machine code -¿ instruction name. This has been included within the assembler to enhance debugging of pseudo-instructions. It enables the assembler to automatically locate the name of any given sub-instruction within a more complex pseudo-instruction, and print it for the user to identify what exactly is being identified. This feature is not necessary for the general implementation, but has been left within the project since it is possible extra instructions may be added to YAMS at a later date.

`INSTRUCTIONTABLEROW`

```
String intructionName
String instructionType
String coreMachineCode
TreeMap representations
  KEY = Operand Code (String)
  VALUE = Specific Representation (Representation)

REPRESENTATION

String operandsEncoding
String machineCode
TreeMap representations
  KEY = Operand Number (Integer)
  VALUE = Operand Details (OperandDetails)

OPERANDDETAILS

int operandNumber
String operandType
```

GUIMAP

Structure

```
TreeMap table
  KEY = Line Number (Integer)
  VALUE = Absolute Memory Address (Integer)
```

Description

The GUIMap maps between line numbers from the LineList AST provided initially by the parser and their absolute memory addresses.

SYMBOLTABLE

Structure

```
TreeMap table
  KEY = Symbol (String)
  VALUE = Absolute Memory Address (Integer)
```

Description

The SymbolTable maps between the symbol (e.g. `_location`) and its absolute memory address.

TOBEDONETABLE

Structure

```
TreeMap table
  KEY = Number (Integer)
  VALUE = Currently Note Assembled Instruction (Instruction)
```

Description

The ToBeDoneTable provides essentially a tree map of instructions that require re-assembling due to their containing a symbol which could not be found within the SymbolTable during the first assembly pass.

3.2.8 Summary

The structure of the assembler has been designed and implemented to cater for dealing with a dynamic instruction set. Unlike the Parser and the Processor, the Assembler doesn't have handlers that are rebuilt every time the instruction set changes. Instead, great flexibility has been provided in the OperandHandler and Assembler classes to allow all of the information regarding instructions to be read at run-time.

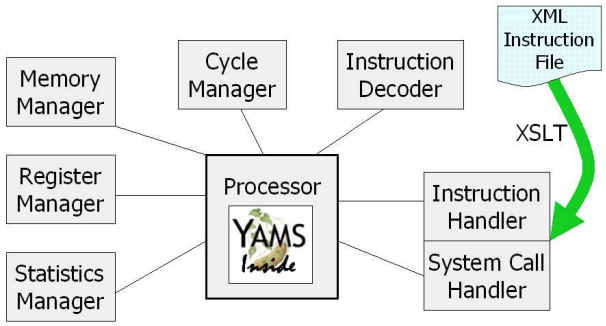

Figure 3.7: The Components of the YAMS Processor

3.3 Processor

3.3.1 Overview and UML Diagram

The processor is the simulation engine inside YAMS.

The processor reads machine code instructions from the memory, decodes and executes them. It models a MIPS R3000 CPU, which has a RISC architecture and all instructions are 32 bits in length. This length is called a MIPS machine word. The following diagram shows the software components that go to make up the processor.

This UML diagram shows how the components are implemented as interfaces and classes. The package that all processor classes may be found in is called 'yams.processor'.

The processor is capable of recognising and executing any instruction present in the XML repository. Example instructions are: mul (performs multiplication), add (performs addition), beq (performs branch if equal) and syscall (performs system call). After an instruction is executed, the state of the processor will be exactly as described by the instruction file. For example, after an 'add' instruction, the value in the destination register will be the result of the addition of the two other operands.

The processor exposes interfaces for each Manager component above, so it is simple to get information about any aspect of the state held by the processor. The processor also keeps many statistics on what has been executed.

3.3.2 Using the Processor

The entrypoint to the processor is the 'Processor' class. Create a new Processor object using the following constructor:

public Processor(YAMSController yamscontroller, InputStream in, PrintStream out, PrintStream verbose)

This will instantiate and initialise each class that is a processor component in turn. Each class with a responsibility to print or receive input has the ap-

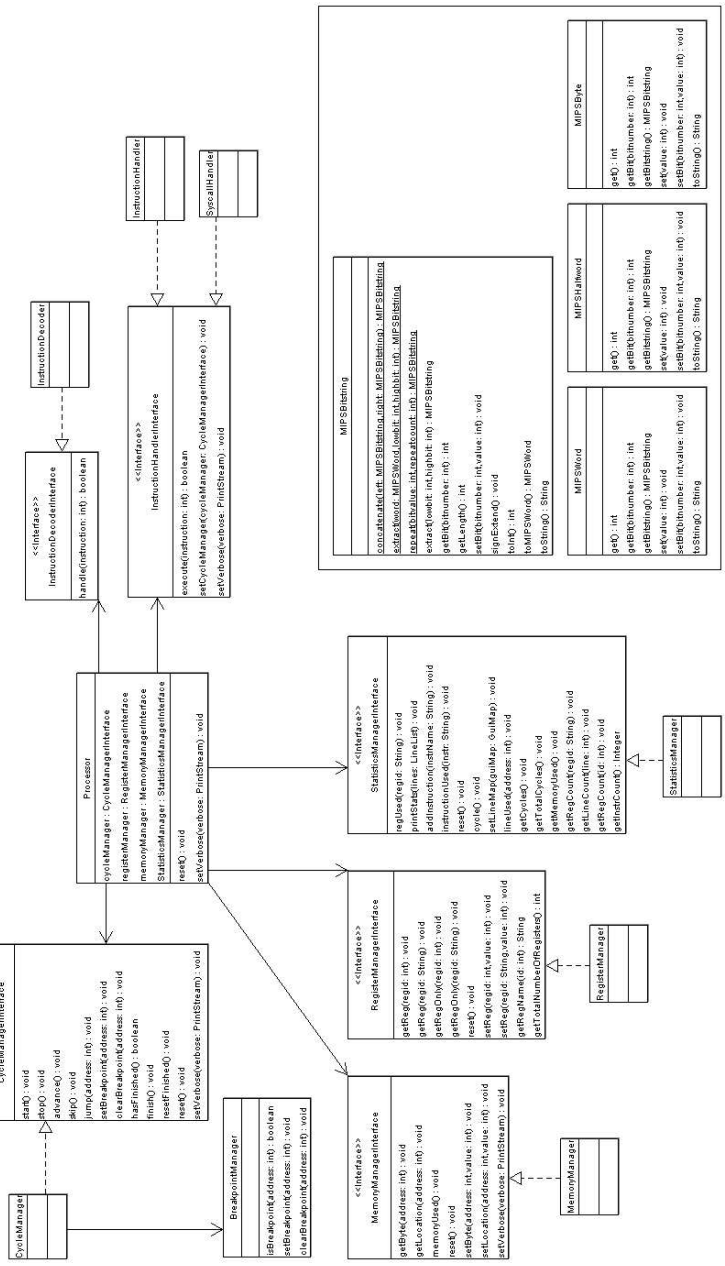

Figure 3.8: The UML Diagram of the Procesor Classes and Interfaces

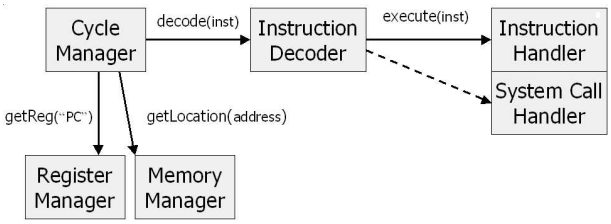

Figure 3.9: Execution of a Single Instruction

propriate Input or PrintStream(s) passed to it. The passed InputStream 'in' will be used for any textual input required from the user, the PrintStream 'out' is used for program output and the PrintStream verbose is used for extended execution information, such as the names of instructions.

The processor exposes four managers as public attributes: cycleManager, registerManager, memoryManager, and statisticsManager. To begin execution, the processor object's cycle manager should be told to jump(addr) to the start of code. Then the cycle manager's advance() method should be called to execute individual instructions, or start() to execute until completion. Completion occurs when an exit 'syscall' instruction is executed. The YAMS console and GUI perform this transparently to the user.

3.3.3 The Manager Classes

Each Manager class (with the exception of the BreakpointManager) implements a similarly-named interface.

CycleManager

The cycle manager sequences the operation of each instruction cycle, and the following diagram shows most of one such cycle.

First, the cycle manager gets the current value of the program counter. The memory manager is then asked for the machine word at the memory location corresponding to that value. The word, an instruction, is passed to the instruction decoder which determines which handler should execute it. The instruction is executed by the appropriate handler and the state of the MIPS registers and memory is affected as a result. Finally, and not shown, the program counter is incremented to point to the next instruction instruction in memory, and the statistics manager is updated.

The methods to control execution are: jump(int address) - sets the program counter to the given address start() - runs until a 'syscall' exit is encountered stop() - stops the processor after the current cycle finishes execution advance() - when stopped, execute the next instruction skip() - when stopped, skip over the next instruction

The cycle manager also supports breakpoints. When a breakpoint is encountered, the cycle manager stops execution before the instruction at the address of the breakpoint is executed. The BreakpointManager class maintains the list of breakpoints for the cycle manager. The breakpoints are held in a HashSet.

The cycle manager maintains a series of flags, to tell the controller (console or GUI) whether it is currently running (executing instructions) and if an exit 'syscall' has been executed. The reset() method can be invoked to reset these flags.

RegisterManager

The register manager simulates the registers that are available to the MIPS processor. There are 32 generic registers available to use, and 3 special registers `$PC`, `$LO` and `$HI`.

The register manager uses treemap data structures as they have fast access times. When it is created before the run of the program, the registers are loaded into the treemap with a 0 count.

There are then access procedures to get and set values for the registers. The value currently in a register is returned if a get request is made, or set with a specified value if a set request is made. Each time a register is accessed, a method in the statistics manager is called to update the usage count for that register. There are also access procedures that do not call the statistics manager, this is for when the content of a register needs to be read by the simulator, for example while displaying values in the GUI.

There is also a reset procedure to reset all of the values to 0 making the register manager ready for the next run program.

MemoryManager

The Memory Manager simulates the memory available to the MIPS processor. It is assumed that initially all the memory locations are set to 0. The memory is split up into 3 segments - reserved, text and data.

When the memory manager is created, the divisions of these sectors can be specified in the constructor, if these are invalid, the default segmentation is used.

The memory manager uses tree map data structures, one for each memory segment, as these have fast access times. The objects held in the treemaps are of type MIPSWord which can store 32bits and have various manipulation procedures making it easy to change the values and bits held in each. There are access procedures to get and set memory words, and also the same procedures to get and set individual bytes in memory.

