AUSTRALIAN NATIONAL UNIVERSITY

COMP4560 – ADVANCED COMPUTING PROJECT
Semester 1, 2015

Enhancing and Extending the rPeANUt Illustrative Computer Simulator

Author: Yixiang Ren
Supervisor: Assoc Professor Peter Strazdins
U5061890
Abstract

rPeANUt illustrative computer simulator is a pseudo-assembly simulator currently used in the Introduction to Computer System course in the Australian National University for teaching the fundamental underlying principles of microprocessors. The aim of this project is to implement some enhancements and extends for rPeANUt to benefit the teaching and studying of this course in the future.
Contents

1 Introduction .............................................................................................................. 5
   1.1 Background ........................................................................................................... 5
   1.2 Contribution .......................................................................................................... 6
   1.3 Tasks ....................................................................................................................... 6
   1.4 Report Structure ................................................................................................. 6

2 rPeANUt ..................................................................................................................... 7
   2.1 Overview ............................................................................................................... 7
   2.2 Instruction Set ....................................................................................................... 9
   2.3 Hardware and Interrupt ....................................................................................... 11
   2.4 Assembler ............................................................................................................. 12

3 Design ....................................................................................................................... 13
   3.1 Architecture ......................................................................................................... 13
   3.2 Language Specifics ............................................................................................... 13
   3.3 Project License ..................................................................................................... 14
   3.4 Code ....................................................................................................................... 15

4 Implementation ........................................................................................................ 15
   4.1 #include .............................................................................................................. 16
   4.2 #define ............................................................................................................... 18
   4.3 Decimal Display ................................................................................................... 20
   4.4 Decoding of Assembled Instructions .................................................................... 26
   4.5 Multi-core Mode .................................................................................................. 28
5 Conclusion and Future Directions .......................................................... 32
6 References ........................................................................................................... 32
7 Appendix ............................................................................................................ 33
   7.1 Project Contract ............................................................................................. 33
   7.2 ReadMe .......................................................................................................... 35

List of Figures

Figure 1: rPeANUt Architecture (Eric McCreath, 2013) ......................... 9
Figure 2: rPeANUt Instruction Set 1 (Eric McCreath, 2013) ............... 10
Figure 3: rPeANUt Instruction Set 2 (Eric McCreath, 2013) ............... 11
Figure 4: rPeANUt Interrupts (Eric McCreath, 2013) .............................. 12
Figure 5: Class Diagram of rPeANUt ............................................................. 16
Figure 6: Detail Class Diagram of Assemble ............................................. 18
Figure 7: GUI of rPeANUt v2.4 ................................................................. 21
Figure 8: Memory Cell Indexes in Decimal ................................................. 22
Figure 9: Detail Class Diagram of Simulate (attributes) ............... 24
Figure 10: Detail Class Diagram of Simulate (operations) .................... 25
Figure 11: Example of Complete Decoding .............................................. 27
Figure 12: Detail Class Diagram of Memory ............................................. 28
Figure 13: rPeANUt GUI in multi-core mode ......................................... 30
1 Introduction

1.1. Background

An assembly language is a low-level programming language for a programmable device (Wikipedia), which is usually working on a particular computer architecture rather than multiple architectures like a high-level language. It is one of the most important things when learning programming, because it helps to understand how high-level languages work and what is going on underneath. This contributes to understand the efficiency of codes, which is benefit to improve programming skills with high-level language.

COMP2300, the Introduction to Computer Systems, refers to these contents. In this course, a program called PeANUUt was created several years ago. PeANUUt is a simple microprocessor simulator with an extensive instruction set. Then, Eric McCreath wrote a new software called rPeANUUt in 2011. rPeANUUt is a RISC (Reduced Instruction-Set Computer) version of PeANUUt, which means the architecture is load-to-register, perform-operation and store-to memory. rPeANUUt had a much simpler design that makes teaching and learning more efficient.

rPeANUUt is still not perfect. Compared with many usual programming languages, rPeANUUt is lack of many facilities that can make it user-friendlier. Besides, as an illustrative computer simulator, rPeANUUt only has basic virtual devices, which means more virtual devices can be implemented. The project is aim to do enhancements and extensions for rPeANUUt in these two fields.
1.2. Contribution

This project contributes to the teaching program of the Introduction for Computer Systems course by providing enhancements and extensions for rPeANUt. Facilities that are similar with usual programming language will make coding on rPeANUt easier. And then, more virtual devices will help students build a deep understanding on real-life computers.