When an incoming request is made to get or set a location, the address given is used to resolve the treemap that it corresponds to. The location value is then returned if it is a get request, or set with the supplied value if it is a set request.

There is also a reset procedure which clears the treemaps ready for the next program to be run. Since it is assumed that all value in memory are initially

zero, it is not necessary to store all locations in the treemap - any request that is made for a location not stored in the map will just return 0.

StatisticsManager

The statistics manager keeps a record of various statistics that are updated as the program is executed. These are:

Line execution count Showing how many times each line of the assembly file has been executed.

Instruction count Giving how many times each instruction has been used in the program.

Register usage count Showing how many times each register has been used in the program

CPU Cycles required to run the program

Total memory in words taken up by the program

The class uses tree maps to store the counts as these are ordered and fast to access. The CPU Cycles and Total Memory are just stored in variables.

Line Execution Count For the line execution count, the statistics manager needs a reference to the GUIMap class in the assembler which maps memory addresses to line numbers. Every time the Cycle Manager cycles, it passes the statistics manager the value of the program counter. The GUIMap is then used to resolve this address to a line number, so the line execution count can then be incremented.

Instruction Count When the instructions are initially loaded by the assembler from the XML instruction file, the names are also loaded into the statistics manager so the counts can be tracked for each program run.

When the instruction handler executes an instruction, a method is called in the statistics manager with the name of the instruction. This then updates the usage count for that run of the program.

Register Usage Count The register usage count is incremented by an access procedure called by the register manager each time a register is accessed.

CPU Cycle Count The CPU cycle count is incremented by an access procedure called by the cycle manager on every cycle.

Total Memory The total memory in words is given by calling an access procedure in the memory manager that returns the total size of all 3 memory maps held.

R type	The 'op' field (bits 26..31) are 000000
	The 'func' field (bits 0..5) uniquely identifies the instruction
I types	The 'op' field (bits 26..31) uniquely identifies the instruction
J types	As for I types, the 'op' field (bits 26..31) uniquely identifies the instruction

Table 3.2: Instruction Characteristics

Reset There is also a reset procedure to set all the statistics to 0 ready for the next program to run. For performance, the instructions names are not deleted from the instruction count treemap so they do not have to be loaded from the XML each time a new program is run. Instead the counts are just set to 0.

3.3.4 The Instruction Execution Classes

InstructionDecoder

The instruction decoder in the processor determines which handler should execute a given instruction. 'Syscall' instructions are executed in the syscall handler, and all other types execute in the instruction handler.

Both handlers implement the InstructionHandlerInterface, which contains the execute(int instruction) method that the instruction decoder invokes.

3.3.5 InstructionHandler

This class is one of the most important in all of YAMS, since it forms the point of execution of 99% of instructions.

In order to understand how instructions can be decoded and executed, the format of MIPS instructions must be understood. MIPS instructions can be broadly classed into 3 types:

R types which only operate on registers e.g. add rd, rs, rt

I types which have an immediate operand e.g. lui rd, imm

J types which are jump instructions with an address e.g. j label1

The characteristics of the instruction types, so that they can be detected and decoded, are shown in table 3.2

When the `execute()` call is invoked, an instruction to be executed is passed to the instruction handler. The handler tests it to see whether it has 000000 for its 'op' field, and thus whether it is an R type instruction. Otherwise it is an I or J type instruction. Once the broad type of the instruction is determine, the operands are decoded from the instruction. Not all operands are used in every instruction.

MIPS instruction operands are shown in table 3.3.

R types	rd (5 bit register identifier)
	rs (5 bit register identifier)
	rt (5 bit register identifier)
	shamt (5 bit shift amount)
	func (instruction identifier)
I types	rs (5 bit register identifier)
	rt (5 bit register identifier)
	imm (16 bit immediate)
J types	addr (26 bit register identifier)

Table 3.3: MIPS Instruction Operands

Then for an R type, the value of the 'func' field is compared against all the R type instructions in the instruction handler, and the correct MIPS instruction is executed using the operands as extracted above. For an I or J type, the 'op' field is compared against the all the I & J type instructions in the instruction handler, and the correct MIPS instruction is executed using the operands as extracted above.

If the instruction to be executed cannot be handled by the instruction handler, then an YAMSUnsupportedInstructionException is thrown. This would be the case, for example, if a floating point instruction was executed by an instruction handler with no floating point instruction support.

instructionHandler.xslt

Not a single instruction is hard-coded in YAMS. It would be undesirable and inflexible to do so. Instead, the XML repository's instruction file is used to produce an instruction handler at compile time through an XSLT transformation. When YAMS is built, `yams.auto.xslt.instructionHandler.xlst` is applied to the XML repository, `Instruction_file.xml`, and the transformation produces as output a Java source file: `yams.processor.InstructionHandler.java` This output Java file is compiled in with all other YAMS sources during the build process.

The instruction handler's XSLT transformation contains the template for an instruction handler class, but without any instructions present. Java code is produced for each separate instruction as if-statements, which are then added to the template instruction handler to form the generated instruction handler.

The transformation determines for each instruction detailed in the XML file, whether it is an R or I/J type. If it is an R type, it extracts the 'func' bitfield from the instruction's machine code representation and creates an if-statement to detect an R type instruction with that 'func' value. If it is an I/J type it extracts the 'op' bitfield instead and creates an if-statement to detect an I/J type instruction with that 'op' value.

Inside the if-statement, a statement is added to print the instruction name to the verbose PrintStream, and another to inform the statistics manager that

an instruction of that name is being executed.

The Javacode associated with the instruction is placed inside the if-statement, so that the semantics of the instruction are exactly executed. Finally, a 'return true' is added and the if-statement is closed.

An example follows.

SOURCE: Pertinent tags for 'add' instruction in XML file, `Instruction_file.xml`

```
<Instruction>
  <Name>add</Name>
  ...
  <Javacode>
  regs.setReg(rd, regs.getReg(rs) + regs.getReg(rt));
  </Javacode>
  <Type>Extended</Type>
  <CoreMachineCode>000000bbbbbccccc aaaaa00000100000</CoreMachineCode>
  ...
</Instruction>
```

OUTPUT: If-statement for 'add' instruction in generated `InstructionHandler.java`

```
if(func == toDecimal("100000")) {
  verbose.println("add");
  stats.instructionUsed("add");

  regs.setReg(rd, regs.getReg(rs) + regs.getReg(rt));

  return true;
}
```

SyscallHandler

The Syscall Handler only handles the 'syscall' instruction. This instruction is special because it can communicate with the operating system, that is, the 'outside world' to the simulator. Examples of uses are to print a string to the console, or read a number from the user.

The value in register $v0 is the system call code, the 'syscall' instruction performs a different function depending on this code.

SYSCALLHANDLER.XSLT

Like for the instruction handler, the XML repository's instruction file is used to produce a syscall handler at compile time through an XSLT transformation. When YAMS is built, `yams.auto.xslt.syscallHandler.xlst` is applied to the XML repository, `Instruction_file.xml`, and the transformation produces as output a Java source file: `yams.processor.SyscallHandler.java` This output Java file is compiled in with all other YAMS sources during the build process.

Service	Code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		

Table 3.4: System Calls

The syscall handler's XSLT transformation contains the template for an syscall handler class, but without any instructions present. Java code is produced for the syscall instruction as an if-statement, which is added to the template syscall handler to form the generated syscall handler.

The syscall handler's transformation only produces code for the 'syscall' instruction, all other instructions are ignored, since they are catered for by the instruction handler.

It may be argued that the separation of the instruction and syscall handlers is only a logical separation, but the YAMS developers believed it was a good idea to have the potentially exotic syscalls to be in a source file of their own.

3.3.6 The Bitstring Classes

MIPSBitstring

The processor helper classes were developed to help with bitstring and binary operations. The MIPSBitstring class represents bitstrings, that is, sequences of binary digits. It supports concatenation and splitting of bitstrings, as well as giving decimal integer representations. This class is used extensively to extract and test bitfields inside MIPS instructions. It can also provide sign-extended decimal integer representations of bitstrings, which is very useful for converting a 16-bit two's complement immediate into a 32-bit signed Java int.

MIPSByte, MIPSHalfword, MIPSWord

The Java int datatype is used throughout YAMS as a 32-bit integer, that can represent a MIPS machine word. The MIPSWord class (and similarly for its smaller-width sister classes) is a wrapper for the MIPSBitstring so that the machine word has simple get() and set() methods, as well as bit manipulation and bitstring extraction methods.

3.4 Graphical User Interface

The YAMS class creates a new YAMSGui object and invokes its `start` method.

The `start` method begins by calling `setUpGui()` which creates the `mainFrame` and other window elements. The main window is divided into six panels as follows:

- `FileSelectorPanel`
- `ProgramCodePanel`
- `DataPanel`
- `RegistersPanel`
- `StatisticsPanel`
- `DialogPanel`

Each of these will be described in more detail later in this section. They are all created with a reference to the `YAMSGui` class and so any communication between panels is done via `YAMSGui`.

Also created is a `JMenuBar` with three menus:

- A `File` menu for simple file operations and exiting the application
- A `Processor` menu for changing the processor state and visualizing it
- A `Help` menu for accessing information of interest to the user

The `MenuHandler` action listener, and inner class of YAMSGui, invokes the necessary actions when any of these menu items are selected.

The methods available to the panels (and `MenuHandler`) are of two types. They are either methods to respond to events (such as buttons being clicked):

```
public void addFile()
public void addFileList()
public void removeFile()
public void loadFile()
public void exit()
public void reloadFile()
public void resetBreakpoints()
public void displayStats()
public void verboseOutput(boolean value)
public void about()
public void processorStart()
public void processorStop()
public void processorStep()
public void processorSkip()
```

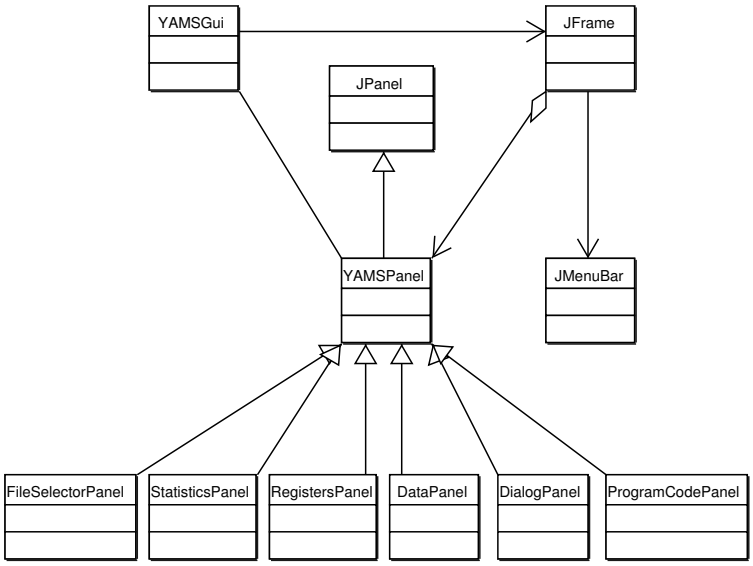

Figure 3.10: YAMS Panels and Related Classes

Or they are more general methods for controlling the program:

```
public void loadFile(File file)
public void loadFileList(File fileName)
public void setRemoveLoadStatus(boolean value)
public Processor getProcessor()
public void updateProcessorStatus(String message)
public void setProcessorSpeed(int speed)
public int getProcessorSpeedFromProcessor()
public int getProcessorSpeedFromGUI()
public void enableProgramCodeButtons()
public void disableProgramCodeButtons()
public int getCurrentLine()
public void updateBreakPointPanel()
public boolean currentLineHasBreakPoint()
public void updateStatistics()
public void regChanged(String regID)
public void memoryChanged(int address)
private void setFileLoaded(boolean value)
public boolean getFileLoaded()
public JFrame getMainFrame()
```

Most of these methods are self explanatory and they are all described in detail in the JavaDoc documentation.

Next, `YAMSGui` creates the `asm` and `processor` classes, and then it processes command line arguments.

Finally, the `init()` methods in the `data` and `code` (`DataPanel` and `ProgramCodePanel`) are called. This needs to be done because these panels can only be properly initialised after the processor has been created, since they refer to the `RegisterManager` within the processor. However, when the processor is created, it must be passed a reference to the output stream in the `dialog` (`DialogPanel`) class.

3.4.1 File Selector Panel

A `JList` is used to display all files the user wishes to have access to. The list model used is a simple `DefaultListModel`. A `JScrollPane` is required for this `JList` should the user have more files open than can be displayed in a single view.

A sub-panel (`buttonPanel`) is required for the 4 buttons associated with manipulation of the files. These are normal `JButtons`.

Some of the buttons' functionalities are not applicable to the initial state of the application, and are thus disabled on creation. However, a `ListController` (implementation of `ListSelectionListener`) listens for any changes and enables/disables the buttons (and menu items) accordingly.

The public method `addFile(File file)` is called by the `YAMSGui` class when a file is added to the list, either through a command line argument, menu option, or button click.

3.4.2 Program Code Panel

This panel consists of two `JPanels`.

The first contains a label indicating the status of the processor, four `JButtons` to control it, and a `JSlider` to change it's speed. The buttons must be disabled until a program is loaded. Each button has the `ProgramCodeButtonController` action listener associated with it. This listener invokes processor operations via `YAMSGui` controller. The `JSlider`, has a `SlideController` subscribed to it. When this slider is moved, its new value is set also via the YAMSGui and has the consequence of the processor running with a greater/lesser intra-instruction delay.

The second panel contains a `JTable`. Only when the `setSourcecode` method is called does the second panel become populated with a `JTable` of instructions and breakpoints. The internal modelling of the table is handled by the `BreakPointTable` class, an extension of the `AbstractTableModel` given in the Swing package. Breakpoint setting is handled by `BreakPointTableController` which listens for changes to the state of the group of checkboxes. On checking or unchecking any of the boxes, the `BreakPointTable` is updated. This panel requires a `JScrollPane` so that all lines of a long program can be viewed.

3.4.3 Data Segment Panel

As mentioned earlier, this panel cannot be initialized until the `processor` has been created, since this display gets it's information from the `MemoryManager`.

The code to display the data was taken from a set of classes which extend the JTable, called JHexTable [5]. It gets data from a `HexData` class, which we programmed to retreive data from the `MemoryManager`.

3.4.4 Register Contents Panel

Again, little can be done on creation of the Registers panel. After the processor has been created, a `JTable` is created with two columns. This time, a `RegisterTableModel` extends the `AbstractTableModel` and retreives the data from the `RegisterManager`.

The method `regChanged(String regID)` updates the table when a change is made to a register.

3.4.5 Statistics Panel

`JLabels` are created for each of the statistics of interest. A public method `update()` is called by the `CycleManager` after each instruction is executed.

Also included is a `JButton` which launches the graphs window (see 3.4.6). Rather than have a separate class for handling clicks of this button, the `StatisticsPanel` class itself implements the `ActionListener`.

3.4.6 Graphs Popup Window

Three panels are created and added to this `JFrame`, and the `JTabbedPane` controls the mutually exclusive visibility of them. All panels contain a `JEditorPane`. The `JEditorPane` has the advantage of the ability to render HTML text, unlike `JTextArea`. Thus bar charts can be displayed using `div` tags with dimensions which reflect the statistics of interest.

The `RegisterGraphPanel` is passed the `processor` itself as a parameter on creation as it needs to obtain data from both the `StatisticsManager` and `RegisterManager`, both of which are accessible via the `processor`. The `render()` method accumulates the html code generated and this is set as the `JEditorPane` text at the end of the constructor.

The `LineAccessGraphPanel` is also passed the `processor`, but also requires access to the individual instructions of a program and so is passed the `LineList` corresponding to that program. The `render()` method iterates through the `LineList` and generates the code of an HTML table with bars reflecting the execution frequency of each line. Note that directives, a subclass of an instruction in our implementation, are omitted as they are of no interest to the user when it comes to compiler optimization.

[5] $http://www.fawcette.com/javapro/2002\_02/magazine/columns/visualcomponents/default\_pf.aspx$

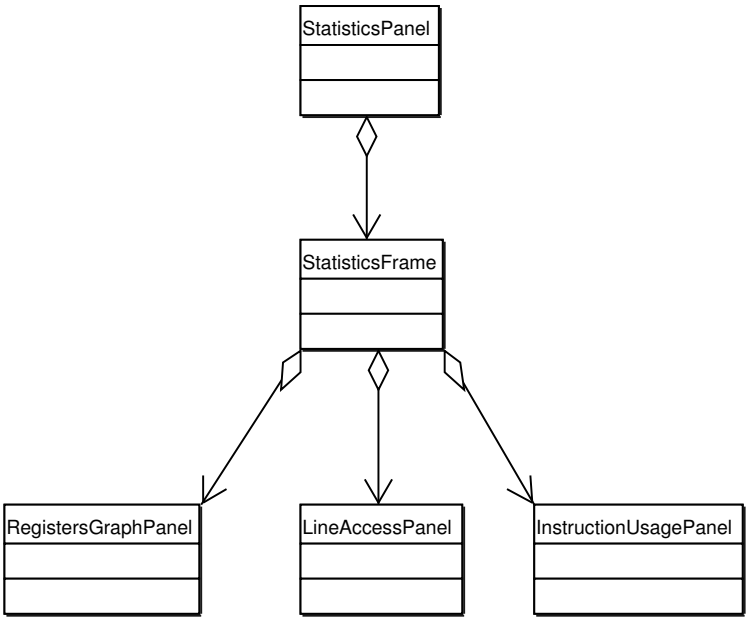

Figure 3.11: Graph Related Classes

The `InstructionUsageGraphPanel` follows the same format as the `LineAccessGraphPanel` but obtains its data from the statistics manager, accessible from the processor. This data is essentially two arrays, one for the names of the instructions and a corresponding one for the number of times that instruction has been used.

3.4.7 Dialog Panel

This panel contains a `JTextArea` within a `JScrollPane`. The class contains a `ConsoleStream` which is an extension of an `OutputStream` so that the `processor` can use a standard `PrintStream` to write it's output to, both in console mode and in GUI mode.

3.4.8 Processor Handler

This class controls the running of the `processor`. Once created, the `run()` method is immediately called. Since the `running` variable is initially false, the thread goes into an infinite loop, doing nothing except sleeping for half a second on each iteration.

There are three ways to start the processor. `ProcessorStart()`, `runOnce()` and `runSkip`. These cause the processor to run continuously, run one line, or skip one line respectively. The `destroy()` method sets `alive` to false and hence causes the `run()` method to exit.

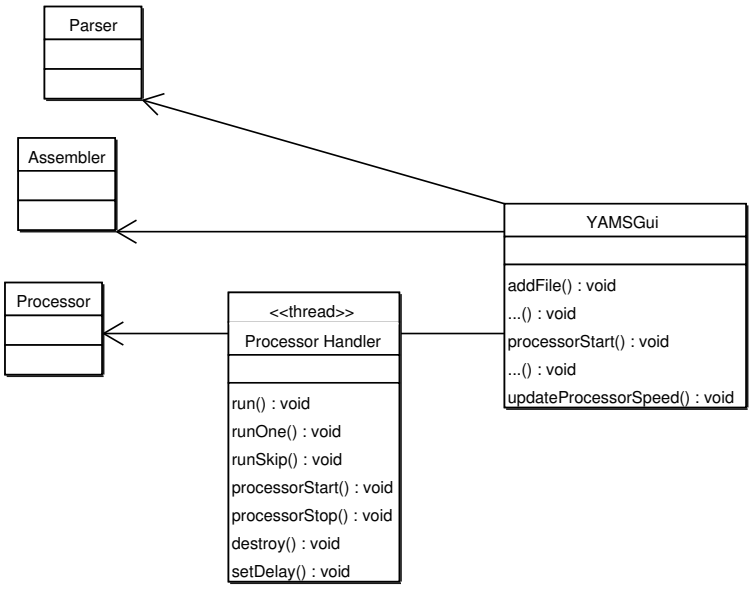

Figure 3.12: Processor Handler

Chapter 4

Using the Software

4.1 Welcome to YAMS!

YAMS is a simulator for the MIPS processor, which runs a RISC (Reduced Instruction Set Chip) architecture. As a simulator, it's main task is to correctly simulate a given set of assembler instructions and produce any necessary output. In addition, there are many features that let you see exactly what is happening as each instruction is executed, such as register and memory contents. YAMS also includes a comprehensive statistics module so you can see just how efficient the assmebly code is.

The YAMS package will work in two formats - in a console that will run from a command line, and in a user friendly graphical version, so you can see clearly what is happening inside the processor as the assmebly program is being executed.