1.3. Tasks

In this project, the tasks that have been finished are as below:

- Implement a #include facility
- Implement a #define facility
- Enable the display of a memory cell in decimal
- Display a complete decoding of assembled instructions
- Enable multiple cores

1.4. Report Structure

The rest of this report is structured as follows. First, there is
an introduction of the original version of rPeANUt so that the architecture and behaviours of it can be understood. Then the design of the project will be listed to provide an overview. After that, the implementation will be described in detail. Finally, conclusions and future directions will be provided.

2 rPeANUt

This chapter is based on the original version (v2.4) of rPeANUt rather than the product of this project (v2.4e).

This chapter is based on Specification of the rPeANUt Computer and Assembler (v2.1) (Eric McCreath, 2013).

2.1 Overview

The rPeANUt is a 32-bit microprocessor that contains 16-bit addresses, 32-bit registers, and memory addressed in words of 32 bits.

In this condition, the maximum amount of addressable memory is $2^{16} = 65536$ words or 262144 bytes (256k), while only the addresses 0x0000 to 0x7FFF are connected to actual memory. The other addresses (0x8000 to 0xFFFF) are reserved for memory mapped IO. Addresses 0x0000 to 0x00FF work for the interrupt vector and other OS code, and the last 960 words for actual memory is used for the frame buffer. There is an IP (instruction pointer) pointing to the address where the
instruction in the memory will be executed, which points to 0x0100 initially.

There are 13 32-bit registers as below:

- **R0, ...R7**: 8 generally purpose registers, which may be used for storing either data or addresses.
- **IR**: An instruction register, which stores the current instructions to be executed.
- **SR**: A status register, which stores the status information about the CPU. The status information can be OF (integer overflow), IM (interrupt mask) or TI (enabling the interrupt timer).
- **ONE, ZERO and MONE**: 3 constant registers, which contain the constants 1, 0 and -1.

There are 2 16-bit registers as below:

- **SP**: A stack pointer, which is used for method calls, method returns and interrupts.
- **PC**: A program counter, which is used to contain the address of the next instruction to execute.

The execution cycle of the micro processor is as follows:

```plaintext
do {
    IR = mem[PC];
    PC = PC + 1;
    execute_instruction in IR;
    check for interrupts;
} while(!halt);
```

The architecture of rPeANUt microprocessor is as figure 1.
2.2 Instruction Set
Instructions are all 32 bits (one word long). Registers R0, …R7, SP, SR, PC, ONE, ZERO, MONE can be accessed via instructions and take 4 bits in the machine code, which are encoded as 0x0, …0xD. Addressed and values take 16 bits in the machine code. The followed figures describes the instruction set of rPeANUUt.