4.2 Requirements For All Versions of YAMS

YAMS is written in Java. Hence it requires a JRE (Java Runtime Environment) to run.

If you are unsure if you have one, type in console and fingers crossed:

```
$ java -version
```

If you don't get something like `Command not found`, YAMS should run happily on your machine.

Otherwise, you'll need to download and install a JRE from sun at: http://java.sun.com.

4.3 Requirement For GUI Version

- Microsoft Windows Users will not need to do any additional configuration
- Unix/Linux users will need to start a X-Window session in order to use the GUI version.

4.4 Using YAMS - Console Version

4.4.1 Command Line Options

You can run YAMS from the command line by navigating to the directory where the executable is held, and typing YAMS -console, along with the following options:

```
YAMS [OPTIONS] [FILENAMES]
```

```
[Options]
```

```
-s, --stats
```

Outputs statistics from the run of the program

```
-o, --output <filename>
```

Outputs statistics to a specified file

```
-v, --verbose
```

Runs the program in verbose mode. Recommended only for advanced users or debugging as this can output a lot of text.

```
-if <file>
```

Input a list of files - the proceeding filenamevis a text file containing a list of files that should be executed by the simulator.

```
[Filenames]
```

Input files - One or more input files can be specified to be run by YAMS. If more than one is specified, they will be run in turn and the output for each displayed in the console window.

Examples:

```
YAMS -console --stats -if filelist.txt
YAMS -console file1.asm file2.asm
```

Handy Hint

If there is too much text output on the screen, you can redirect the output to go to a seperate file so you can look at it in a text editor instead. Do this by using the output redirect character '¿'. In the example above, this would be:

```
YAMS -console --stats -if filelist.txt > outputfile.txt
```

4.4.2 Interpreting the Output

Standard Mode

When running in the console mode on a single file, YAMS will first parse the file to make sure it is syntactically correct and all the instructions are valid. If an error is encountered, the type of error and line number in the assembly program where it was found are reported on the screen. For example:

```
=====STARTING NEW FILE=====
Input File: square.asm
Parser Error:Parse error at line: 14
Unsupported Instruction 'lkj'
```

If no errors are found, the output by YAMS will just be the same as the output of the program. An example run is shown below:

```
=====STARTING NEW FILE=====
Input File: square.asm
1111111111
0111111111
0011111111
0001111111
0000111111
0000011111
0000001111
0000000111
0000000011
0000000001
0000000000
```

Statistics Mode

When the statistics option is turned on, YAMS will also output the following statistics after the file has been executed : - How many times each line of the program was executed - How many times each instruction was used - How many times each register was used - The total number of CPU cycles required - The total memory required.

An example of the output is show below:

```
=====STARTING NEW FILE=====
Input File: EvalExpr.asm
1234 <--Output of the program

==============
  STATISTICS
==============
```

```
------------LINES EXECUTED------------

Line                                    Count
9    li $t0,10:                           1
10    sw $t0,_x:                          1
11    li $t6,1:                           1
12    lw $t7,_x:                          1
13    mul $t5,$t6,$t7:                    1
14    li $t6,2:                           1
15    add $t4,$t5,$t6:                    1
16    lw $t5,_x:                          1          Number of
17    mul $t3,$t4,$t5:                    1     <--  times each
18    li $t4,3:                           1          line executed
19    add $t2,$t3,$t4:                    1
20    lw $t3,_x:                          1
21    mul $t1,$t2,$t3:                    1
22    li $t2,4:                           1
23    add $t0,$t1,$t2:                    1
24    sw $t0,_x:                          1
25    lw $a0,_x:                          1
26    li $v0,1:                           1
27    syscall:                            1
28    la $a0,_newline:                    1
29    li $v0,4:                           1
30    syscall:                            1
31   # exit call to Stop the program
32    li $v0,10:                          1
33    syscall:                            1


-----------INSTRUCTIONS USED-----------

Instr   Count
ADD:    2
ADDI:   9
BEQ:    5                    Number of
BNE:    3               <--  times each
LUI:    1                    instruction used.
LW:     7
ORI:    6
SW:     3

------------REGISTERS USED------------

Reg   Count
$0:   27
```

```
$1:   4              Number of
$2:   4         <--  times each
$4:   4              register used
$8:   2
$9:   8
$10:  2

CPU CYCLES: 313

MEMORY USED: 65 WORDS
```

4.5 Using YAMS - GUI Version

4.5.1 Command Line Options

To use the GUI version of YAMS, navigate to the directory where the executable is held, and type YAMS -gui, along with the following options:

```
YAMS -gui [-if <file>] | [FILENAMES]

-if <file>
```

Input a list of files - the proceeding filename is a text file containing a list of files that should be executed by the simulator.

```
[Filenames]
```

Input files - One or more input files can be specified to be run by YAMS. If more than one is specified, they will be run in turn and the output for each displayed in the console window.

The graphical interface will then launch and show the screen below:

4.5.2 Adding Files to the File Selector Box

If input files were specified at the command line, they will be displayed in the File Selector Box. Otherwise, files can be added to the File Selector Box by clicking Add File and then using the dialogue box to add a file.

Here you can select an assembly file to load into the simulator.

To add a list of files, click the 'Add Filelist' button, and select the file that contains the list of assembly files you wish to load. The filelist should contain the file names of the assembly files, one per line.

If you wish to remove a file from the list, select it in the list box, then click the Remove File button below.

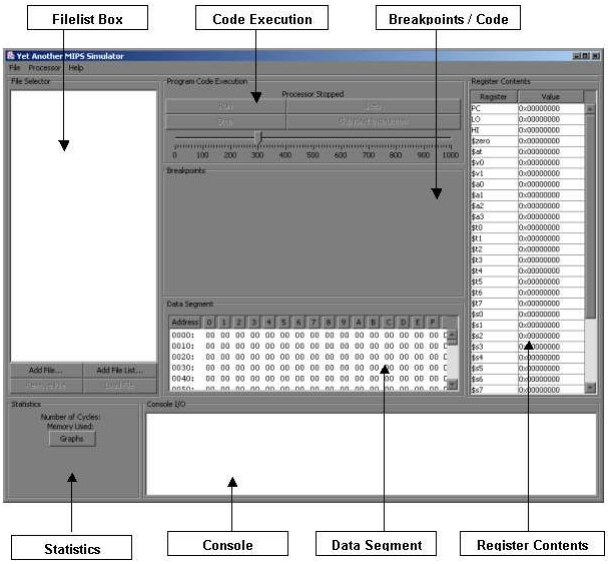

Figure 4.1: Graphical User Interface

4.5.3 Loading a File

Once you have chosen the file you wish YAMS to simulate, you can then load the file by pressing the 'Load File' button.

The program code, memory locations and register contents are shown once the file is loaded, as can be seen below, with the first line of code highlighted.

4.5.4 Adding Breakpoints

Adding a breakpoint to the program is simple. Just use the checkbox that is next to the line you where you wish to pause execution. When the simulator reaches here, you can examine the contents of the register and memory locations.

4.5.5 Controlling Execution

Speed

The speed of the processor can be controlled by the slider bar - a larger value means a longer gap between each instruction so you are able to execute the program in 'slow motion'.

Run

Press the Run button to begin the execution of the assembly program. The execution of the program is done slowly line by line so that you can see each line being executed individually and the contents of the registers and memory

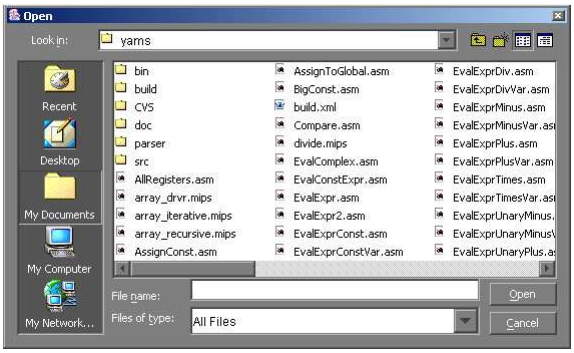

Figure 4.2: Adding a File

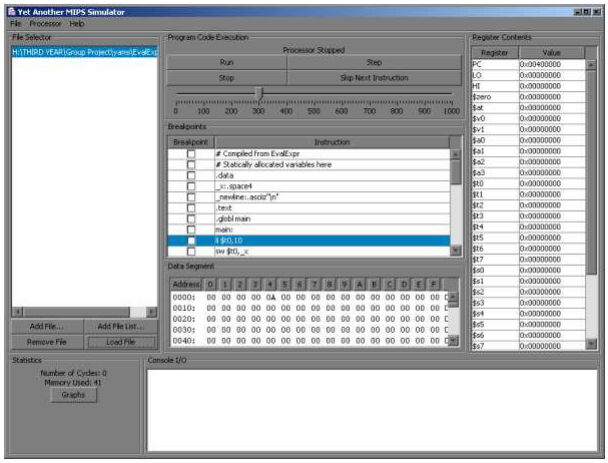

Figure 4.3: Loading a File

changing. The current line being executed is shown by the highlighted line in the program code.

Stop

To Stop the execution of the program, press the 'Stop' button. The instruction that is currently being executed will remain highlighted. From here, it is possible to resume the execution of the program by pressing 'Run', run just the next instruction by pressing 'Step' or skip the next instruction by pressing the 'Skip Next Instruction' button.

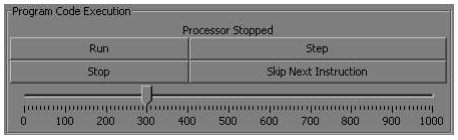

Figure 4.4: Controlling Execution

Single Step

If you wish to single step each line of the program, first of all, press the Stop button which will pause the programs execution. It is then possible to use the Step button to execute each instruction one at a time. Note that one line of assembly code can be made up of several processor operations, so pressing Step will not always move to the next instruction, but will execute the next processor operation for that line.

Skip

Pressing 'Skip Next Instruction' will ignore the next insturction that is to be executed. It is recommended that this feature is used while in stepping mode so that it can be seen clearly which instruction is to be skipped.

4.5.6 Interpreting Output

Console Output

The Console I/O box is used for 2 things: - Error messages from the parser - if an error was encountered while parsing the assmebly file, the type of error and related line number are displayed. Correct the error then try loading the file again.

- Output of the program - during the execution of the program, any output is displayed in the Console I/O box.

Statistics

Statistics for the run of the program are available in the Statistics Panel of the interface. This shows the number of CPU cycles required so far for the program executing, and the total amount of memory used by the program. These are updated constantly as the program is executed.