<table>
<thead>
<tr>
<th>Name</th>
<th>Assembly Instruction</th>
<th>Machine code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>add &lt;RS1&gt; &lt;RS2&gt; &lt;RD&gt;</td>
<td>0x1&lt;RS1&gt;&lt;RS2&gt;&lt;RD&gt;&lt;0000</td>
<td>RD ← RS1 + RS2</td>
</tr>
<tr>
<td>subtraction</td>
<td>sub &lt;RS1&gt; &lt;RS2&gt; &lt;RD&gt;</td>
<td>0x2&lt;RS1&gt;&lt;RS2&gt;&lt;RD&gt;&lt;0000</td>
<td>RD ← RS1 - RS2</td>
</tr>
<tr>
<td>multiply</td>
<td>mul &lt;RS1&gt; &lt;RS2&gt; &lt;RD&gt;</td>
<td>0x3&lt;RS1&gt;&lt;RS2&gt;&lt;RD&gt;&lt;0000</td>
<td>RD ← RS1 * RS2</td>
</tr>
<tr>
<td>divide</td>
<td>div &lt;RS1&gt; &lt;RS2&gt; &lt;RD&gt;</td>
<td>0x4&lt;RS1&gt;&lt;RS2&gt;&lt;RD&gt;&lt;0000</td>
<td>RD ← RS1 / RS2</td>
</tr>
<tr>
<td>modulo</td>
<td>mod &lt;RS1&gt; &lt;RS2&gt; &lt;RD&gt;</td>
<td>0x5&lt;RS1&gt;&lt;RS2&gt;&lt;RD&gt;&lt;0000</td>
<td>RD ← RS1 % RS2</td>
</tr>
<tr>
<td>bit and</td>
<td>and &lt;RS1&gt; &lt;RS2&gt; &lt;RD&gt;</td>
<td>0x6&lt;RS1&gt;&lt;RS2&gt;&lt;RD&gt;&lt;0000</td>
<td>RD ← RS1 &amp; RS2</td>
</tr>
<tr>
<td>bit or</td>
<td>or &lt;RS1&gt; &lt;RS2&gt; &lt;RD&gt;</td>
<td>0x7&lt;RS1&gt;&lt;RS2&gt;&lt;RD&gt;&lt;0000</td>
<td>RD ← RS1</td>
</tr>
<tr>
<td>bit xor</td>
<td>xor &lt;RS1&gt; &lt;RS2&gt; &lt;RD&gt;</td>
<td>0x8&lt;RS1&gt;&lt;RS2&gt;&lt;RD&gt;&lt;0000</td>
<td>RD ← RS1 ^ RS2</td>
</tr>
<tr>
<td>negate</td>
<td>neg &lt;RS&gt; &lt;RD&gt;</td>
<td>0xA&lt;RS&gt;&lt;RD&gt;&lt;0000</td>
<td>RD ← ~ RS</td>
</tr>
<tr>
<td>bit not</td>
<td>not &lt;RS&gt; &lt;RD&gt;</td>
<td>0xA&lt;RS&gt;&lt;RD&gt;&lt;0000</td>
<td>RD ← ~ RS</td>
</tr>
<tr>
<td>copy register</td>
<td>move &lt;RS&gt; &lt;RD&gt;</td>
<td>0xA2&lt;RS&gt;&lt;RD&gt;&lt;0000</td>
<td>RD ← RS</td>
</tr>
<tr>
<td>call immediate</td>
<td>call &lt;address&gt;</td>
<td>0xA300&lt;address&gt;</td>
<td>SP ← SP +1 mem[SP] ← PC PC ← address</td>
</tr>
<tr>
<td>return from call</td>
<td>return</td>
<td>0xA3010000</td>
<td>PC ← mem[SP] SP ← SP-1</td>
</tr>
<tr>
<td>trap</td>
<td>trap</td>
<td>0xA3020000</td>
<td>SP ← SP +1 mem[SP] ← PC PC ← 0x0002 SR ← SR</td>
</tr>
<tr>
<td>jump</td>
<td>jump &lt;address&gt;</td>
<td>0xA400&lt;address&gt;</td>
<td>PC ← address</td>
</tr>
<tr>
<td>jump if zero</td>
<td>jumpz &lt;R&gt; &lt;address&gt;</td>
<td>0xA41&lt;R&gt;&lt;address&gt;</td>
<td>if (R == 0x00000000) { PC ← address }</td>
</tr>
</tbody>
</table>

Figure 2: rPeANUUt Instruction Set 1 (Eric McCreath, 2013)
Figure 3: rPeANUt Instruction Set 2 (Eric McCreath, 2013)

### 2.3 Hardware and Interrupt

A simple terminal is implemented via memory mapped IO. Three addresses are used to interact with this device: dataIO at 0xFFF0, status at 0xFFF1 and control at 0xFFF2. Bit 0 (OF) of
SR shows whether data is available (1 for available), and bit 1 (IM) shows whether it is ready to receive data (0 for ready). To write to the memory address of the dataIO location plays the role of writing to this device, and to read from the dataIO address means to read from the device. The control address is used to set interrupts on for the device (0 in bit 0 to disable interrupts and 1 to enable interrupts, which is 0 by default).

Bit 3 (TI) of SR enables a timer interrupt (1 enabled).

rPeANUt has a black and white screen, whose contents are determined by a frame buffer that starts at 0x7C40 and ends at 0x7FFF.