Pressing the 'Graph' button will display a window with 2 tabs: - How many times each register was used - How many times each line was executed

This can be useful in seeing where loops are in the program as the lines in the loop are executed more often than the rest of the instructions.

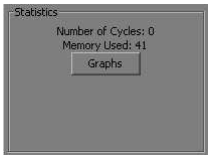

Figure 4.5: Statistics Panel

4.6 Adding Instructions

The core functionality of YAMS is extensible and has been designed so that advanced users can add new instructions to its framework. The purpose of this aspect to the application is to provide flexibility, users can add new Regular and Pseudo instructions to the system. This is to cater for the differing standards within MIPS instruction sets and allows the application to be specifically adapted to certain situations (i.e. restrict the instructions that can be used to encourage low-level optimisation of programs). A potential further extension of this implies that an entirely new instruction set (different to MIPS) could be developed and tested within the YAMS framework, taking into account the restraints of the MIPS processor.

The following guide will explain how to add a new instruction to the existing application, providing information on the three specific areas such an addition will have a significant impact upon: the Parser, the Assembler and the Processor.

4.6.1 Overview

Fundamentally the process of adding an instruction can be broken down into four main stages:

1. Parser Extension
2. Assembler Extension
3. Processor Extension
4. Building

4.6.2 YAMS and XML

As has already been mentioned, YAMS has been designed to work flexibly with different instruction sets. To achieve this, a central XML repository of data regarding each instruction is maintained within the application containing all relevant details and some code. The various software components behave according to the data contained within this repository. This behaviour is achieved in two main ways:

1. Specific Java Code

   Some project components will require specific code handlers to determine their behaviour on encountering a specific instruction. YAMS has been designed such that this code can be directly contained within `<Javacode></Javacode>` within the XML file. XSLT will then be used to re-generate the specific classes required for YAMS to handle the newly added instructions. Thus, it will be necessary to perform a recompilation of YAMS before the changes to the project will take effect.

   Software components affected: Parser, Processor (Instruction and System-call Handlers)

2. Real-time XML Parsing

   Some project elements will not require specific code, and instead need only to refer to the data contained within the XML Repository. In these situations, we can make use of XML DOM parsing to read in the data from the file at startup and fill object tables with the required information.

   Software components affected: Assembler, Statistics Manager (Processor sub component)

Therefore, to add a new instruction (or manipulate the current instruction set), the advanced user need only design the instruction, modfiy the XML file and then recompile YAMS. There are specific scripts available within the YAMS package to make this as user-friendly as possible.

4.6.3 Step-by-step Design / Addition of Instruction

The following guide explains how to add an instruction to the system step-by-step (R steps are REQUIRED fields that are needed by YAMS). When constructing the XML representation it would be useful to refer to the quick-reference tables included at the conclusion of this section of the guide, which will indicate which tags need to be filled out for which instruction, and their allowed content types.

The general XML structure required is shown in figure 4.6.

FRAMEWORK MODIFICATION

Some parts of the XML Instruction representation are generic, and will be used by multiple software components within YAMS (Parser,Assembler,Processor), these tags are described below.

1. R - Instruction Name: `<Name>`

   This field contains the name of the instruction, as will be used within the MIPS code e.g. "add","neg" etc.

```
<Instruction>
  <Name> ......... </Name>
  <OperandTypes> ......... </OperandTypes>
  <Type fixedRt="BOOLEAN"> ......... </Type>
  <Javacode></Javacode>
  <CoreMachineCode> ......... </CoreMachineCode>
  <MachineCodeRepresentations>
    <Representation>
      <OperandsCoding> ......... </OperandsCoding>
      <MachineCode> ......... </MachineCode>
      <Operands>
        <Op>
          <Number> ......... </Number>
          <Type> ......... </Type>
          <Mask> ......... </Mask>
          <EncodeBits> ......... </EncodeBits>
          <OutputBits> ......... </OutputBits>
          <OffSetMode> ......... </OffSetMode>
        </Op>
        ......
      </Operands>
    </Representation>
    ......
  </MachineCodeRepresentations>
  <Help>
    <FullName></FullName>
    <Format></Format>
    <Description></Description>
  </Help>
</Instruction>
```

Figure 4.6: XML Template

2. R - Type: `<Type>`

   Tells Assembler/Processor/Parser what type of instruction this in terms of structure. This is the FIRST major instruction decision to be made, and will have an impact on other sections of the XML.

   Regular A REGULAR INSTRUCTION has one set of possible operands, therefore one machine code representation.

   Details on regular instructions are required in the following software components:

   - included in parser (usable in mips code)
   - included in assembler (encoded to machine code)
   - included in processor (processor knows what specific steps to take to execute this instruction)

   Extended An EXTENDED INSTRUCTION has multiple representations depending on the different operand combinations, thus will require and entry for the Core Machine Code field.

   More than one set of possible operands e.g. add REG REG REG / add REG REG IMMEDIATE -¿ One machine code representation per possible operand construct NOT a pseudoinstruction because underlying instruction is still atomic but for some operand representations, other steps have to be taken to cater for the different operand before the core function is executed

   e.g. there is a specific instruction machine code representation add REG REG REG = 00-9o-0os, however if add REG REG IMMEDIATE is required as a variation on this instruction a li REG IMMEDIATE must 'first' be executed before the regular add can be made. To cater for this conceptual difference, these are called Extended instructions.

   NB with extended instructions, since they still have a core instruction (e.g. add still has atomic underlying expression) it is now NECESSARY to fill in the CoreMachineCode tag (described in a moment) included in parser (usable in mips code).

   Details on extended instructions are required in the following software components:

   - included in parser (instruction is usable in mips code)
   - included in assembler (encoded to machine code)
   - included in processor (processor knows what specific steps to take to execute the CORE VERSION of this instruction)

   Pseudoinstruction A PSEUDOINSTRUCTION is not specifically executed by processor,rather is made up of other atomic instructions.

   Can be one set of operands or more than one set of operands (and therefore representations), and has no core representation (i.e. there is no explicit core machine code representation for this instruction).

Therefore it is composed of a series of other atomic instructions that overall have the required effect of the instruction. It will not be known the processor, since it will already know the core instructions it uses.

Details on pseudoinstructions are required in the following software components:

- included in parser (usable in mips code)
- included in assembler (encoded to machine code)
- NOT included in processor

There also exists an attribute for the Type tag which must be filled in as fixedRt="true" for instructions

The fixedRt attribute must be set to true for I type branch instructions that use the Rt field of the instruction to identify the branch type. This attribute is require to identify these instructions for requiring extra decoding in the Instruction Handler. The only MIPS instructions to require this attribute are: bltz, bltzal, bgez and bgezal.

If the attribute is not specified, the default value is false.

3. D - Core Machine Code : `<CoreMachineCode>`

   This is ONLY required when an Extended instruction is created (multiple representations depending on operands, while using underlying machine code representation). It contains the machine code for the actual instruction, not including an pre-instruction execution steps that need to occur for more complicated combinations of operands e.g. add REG REG REG has a specific machine code representation, but add REG REG IMMEDIATE uses this representation plus some other atomic instructions to cater for the different operands.

   The value of this field should be the simplest representation within the MachineCodeRepresentations/Representation/MachineCode tag, and should be 1 x 32 bit machine word.

PARSER MODIFICATIONS

The Parser will use method 1 (see earlier) to obtain the information from the XML file. Essentially it will use an XSLT transformation to convert selected data fields contained within the XML to parts of the Parser handler. This handler will deal with recognising an instruction and will be able to instruct the parser how to recognise and validate the presented instruction. This information includes the name of the instruction, and the possible operands that it takes. The following field is solely used by the Parser in this respect:

4. R - Operand Types : `<OperandTypes>`

   This indicates to the parser what number and type of operands are being used for the specified instruction. For the purposes of parsing (decided to

Type	Addressing Mode
Immediate	16 bit Immediate offset, 32 bit Absolute address
Immediate(Register)	Immediate (16/32 bit) plus register value gives absolute address
Register	Register value gives absolute address
Label	Label (Evaluate offset)
Label+-Immediate	Label (Evaluate offset) +- 16/32 bit offset
Label+-Immediate(Register)	Label (Evaluate offset) +- 16/32 bit offset

Table 4.1: Addressing Modes

be more of a syntax check than a semantic check, which will be dealt with more within the assembling phase), there are four main operand types:

Operand.TYPE_REGISTER Indicates the operand is a register (from available list in MIPS architecture)

Operand.TYPE_IMMEDIATE Operand is an immediate (NB encompasses all forms of immediates, no range checking this occurs within the assembler)

Operand.TYPE_LABEL Operand is a Label (String referring to another point within the program)

Operand.TYPE_ADDRESSING Operand is an Addressing Operand, referring to some address within memory. This can take one of 6 subforms (please consult MIPS addressing documentation and following table for further information):

The addressing mode can either be an absolute value, an offset from the current text address or an offset from a fixed block address (for YAMS it is 0x10008000 in main memory).

Key: where I(R) = I+R

This will tell the parser exactly what operands to expect when an instruction of this type is presented within the MIPS code. To specify a representation, place comma separated operand types (in corresponding order to desired instruction) within the ¡OperandTypes¿¡/OperandTypes¿ tags.

Cardinality:

- For Regular instructions there will only be one valid operands representation, thus one `<OperandTypes>` entry.
- For Extended/Pseudoinstructions there can be more than representation for the operand types thus there will be a corresponding `<OperandTypes>` entry for each one.

Example: Instruction Name = add

```
Regular Add instruction dealing with registers
<OperandTypes>Register,Register,Register</OperandTypes>
Add instruction allowing an immediate to be included
<OperandTypes>Register,Register,Immediate</OperandTypes>
```

ASSEMBLER MODIFICATIONS

The assembler requires complex and specific information regarding the structure and operand types involved in an instruction. It utilies method 2 to obtain information from the XML file, reading it on startup, parsing the information and loading it into table objects for further reference to during the assembling phase. The following sections must be specifically included to tell the assembler exactly what to do with the operands you provide it.

5. R - Machine Code Representations: `<MachineCodeRepresentations>`

   The assembler needs to know what the specific machine code representation(s) for the current instruction is (are).

   If the instruction is a Regular Instruction, then there must be ONLY ONE Representation node within this higher level tag. If however, the instruction is an Extended or Pseudo Instruction, then there must be more of these representations to cope with the different alternatives.

   Operand Semantic Differences

   It is important to note, that when we come to defining the details of the instruction for the assembler, our notion of operands must change slightly. Within the parser operand definition tags (

   ```
   <OperandTypes>
   ```

) we only consider there to be four possible operand types. However, the assembler must take into account the fact that there will in fact be different machine code representations for separate instructions using the same parser operand type.

   For example, the parser has no notion of there being a difference between a 16 and 32 bit integer, however when it comes to assembling we can only deal with 16 bit integers in the majority of our operations. Thus we must solve this problem by using other commands to load in order the bottom half and top half of 32 bit integers into a register. Once there we can manipulate it to perform the required operation. So that our generated code remains optimal (i.e instructions using only 16 bit immediates dont need to perform as many instructions as 32 bit immediates will), we separate dealing with 16 and 32 bit immediates into different

   ```
   <Representation>
   ```

tags.

The required contents of the

```
<Representation>
```

node(s) within this lower level will be discussed within this section, but essentially they are differentiated from each other by an encoded representation of the operands (discussed later) and any further details regarding these operands are contained within its

```
<Operands>
```

tag. The following example illustrates this and the points discussed regarding the operands.

e.g. "add" instruction has two ¡OperandType¿ tags and thus must have two representation tags to correspond to these:

....

```
<OperandType>Register,Register,Register</OperandType>
<OperandType>Register,Register,Immediate</OperandType>
```

....

```
<MachineCodeRepresentations>
  <Representation>
    <OperandsCoding>111</OperandsCoding>
    <MachineCode>000000bbbbbcccccaaaaa00000100000</MachineCode>
    <Operands>
    </Operands>
  </Representation>
  <Representation>
    <OperandsCoding>110</OperandsCoding>
    <MachineCode>001000bbbbbaaaaaccccccccccccccc</MachineCode>
    <Operands>
      <Op>
        <Number>3</Number>
        <Type>0</Type>
        <Mask>1111111111111111</Mask>
        <EncodeBits>16</EncodeBits>
        <OutputBits>16</OutputBits>
        <OffSetMode>2</OffSetMode>
      </Op>
    </Operands>
  </Representation>
```

```
  <Representation>
    <OperandsCoding>11a</OperandsCoding>
    <MachineCode> 00111100000aaaaaddddddddddddddd
                  001101aaaaaaaaaaccccccccccccccc
                  000000bbbbbaaaaaaaaaa00000100000
    </MachineCode>
    <Operands>
      <Op>
        <Number>3</Number>
        <Type>a</Type>
        <Mask>11111111111111111111111111111111</Mask>
        <EncodeBits>32</EncodeBits>
        <OutputBits>32</OutputBits>
        <OffSetMode>2</OffSetMode>
      </Op>
    </Operands>
  </Representation>
</MachineCodeRepresentations>
```

As can be seen from the above example, "add" only has two possible parser configurations:

add REGISTER REGISTER REGISTER
add REGISTER REGISTER IMMEDIATE

However, the immediate in the second instruction could be dealt with differently depending on whether it is a 16 bit or a 32 bit immediate, as has been reflected by the fact that there are THREE ¡Representation¿ nodes and only TWO ¡OperandType¿ nodes. Therefore as can be seen there are conceptually different concepts between how the Parser and Assembler see operands, and when designing an instruction it would be a good idea to deal with the sections differently.

Please note it is also perfectly possible to restrict the use of immediates to only 16 bit integers, by only including its representation and not the code for 32 bit integer representations. Given this information in the XML, YAMS will infer that the instruction is designed to only take 16 bit ntegers and will throw an error when presented with 32 bit ones instead.

Representation

The basic structure of a ¡Representation¿ node is highlighted below:

```
<Representation>
  <OperandsCoding> ......... </OperandsCoding>
  <MachineCode>
  ..................
  </MachineCode>
  <Operands>
```

```
    <Op>
      <Number> ..... </Number>
      <Type> ..... </Type>
      <Mask> ................... </Mask>
      <EncodeBits> ..... </EncodeBits>
      <OutputBits> ..... </OutputBits>
      <OffSetMode> ..... </OffSetMode>
    </Op>
  </Operands>
</Representation>
```

When adding an instruction, as has already been mentioned, you must add Representation nodes for each of the different operand combinations that there will be (from extended Operand set).

For each representation we must fill out the following information:

(a) Operands Coding

Each representation is uniquely identified by a string of characters which represent the operand types that this representation of the instruction will be catering for. The following table contains the available operand types that we can use in our operand code:

Table : Operand Coding Set

Contains a list of available operand codes that we can use to uniquely describe our representation. As has already been mentioned there are differences between the Parser and Assembler representations of operands, and the following table matches the two distinct sets.

Example: for the "add" instruction

i. add REGISTER REGISTER REGISTER has only one possible representation for registers, thus `<OperandCode>111</OperandCode>`

ii. add REGISTER REGISTER IMMEDIATE has two possible representations, 16/32 bit immediates, thus

`a) <OperandCode>110</OperandCode>`

`b) <OperandCode>11a</OperandCode>`

(b) MachineCode

The representation will contain a ¡MachineCode¿ tag which contains the underlying machine code required to carry out the instruction. However, before receiving the actual instruction it is impossible to identify what the values of these operands are. Therefore the machine code will contain "gaps" in specific parts, which can be filled in by the assembler when it works out the actual values of each operand. These actual values need to be mapped into their correct "gaps" in the code. Thus within the XML file, these gaps are represented as string of characters (from following table) of the required length.

Character Code	Meaning	Corresponding Parser Operand
0	¡=16 Bit Twos Complement Immediate	Immediate
a	32 Bit Twos Complement Immediate	Immediate
1	Register	Register
2	Label	Label
3	Address	Addressing
4	Add: Immediate	Addressing
5	Add: Immediate(Register)	Addressing
6	Add: Label	Addressing
7	Add: Label_Plus_Immediate	Addressing
8	Add: Label_Plus_Immediate_Register	Addressing
c	Add: Label_Minus_Immediate_Register	Addressing
9	Add: Register	Addressing
b	no operands	Addressing

Table 4.2: Operand Coding Set

Character	Operand Number
a	1
b	2
c	3
d	4
e	5
f	6
g	7
h	8
i	9
j	10

Table 4.3: Characters used within strings to indicate a position for operand substitution by the assembler

As can be seen, YAMS has been designed to cater for instructions with up to 10 operands if necessary.

Example : In YAMS, given an instruction

instruction OPERAND_1 OPERAND_2 OPERAND_3

the assembler will retrieve the required representation from the Pre-Loaded XML Tables by working out the required operand code (see above) for this specific "instruction." This will return a pre-defined machine code representation with "gaps" where the computed operands need to go. The assembler thus calculates the machine code values of the user supplied operands and then one-by-one substitutes their values into the required places in the machine code. For example, the machine code for our generic "instruction" example is as follows

010010aaaaabbbbbccccc00000111111

YAMS will then calculate the values of OPERAND_1, _2 and _3 and then map them to their correct positions within the code.

OPERAND_1 = 00001 = aaaaa OPERAND_2 = 00100 = bbbbb OPERAND_3 = 11111 = ccccc

01001000001001001111100000111111

Therefore during the design of an instruction, effort must be made to ensure that the correct operand gaps are left in the right places ready for substitution at run time.

(c) Operands

As has been seen, the specific type of the operand (e.g. immediate, label etc) has already been decided upon for the current instruction being added. However, the assembler will require further information on some of these operands in order to calculate their values correctly.

For example if we have an Immediate value for one of the operands, it is possible we may not want to include all of this immediate value when we write it to memory. Such a situation may occur when considering offsets. If our immediate is a value of 400 bytes = 110010000 bytes, since we are using aligned memory access, then we could feasibly drop the last 2 bytes of our immediate and save some space for further information in our instruction. In such cases, YAMS provides the facility for an instruction writer to tell the assembler to do so.

However, this only applies to some of the operand types that we are using, the following table indicates for which Operands an `<Op>` tag should be included.

For each of the YES cases above, an `<Op>` tag must be filled out as set out below:

Operands

An Operand tag takes the following form:

```
<Op>
<Number> ..... </Number>
```

Operand	Description	Required <Op> Description?
0	<=16 Bit Twos Complement Immediate	Yes
a	32 Bit Twos Complement Immediate	Yes
1	Register	No
2	Label	Yes
3	Address	Yes
4	Add: Immediate	Yes
5	Add: Immediate(Register)	Yes
6	Add: Label	Yes
7	Add: Label_Plus_Immediate	Yes
8	Add: Label_Plus_Immediate_Register	Yes
c	Add: Label_Minus_Immediate_Register	Yes
9	Add: Register	Yes
b	no operands	No

Table 4.4: Requirement for Op Tags for Operand Types

```
<Type> ..... </Type>
<Mask> ................... </Mask>
<EncodeBits> ..... </EncodeBits>
<OutputBits> ..... </OutputBits>
<OffSetMode> ..... </OffSetMode>
</Op>
```

Essentially, the assembler has already been told what machine code representation it has to map the operands to, and the information contained within the <Op> tags for each operand explains to what level of accuracy to perform calculations, what offset mode to be using if we are calculating symbolic labels and which specific bits to use in the final substitution.

i. Number
This refers to which operand within the <OperandCode> we are referring to, via indexing. The first operand takes a Number=1, the second Number=2 and so on.

ii. Type
This refers to the operand type which we are placing constraints upon (from extended Assembler Operand Type list described already)
Example: for the "add" instruction, under the 11a representation, the XML will contain:

```
<Op>
<Number>3</Number>
<Type>a</Type>
```

. . . . .

iii. Encode Bits
This field specifies to how many bits we wish to encode the binary value which YAMS' Operand Handler will be calculating the current operand to. YAMS will thus throw an error if the provided value cannot be represented within this range. For example, for a Label operand, we are dealing with an 16 bit "gap" in which we can place our offset. Therefore if we encode to 16 bits accuracy we make sure that the label is within our addressable range.
The number of Encode Bits must be ¿= OutputBits field (number of bits we use in the substitution).

iv. Mask
The Mask will highlight which bits of the encoded representation we wish to include in the final output for our operand (the value we will actually be substituing into the predefined machine word. This is done bitwise:
1 (at index i) Indicates that the encoded bit at index i IS included in the substituting value 0 (at index i) Indicates that the encoded bit at index i IS NOT included in the substituting value
Therefore, the restriction here is that the number of 1s in the Mask must be equal to the OutputBits field.
For example, to mask out the bottom two bits of a 16 bit number, use 1111111111111100 as the Mask value.

v. Output Bits
This indicates the final number of bits that we will be using as the substituting bit string (the final value we substitute into the machine word as the operand). It must equal the number of 1s we have in the Mask field.

vi. Off Set Mode
For certain operands (e.g. Labels), the value of the provided operand that we want to place into our instruction will need to be a calcualated offset. This offset can be from the current line address or maybe from the fixed block address of 0x10008000 used in MIPS. On other occasions we may want to calculate the absolute address. Therefore, YAMS provides instructions with three calculation modes for their operands, as below:

PROCESSOR MODIFICATIONS

The processor must be able to handle instructions correctly, and the XML file will contain all the data regarding each instruction. The processor will read this data using Method 1, so at compile time the java code contained within the XML should be able to be placed within the Processor Instruction Handlers. This code is contained within the `<Javacode>` tag for each instruction, highlighted below.