When an interrupt occurs, the current PC is pushed onto the stack and the PC is set to the address associated with the interrupt. The ‘return’ instruction is called to return from interrupts. The followed table is a list of interrupts:

<table>
<thead>
<tr>
<th>interrupt</th>
<th>address</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory fault</td>
<td>0x0000</td>
<td>This happens when memory is accessed that is not addressable.</td>
</tr>
<tr>
<td>IO device</td>
<td>0x0001</td>
<td>This happens when interrupts are enabled on the terminal device and a key is hit.</td>
</tr>
<tr>
<td>trap</td>
<td>0x0002</td>
<td>This interrupt happens when the trap instruction is executed.</td>
</tr>
<tr>
<td>timer</td>
<td>0x0003</td>
<td>When the timer is enabled and every 1000 clock cycles.</td>
</tr>
</tbody>
</table>

Figure 4: rPeANUt Interrupts (Eric McCreath, 2013)

2.4 Assembler

rPeANUt provides a simple way to assemble assembly code to rPeANUt machine code. There are two phases:

i. Line by line translation into machine code where a symbol table and a list of addresses that need resolving
are created.

ii. Resolving all these missing addresses.

The assembler will write the code directly into the memory.

3 Design

3.1 Architecture

The instruction architecture in this project is pseudo assembly language, which is the same with the original rPeANUt. The reason for this is that it will not lead to an extreme chance on the current teaching materials, as one of the aims of the project is to make teaching easier. Besides, it is more readable for students who are not familiar with assembly language.

3.2 Language Specifics

Java is used as the language for the development of this project to keep the coherence of the software. And also it is cross-platform enough, which means that students can run rPeANUt on Windows/Mac/Linux etc. as long as they have JRE (Java Runtime Environment) installed.
Swing is used in rPeANUt to develop the GUI. So in this project, all enhancements and extends on the GUI is using Swing toolkit.

3.3 Project License

The GNU General Public License Version 3 is used by rPeANUt. Developers can redistribute it and/or modify it under the terms of the license. Here is the full text of the license:

rPeanut - is a simple simulator of the rPeANUt computer.

Copyright (C) 2011-2012 Eric McCreath
Copyright (C) 2012 Tim Sergeant
Copyright (C) 2012 Joshua Worth
Copyright (C) 2015 Yixiang Ren

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.


3.4 Code

All of the code for this project is designed with flexibility and maintainability in mind, which is written in the same fashion as the original version to avoid misunderstanding.

4 Implementation

This chapter describes the implementation of each part of this project in detail. The figure below is a class diagram for all classes in rPeANUt.
4.1 #include

#include is a usual facility in high-level programming language, such as C/C++. It meets the need that call functions or other components from other files. In rPeANUit, an example is as follows:

addoperation.s:
addIn2: add R1 R2 R0

...

mainprogram.s:

#include addoperation.s

...

call addIn2

...

The principle of #include in rPeANUt is simple, which is the same with C. It is to copy the content of included file and insert it into the current file. When reading the #include instruction, the file name is stored in Attribute a1. FileInputStream fis is used here to read the content of the included file. The content is read to a byte array fdata. Then a String ftext is created to store the data in fdata. The current instruction is pushed on to the stack, and the assembler starts to assemble the instructions in ftext. When it is finished, the instruction in the stack is popped and the next instruction will be ready to be assembled.

There has already been a include facility in rPeANUt2.4. However, it only works for the files in the current directory. Considering popular programming languages like C, my supervisor believes that it is important to use preinstall functions located in the install directory of the language. To meet it, an environment variable RP_INC_PATH is involved. After importing this environment variable into their operation system, users can include files under RP_INC_PATH/ now.

In the implementation of the using of RP_INC_PATH, the directory in this environment variable is read into a String incpath via System.getenv(). If incpath is not null, it will be
translated and connected with the file name to form the directory for fis to use.

The figure below is the detail class diagram for the class assemble. The implementation of #include is in this class.

![Figure 6: Detail Class Diagram of Assemble](image)

### 4.2 #define

#define is another important facility in popular programming languages. In C, it is used to replace constant with meaningful string so that the code will become more logical and readable. In rPeANUt, it is expected that users can replace number or register name with any symble they want. It is also known as short macro in rPeANUt. The aim of it is to increase the readability of codes. A simple example is as follows:

```c
#define _FRAMEBUF 0x7C40    ; // rPeANUt address of start of frame buffer

#define WORDSZ 32           ; // size of a word in rPeANUt

#define xOffs -1            ; // stack offsets for func()
```
```c
#define x R1                ; // registers holding vals of
parameters & locals
#define i R0                ;
#define y R2                ;
#define z R3                ;

func:                       ; void func(int x) {
  load    SP #xOffs x ;
  load    #WORDSZ i   ;   int i = WORDSZ; //
same as load #32 R0
  move    x y         ;   int y = x; // same as
move R1 R2
  load    x #_FRAMEBUF z; int z =
  _FRAMEBUF[x];
...
```