Mode	Title	Description
0	Current Position Mode	Will calculate the offset based on the current line address (e.g. jump)
1	Block Address Mode	Calculates offset from 0x10008000
2	Absolute Address Mode	Calculates the absolute address of the operand

Table 4.5: Offset Modes For Operands

Instruction Type	Javacode Tag
Regular	Required
Extended	Required
Pseudoinstruction	Not Required

Table 4.6: Requirements For Javacode Modification

6. D - Javacode: `<Javacode>`

   For newly added REGULAR and EXTENDED instructions, the processor will have to be able to carry out some new functionality in order to execute the required behaviour for the specific instruction. However, for PSEDUOINSTRUCTIONS, this will not be necessary since they will be defined in terms of already existing instructions.

   In order to implement the functionality of a new instruction, it will be necessary to understand the functions available to this new instruction through the processor. There are interfaces available in the YAMS procesor documentation that highlight the features of the simulated mips processor, and they can be used to construct the java code appropriately.

   For an instruction to be handled by the processor correctly it should have a valid and unique machine code representation. For Regular instructions, this is described in the MachineCode tag in the MachineCodeRepresentation. For Extended instructions, this is found in the CoreMachineCode tag. The processor does not generate handling code for pseudoinstructions, for these are assembled into their consitituent instructions before they can be executed.

   Java code is used to describe the semantics of the instruction. The semantics will likely include reading values to/from registers and/or memory, depending on what the instruction is designed to do. The code is placed inside the ¡Javacode¿ opening and ¡/Javacode¿ closing tags.

   Before writing any Java code, it is recommended to be familiar with the section of this report that describes the InstructionHandler component of the Processor. Depending on the type of MIPS instruction (R, I or J) that is being added, the interfaces and variables available to the programmer vary.

mem	MemoryManagerInterface	the memory manager
regs	RegisterManagerInterface	the register manager
stats	StatisticsManagerInterface	the statistics manager
out	PrintStream	the console output
verbose	PrintStream	the verbose output
op	int	the 'op' bitfield
op_bitstr	MIPSBitstring	ditto (as bitstring)

Table 4.7: Processor Interfaces

rs	int	the first source register
rs_bitstr	MIPSBitstring	ditto (as bitstring)
rt	int	the second source register
rt_bitstr	MIPSBitstring	ditto (as bitstring)
rd	int	the destination register
rd_bitstr	MIPSBitstring	ditto (as bitstring)
shamt	int	the shift amount
shamt_bitstr	MIPSBitstring	ditto (as bitstring)
func	int	the 'func' bitfield
func_bitstr	MIPSBitstring	ditto (as bitstring)

Table 4.8: R type only

Tables 4.7, 4.8, 4.9 and 4.10 show the processor interfaces and variables available, by type of

The Java code should be placed between the `<Javacode>` and the `</Javacode>` tags.

What the Java code for new instructions should NOT contain:

- return statements
- frivolous use of the 'out' PrintStream
- attempts to reference classes outside the yams.processor package

rs	int	the first register
rs_bitstr	MIPSBitstring	ditto (as bitstring)
rt	int	the second register
rt_bitstr	MIPSBitstring	ditto (as bitstring)
i	int	the SIGN-EXTENDED version of 'immediate' bitfield
i_bitstr	MIPSBitstring	the 'immediate' bitfield as bitstring

Table 4.9: I type only

addr	int	the 'addr' bitfield
addr_bitstr	MIPSBitstring	ditto (as bitstring)

Table 4.10: J type only

- attempts to use interfaces/variables not accessible for the instruction type (R, I or J)

Violations of these recommendations will cause either a failure at XSLT transformation stage or when compiling the generated InstructionHandler.java file.

N.B. All special characters, such as & (ampersand), < (left chevron) and > (right chevron) MUST be converted to their HTML equivalent.

```
&  ->  &amp;
<  ->  &lt;
>  ->  &gt;
```

This is because XSLT has its own individual use for these characters in XSLT files.

Failure to do this conversion for all special characters will result in the following error on applying the transformation during the build process.

```
    [xslt] :  Fatal Error!  org.xml.sax.SAXParseException:
    Illegal character or entity reference syntax.  Cause:
    org.xml.sax.SAXParseException:  Illegal character or entity
    reference syntax.
```

Example Javacode for existing instructions:

```
  'add rd, rs, rt' (addition)
  'div rs, rt' (divide)
  'beq rs, rt, label' (branch on equals)

<Instruction>
  <Name>add</Name>
  ...
  <Javacode>
    regs.setReg(rd, regs.getReg(rs) + regs.getReg(rt));
  </Javacode>
  ...
</Instruction>

<Instruction>
```

```
  <Name>div</Name>
  ...
  <Javacode>
    int quotient;
    int remainder;
    int rscontents = regs.getReg(rs);
    int rtcontents = regs.getReg(rt);
    quotient = rscontents / rtcontents;
    remainder = rscontents % rtcontents;
    regs.setReg("LO", quotient);
    regs.setReg("HI", remainder);
  </Javacode>
  ...
</Instruction>

<Instruction>
  <Name>beq</Name>
  ...
  <Javacode>
    if(regs.getReg(rs) == regs.getReg(rt)) {
      int pc = 0;
      pc = regs.getReg("PC");
      pc += i * 4 - 4;
      regs.setReg("PC", pc);
    }
  </Javacode>
  ...
</Instruction>
```

7. R - Help Modification: `<Help>`

   The final stage to adding an instruction would be to add the help tags, so that the help documentation can be created appropriately:

```
<Help>
  <FullName></FullName>
  <Format></Format>
  <Description></Description>
</Help>
```

4.6.4 Building The Project

As mentioned in the YAMS spec. YAMS is built in such a way the adding and removing instructions can be done without modifying any YAMS' program code.

You'll, obviously, need to get the source code bundle.

Type
Operand.TYPE_REGISTER
Operand.TYPE_IMMEDIATE
Operand.TYPE_LABEL
Operand.TYPE_ADDRESSING

Table 4.11: <OperandType> Tag Comma Separated Available Types

Symbol	Meaning
R	Required
N	Not Required
1	1 tag only
	1 or more tags

Table 4.12: Reference Key

Software Requirement

Apache Ant is required for building YAMS.

To check whether Ant is installed on the system, type in console:

```
$ ant -version
```

If you see something similar to:

```
Apache Ant version 1.5.3 compiled on April 16 2003
```

You are good to go on and start the build, otherwise, please install Ant from the `/extras` directory in the YAMS distribution. An installation guide can be found at *http://ant.apache.org*

Doing the build

Running the following command in the root directory or the YAMS distribution (that is, where the `build.xml` locates) will build a YAMS.jar file in the build directory:

```
$ ant
```

Simple as that, your new customized YAMS is there waiting to be run.

QUICK-REFERENCE PAGE for adding an instruction

The following quick-reference tables give an overview of how to add an insruction to the YAMS application:

PARSER TABLE

Table 4.11 shows the operand types used by the Parser.

ASSEMBLER TABLES

Table 4.12 is a key for table 4.13.

Tag	Regular	Extended	Pseudo	Value
<Instruction>				
<Name>	R,1	R,1	R,1	1 x Character String
<Type>	R,1	R,1	R,1	1 x INSTRUCTION_TYPE + 1 x fixedRt="true" attribute
<CoreMachineCode>	N	R,1	N	1 x 32 Bit BitString (with possible character-gaps)
<OperandTypes>	R,1	R,*	R,*	1+ x comma separated PARSER_OPERAND
<MachineCodeRepresentation>	R,1	R,1	R,1	-
<Representation>	R,1	R,*	R,*	-
<OperandsCoding>	R,1	R,1	R,1	1+ x string from ASSEMBLER_OPERAND
<MachineCode>	R,1	R,1	R,1	1 x 32 bit BitString (with appropriate character gaps)
<Operands> see operand table				-
</Representation>				
</MachineCodeRepresentation>				
<Javacode>	R,1	R,1	N	Written java code for compilation
<Help>	R,1	R,1	R,1	-
<FullName>	R,1	R,1	R,1	1 x Character String
<Format>	R,1	R,1	R,1	1 x Character String
<Description>	R,1	R,1	R,1	1 x Character String
</Help>				
</Instruction>				

Table 4.13: XML Reference Table (INSTRUCTIONS)

Table 4.13 shows the contents and cardinality of these tags which depend on whether the instructions are of Regular/Extended/Pseudoinstruction type. For discrete value types (referred to with capital letters) see tables 4.15 and 4.16.

For every operand within the operand encoding must have an ¡Op¿¡/Op¿ in the ¡Operands¿ section (exception: register operand). There are certain restrictions upon the contents of the tags as shown in table 4.14.

PROCESSOR TABLE

4.6.5 EXTENSION: Adding A System Call

It is also possible to add a system call to the YAMS framework. In general, MIPS system calls are carried out in three stages:

1. The parameter required for the system call is placed into the register a0
2. The call code is placed into the register `$v0`

Tag	Requirements	Type	Restrictions
	0 a 1 2 3 4 5 6 7 8 c 9		
<Op>	R R N R R R R R R R R		R = Entry required, N = No entry required
<Number>	| | | | | | | | | | |	Int	Index 1+ corresponding in operand encoding
<Type>	| | | | | | | | | | |	ASSEMBLER_OPERAND	Corresponding operand encoding symbol
<Mask>	| | | | | | | | | | |	Bit String	Length(Mask) = OutputBits
<EncodeBits>	| | | | | | | | | | |	Int	EncodeBits ¿= OutputBits
<OutputBits>	| | | | | | | | | | |	Int	OutputBits = Length(Mask)
<OffSetMode>	2 2 2 n n n n n n n n n	Int	OFFSET_MODE
</Op>	| | | | | | | | | | |		

Table 4.14: XML Reference Table (OPERANDS)

INSTRUCTION_TYPE	PARSER_OPERAND
Regular	Register
Extended	Immediate
Pseudoinstruction	Label
	Immediate

Table 4.15: Discrete Value Types

ASSEMBLER_OPERAND	OFFSET MODE
0 <=16 Bit Immediate	0 Current Position Mode
a 32 Bit Immediate	1 Block Address Mode
1 Register	2 Absolute Address Mode
2 Label	
3 Address	
4 Add: Immediate	
5 Add: Immediate(Register)	
6 Add: Label	
7 Add: Label_Plus_Immediate	
8 Add: Label_Plus_Immediate_Register	
c Add: Label_Minus_Immediate_Register	
9 Add: Register	
b no operands	

Table 4.16: Discrete Value Types 2

Instruction Type	Javacode Tag
Regular	Required
Extended	Required
Pseudoinstruction	Not Required

Table 4.17: Requirements For Javacode Modification

3. A "syscall" instruction is executed

This can be illustrated in the following example code:

```
    .data
__line:
    .asciiz "Line"
    .text
    .globl main
main:    #first instruction
    la $a0,__newline
    li $v0,4
    syscall
    .
    .
    .
```

As can be seen above, the address of `_newline` is placed into the register `$a0`, next the call code 4 is placed in `$v0` (indicates to processor print an ascii to the outputstream). Then a `syscall` instruction is called, which tells the processor to look at register `$v0`, find out what kind of call to execute, and then do so on the paramter given in `$a0`.

Within the XML Repository, system calls are not stored as separate instructions within `<Instruction>` tags. Instead, they are all stored within the `<Javacode>` tag for the "syscall" instruction. Therefore, adding a system call requires altering this specific instruction.

An example view of the `Instruction_file.xml` `syscall` entry:

```
  <Instruction>
    <Name>syscall</Name>
    <OperandTypes></OperandTypes>
    <Javacode>
    switch(regs.getReg(2)) {
      case 1: {
        verbose.println("SYSCALL : print_int");
        // print_int syscall
        out.print(regs.getReg(4));
        break;
      }
      .
      .
      .
```

Therefore adding an instruction requires simply adding some more Javacode as another case statement of the format:

```
case n {
```

mem	MemoryManagerInterface	the memory manager
regs	RegisterManagerInterface	the register manager
stats	StatisticsManagerInterface	the statistics manager
cycleManager	CycleManagerInterface	the cycle manager
in	InputStream	the std input
out	PrintStream	the console output
verbose	PrintStream	the verbose output

Table 4.18: System Call Interfaces

```
verbose.println("SYSCALL : new_call");
// new_call syscall
CODE
break;
}
```

The interfaces available to the programmer are described in table 4.18

Also three extra java.io classes are imported: BufferedReader, InputStream-Reader and IOException.

In the template above, n is the call code for the `new_call` System Call we wish to implement. The `verbose.println()` statement should be included for debugging purposes (Verbose Mode can be set to on within the YAMS Gui environment). Finally, the code to carry out the system call should be included. For further information on how to implement java within the XML tags please review the processor documentation, as the interfaces outlined there will allow you to see the potential capabilities of a system call.

Finally, once the modifications are complete to the `Instruction_file.xml`, YAMS must be rebuilt to add this new capability to the processor (see earler subsection regarding "ant" rebuild of project).

Chapter 5

Evaluation

5.1 Successes

The YAMS project accomplished all that was set out in our original specification to achieve. In short, this was to produce a MIPS simulator that could be used for the second year compiler Lab. The simulator was to have both a console and GUI versions. The YAMS simulator is a unification of the two, depending on the option the simulator is started with, it runs the console version or the version extended with a GUI.

The console version has many command line options, to execute a list of files, printing statistics and logging to a file being a few examples. The GUI-enabled simulation has a very intuitive and easy-to-use interface, which is vastly superior to that offered by the current MIPS simulator of choice Spim, and XSpim.

The development and integration of the components that made up YAMS was extremely rapid. By using numerous well-defined Java interfaces, the parser, assembler and processor meshed from the first integration build. The GUI was produced extremely quickly on top of the simulator, this was made possible by the use of JTables and the existing provision of the interaction between the simulator components and a GUI, through the use of listeners and a ProcessorHandler thread.

5.2 Failures (or missed opportunities)

In the GUI, it was decided that YAMS would not highlight registers that were being read from or written to. This was because the register highlighting would be confusing when pseudoinstructions were executed, since they are seemingly atomic to users of the simulator but in fact contain many smaller, regular MIPS instructions.

It was originally intended to use ANTLR to generate the parser, however ambiguity removal from MIPS BNF grammar proved difficult. After realizing

that all input would be linear (i.e. no nested constructs), it was decided that use of ANTLR would not be necessary.

The Processor was originally intended to have an ALU class with static methods performing various logical and arithmetic operations. Sadly because MIPS has a RISC architecture there is very little code that can be common to multiple instructions, so the ALU class was torpedoed. Also the idea of having hard-coded static methods did not really appeal when YAMS started down the extension of using an XML instruction repository.

5.3 Testing

The YAMS simulator was tested against the specification throughout development to ensure a quality end product. The release build of the simulator was used in place of Spim to test four separate sets of Compiler Lab outputs. One of the these was the Lab compiler's output, the other three were produced by student-written DEC(M2) to MIPS compilers. For each file, YAMS simulated the execution correctly, and printed the correct output '1234'.

5.4 Extensions

The initial design stages for the project detailed not only the core requirements of the final product, but also included potential extensions that could feasibly be reached during the implementation of YAMS. Some of these extensions were achieved during the YAMS development, while others were not. These, along with other extensions developed along the way, will be outlined in the following section:

5.4.1 Achieved Extensions:

1. Extra Statistics

   Extra statistics, such as a tentative method of keeping count of the loops used in the code was correctly implemented within the console and GUI versions of the program. This was achieved by the Statistics Manager keeping count of the number of times any given line was executed by the processor. It was then possible to identify where the majority of processor time was being spent during any given exeuction.

   Another extension correctly implemented was keeping count of the number of times that any type of instruction was executed within any given execution. This was written into the underlying console version, and then tied into the GUI version as well.

2. Graphical Representations Of Statistics

   A further extension that would prove to be very effective in the final product was the graphical representation of statistical counts - represented

as histograms. The regsister counts, instruction counts and line counts were all represented in the GUI using this method. This proved to be very effective for understanding what was going on in the underlying MIPS code.

3. Statistical Comparison

   Another supported extension was the ability to be able to compare statistics from different runs of the programs. In the console version this was achieved through outputing a comma-separated file containing comparative statistics on number of cycles required for running these programs. Additionally, this file could be imported into Excel and a comparison graph very quickly created for visual comparison.

   In the GUI version of YAMS, it is possible to display the statistical graphs for one program next to those for another. Once again, this is another effective method for viewing the efficiency of the compilers that generate the MIPS code.

4. Multiple File Handling

   In both the console and GUI versions, YAMS provides functionality to allow the user to run the simulator with multiple files, through loading a file list. Additionally, within the GUI version, selection of specific programs from this list is also possible.

5. XML Repository

   A significant integration point for YAMS was to be able to have a single XML Repository of instructions, and allow all software components to refer to these sections. This was achieved during the last phase of project development:

   Parser - Have handlers to recognise and validify instructions, autogenerated from the XML File through XSLT Transformation. Assembler - Reads information live from XML File into Object Tables for use during assembly. Processor - Handlers for executing instructions autogenerated from XML file.

   This extension required use of XML, DOM, XSLT and ANT technologies. Setting up these different technologies to work within the YAMS framework while still making it easy to add instruction proved to be a significant point of achievement within the project timeline.

5.4.2 Extensions Not Achieved

1. Full MIPS R3000 Support

   It was not possible to implement the full MIPS instruction set for all areas of YAMS. NOTE: The handlers for all instructions within the processor were in fact completed, however the Assembler / Parser data was not

available by the end of the project timeline for all of these. Therefore these are ommitted from the final project build.

It should be noted that the flexibility provided by using the XML file means that these instructions could quite esily be added at some point in the future.

2. Directives Hard-Coded

   In addition to not supporting all the instructions, the Assembler Directives e.g. .asciiz were in fact hardcoded into the Assembler. It was envisaged that at some point these could be transferred to the XML File as code. This would have meant further XSLT transformations and extended the "ant" build, and this could not be achieved in the time alotted.

   This would require more modifications to the code of the Assembler to able to implement this extension in the future.

3. Help Descriptions

   It was also planned that help entries for each of the instructions could be added to the XML Instruction Repository along with the other instruction information. This was partially implemented, and it can be seen in the XML that the tags for the help section are in fact present. However, further XSLT transformations would be required to generate HTML Help Files from this data. Other "help" is available within YAMS, and it was planned to extend this help feature with more information regarding every instruction supported. This was not achieved, but once again could be a future modification.

4. Graphical Memory Statistics

   This extension was partially completed. It was not possible to implement graphical representations of the exact memory usage for each MIPS file executed. However, the StatisticsManager does indeed keep track of the total amount of memory used during a run and display it numerically. It would require some modification to the GUI to enable this feature, but would not be an extensive change since it would mean simply adding another graph to the "Graphs" pane.

5.5 Project Evaluation

5.5.1 Group Organisation

Initially, the project was split into three sections and two people were allocated to each section. This meant that if one person became ill or had to spend time away, the project could continue.

We organised regular meetings, many informal, to check on progress and resolve any conflicts between sections. This resulted in a successful integration of the components on schedule.

We also maintained close contact with our supervisor through regular meetings and emails to keep her informed of our progress.

5.5.2 Final Product

The final version of YAMS consists of two versions, a console version and a GUI version. Both have their uses within the scope of the compiler lab, however a significant aspect of YAMS is its extensibility.

Through the modification of a single XML Instruction Repository, the functionality of YAMS can be extended. This has potential to support multiple instruction sets (within the bounds of the MIPS processor).

The final product provided is therefore more than just a tool for the compiler lab. It could be tailored for multiple purposes, using more or less complex instruction sets.

While there are flaws within both versions, the layered architecture (GUI and Console layer above the underlying simulation core) allows continuing updates and improvements to the GUI and user-friendly features.

Overall, this has been a complex yet rewarding project, resulting in a versatile product.