#define is a new facility in rPeANUt, which is different from
#include. The data structure HashMap Define is used to store the
identifier and the token-string (the two parts after the word
define) and create relationship between them. The
implementation of #define is divided into two different parts:

The first part happens when the #define instruction is read
into the assembler, which is in the same level with #include. In
this part, when dealing with #define instructions, the two
components in the instruction are store in Attribute a1 and a2.
Then Define.put() is used to put them into the HashMap and
create connection between them. However, considering that in
rPeANUt assembly language there is a # symbol before values, I
also use Define.put() to create connection between ‘#’+a1.str
and ‘#’+a2.str to avoid potential bugs.
The second part happens between the assembler reading and starting to deal with instructions. Each component in an instruction is read into a String \( \text{ins} \) in turn, and then \( \text{Define.containsKey} \) is used to check whether it is in the HashMap. If so, \( \text{Define.get} \) is called and the result will replace the original part in the instruction. Otherwise, the component will be kept in the instruction. After this process, a new instruction will be formed for the assembler to translate.

The implementation of \#define is also in the class assemble, which is referred in figure 6.

4.3 Decimal Display

The GUI of rPeANUt is not satisfied enough for teaching and learning. The followed figure is the GUI of rPeANUtv2.4:
In figure 7, it is easy to notice that the addresses and data displayed in the table in the GUI are all in hexadecimal. For human beings, decimal is much more sensible than hexadecimal. The task here is to show them in both decimal and hexadecimal so that students can have a good understanding of the content of the memory.

There are three ideas to meet this need:

The first one is to create two more columns to display the decimal number of addresses and data. The advantage is that students can see the decimal number and hexadecimal number of all indexes at the same time. However, it makes the table really crowded. Besides, at most conditions, the display of all indexed is not necessary. So this idea is abandoned.

The second idea is to add a button to control the translation between decimal and hexadecimal. It makes the table tidy.
enough, but users cannot see the decimal number and hexadecimal number at the same time. So there is the third idea.

The method accepted in this project is to use JPopupMenu to display the decimal indexes. When users right click on a line in the table, the address and data in this line will be listed in decimal in the menu appeared, which is shown in figure 8.

![Figure 8: Memory Cell Indexes in Decimal](image)

In the implementation, a JPopupMenu `popup` is created, two `JMenuItem` `jMenuItem1` and `jMenuItem2` are also created and added to `popup`. And then a mouselistener is added to `memtable` (the JTable used to show the table in the GUI) to listen to the right-click event `evt`. When the event happened, the row number in the table is worked out by dividing `evt.getY()` (the Y position of the right-click event) by `memtable.getRowHeight()` (the height of one row in the JTable). Then the row number is used to find the value of address and data in the Memory `memory` (the
JTable actually storing all indexes). These two values are translated to decimal and stored in those two JMenuItems, which are listed in the popup menu then. Because the values from memory are in String, the translation should use Long.valueOf() to deal with them (the first two characters “0x” in the Strings should be removed in advance).

The implementation of decimal display is in the class Simulate. The detail class diagram of it is as follows (there are two figures because of too many attributes and operations in Simulate):
Figure 9: Detail Class Diagram of Simulate (attributes)
Figure 10: Detail Class Diagram of Simulate (operations)
4.4 Decoding of Assembled Instructions

The decoding of assembled instructions means to translate the machine language to pseudo code for users to read. There has already been a decoding function in rPeANUt2.4. In figure 7, the fourth column, which is load b+d etc., is the decoding of the machine language shown in the data column. However, the decoding here is not detailed enough, as it only contains the type of instructions rather than the parameters. A more detailed decoding is required for students to build a better understanding on the machine language.

Three types of pseudo code are considered during the design process:

The first one is to keep the current things and add the parameters after them. The work of it is the least, but because the format is not logical enough, students may feel confused when dealing with it.

The second one is as the description of rPeANUt instruction set (figure 2 & 3). The benefit is that it is made up to number and operators so that it is clear enough. However, because there are not enough words describing the action of it, students who are not familiar with pseudo code may meet confusion.

The last one is to translate the machine language back to rPeANUt instructions. It is quite clear and logical for students, and it contains all information of instructions. This idea is used in this project at last. An example is as follows:
The implementation of decoding is to modify the current decoding. The result of decoding is stored in `memory[].dump` in Memory class. To decoding is to deal with `memory[].value` (shown as data in the GUI). All these `values` are in the format of 32-bit hexadecimal integer. The first thing of decoding is to judge the type of instructions. And then, according to the instruction type, different bits of `value` are used for decoding. The principle is that: the first 4/8/12/16 bits stores the type of instructions; the value and address are stored in the last 16 bits; register labels are stored between the fifth bit and the 16\(^{th}\) bit. The translation is based on the machine code column in figure 2. Because the values in rPeANUt assembly language are signed, all values larger than 32767 should be changed to minus.

The implementation of decoding is in the class Memory. The detail class diagram of this class is as below:
Figure 12: Detail Class Diagram of Memory

4.5 Multi-core Mode

The rPeANUt is a simulator for students to learn how
real-life microprocessor works, so some features and devices that real-life computer owns are reasonable to implement in rPeANUt. The most important feature should be multiple cores. In real life, machines use multiple cores to confirm efficiency and robustness. In rPeANUt, the multi-core mode is used to show how instructions run in different cores with a shared-memory.

In this project, because of the limitation of the size of GUI, the multi-core mode uses only two cores in fact. These two cores share the memory, while they have their own registers. Each instruction runs on the two cores in turn. The run of a program is halted when the run in both of the cores are halted. The step will make one core run one instruction at one time.

In the command line mode, multi-core mode can be entered via –multicore.

Compared with the GUI in figure 7, there are some modifications done for the multi-core mode:

The first part is the display of registers. There are two sets of registers and every register has C0/C1 to show its belonging. And there are also two counts for the two cores to count the number of instructions that have run.

The second part is the memory table. Because the two cores share memory, there is only one memory table here. The difference is that there are two different colours highlighting the instructions that each core is running. The instruction highlighted with blue is the instruction that core 0 is running, and the instruction highlighted with green is for core 1. When both of cores are running the same instruction, the colour of core 0 will be displayed.

The GUI in multi-core mode is given in figure 13.
The implementation of multi-core contains two major parts and several minor parts:

The first major part is to create registers for core 1 and modify operations in the class Simulate to fit multi-core mode. A variable numCore is involved to determine whether there are two cores or not. All registers and the int variable count are declared as array[2]. Operations used to get or set values to registers are all modified by including a parameter noc. noc refers to the number of the core that calls these operations.
Another two operations step() and update() are also modified by including *noc*. step() is the CPU, which reads the data from the memory and deals with them, so *noc* is used to separate the behavior of the two cores. update() is the painter(), which is used to repaint the registers and the count in the GUI when changes happen. To repaint the whole register area, update() should be called for the two cores separately.

The other major part is the modification of highlight. TableCellRenderer is used in this project to set the background colour for instructions. When reading a memory cell, the row number will be check. If the row number is the same with the instruction that core 0 is running, the background of the cell will be set as blue. And if it is the same with core 1, the background will be set as green. For other cells, the background will be kept as white. When using the CellRenderer, setCellRenderer() is called for each column of the JTable *memtable*.

A minor part is a operation checkhalt(). As explained, the halt of a program in multi-core mode is defined as both of cores halted. checkhalt() is used for this aim. It returns a boolean value to show whether both of cores are halted.

Other minor parts are the modification of the calls of step() and update() in other classes. *noc* is used in these cases.

These modifications on rPeANUt will not influence the performance with one core because all of them are controlled by the variable *numCore*.

The first major part and the first minor part are implemented in the class Simulate. The second major part are implemented in the class Memory. Simulate class diagram is in figure 9 and 10. Memory class diagram is in figure 12.
5 Conclusion and Future Directions

rPeANUt is developed to be used in the Introduction to Computer Systems course in the Australian National University. Enhancements and extends are designed and implemented in this project to provides more facilities and a clear and logical user interface. The #include and #define have been used in the teaching. And the others will help the teaching in the future.

However, there are still many things to do. More simulated virtual devices, such as MMU memory, hard disk and sound card, can be included in the software to make it more similar with real computer. And the codes of rPeANUt can be modified to be more efficient and logical by dividing classes into different packages and dividing large classes into several small ones. Further developments will be done via the ANU gitlab.

6 References

Wikipedia, Assembly language,

<http://en.wikipedia.org/wiki/Assembly_language>

Eric McCreath, 2013, Specification of the rPeANUt Computer and Assembler (v2.1),

http://cs.anu.edu.au/courses/COMP2300/rpeanut/rPeANUt_v2.1spec.pdf
7 Appendix

7.1 Project Contract

INDEPENDENT STUDY CONTRACT

Note: Enrolment is subject to approval by the projects co-ordinator

SECTION A (Students and Supervisors)

UnitID: _u5061890________
SURNAME: _Ren_ FIRST NAMES: _Yi xiang_  
PROJECT SUPERVISOR (may be external): _Peter Strazzis_  
COURSE SUPERVISOR (a RISC academic): ___________  
COURSE CODE, TITLE AND UNIT: ___COMP4560 Advanced Computing Project 12 Units______

SEMIESTER ☐ S1 ☐ S2 YEAR: ____________

PROJECT TITLE:
Enhancing and Extending the rPeANUt Illustrative Computer Simulator

LEARNING OBJECTIVES:
-Deeply understand computer organisation
-Undertake software engineering practice

PROJECT DESCRIPTION:  
rPeANUt is a RISC version for the long-established PeANUt illustrative computer. It is used for teaching in ANU’s Introduction to Computer Systems course. Its software was principally developed by Eric McCreath over 2011-2014.  
The software requires enhancement and extension for intended use from 2015 and beyond. This project will perform these and also improve the software for longer term maintenance (reverse engineer a design, unit testing etc).  
The intended enhancements and extensions include:
1. implement a #include facility
2. enable the display of a memory cell in decimal
3. display a complete decoding of assembled instructions
4. enable multiple cores
5. implement a Virtual Memory mode with an MMU
6. implement simulated hard disk that backs onto a file
7. implement a simulated sound card

Research School of Computer Science
Jun-12
ASSESSMENT (as per course's project rules web page, with the differences noted below):

<table>
<thead>
<tr>
<th>Assessed project components:</th>
<th>% of mark</th>
<th>Due date</th>
<th>Evaluated by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report: name style: _<strong>software description</strong>, (e.g. research report, software description...)</td>
<td>60%</td>
<td>As per normal projects</td>
<td>Articled academic</td>
</tr>
<tr>
<td>Artefact: name kind: ___codes and test __________ (e.g. software, user interface, robot...)</td>
<td>30%</td>
<td>As per normal projects</td>
<td>Supervisor</td>
</tr>
<tr>
<td>Presentation:</td>
<td>10%</td>
<td>As per normal projects</td>
<td>Project coordinator</td>
</tr>
</tbody>
</table>

MEETING DATES (IF KNOWN):
Weekly, Wednesday 2pm.

STUDENT DECLARATION: I agree to fulfill the above defined contract:

Yi-Xiao Ren
Signature
11/03/15
Date

SECTION B (Supervisor):
I am willing to supervise and support this project. I have checked the student's academic record and believe this student can complete the project.

Signature
11/03/15
Date

REQUIRED DEPARTMENT RESOURCES:

SECTION C (Course coordinator approval)

Signature
Date

SECTION D (Projects coordinator approval)

Signature
Date

Research School of Computer Science
Form updated
7.2 ReadMe

//Software: rPeANUt
//Version: 2.4e
//Student name: Yixiang Ren
//Student ID: u5061890

The tar file contains:

rPeANUt (the source code)
rPeANUt.jar (runnable jar file)
ReadMe

The software can be run by:

GUI:

java -jar rPeANUt.jar (-dump) (-count) (-check) (-screen)
(-objdump) (-load) (-help) (-multicore)

Command Line:

java -jar rPeANUt.jar (-dump) (-count) (-check) (-screen)
(-objdump) (-load) (-help) (-multicore) filename

List of files modified:
Assemble.java
Memory.java
Peanut.java
Simulate.java
SimWorker.java

In Assemble.java:

new hashmap
create new instruction based on #define
main part of #include and #define

In Memory.java:

functions are modified to provide different background of the instructions the two cores are running
a renderer is created for background colour
a detail description is decoding from the data in each address

In Peanut.java:

new parameter multic for Peanut() and Simulate()
-multicore added for multicore mode in terminal
the calls to simulate.update() are modified
In Simulate.java:

`Jpopupmenu` is created for display decimal numbers of address and data

in multicore mode, registers for core2 are created

functions are modified to work for both cores

the calls of these functions are modified

components are resized for multicore mode

In SimWorker.java:

the calls to `simulate.step()` and `simulate.update()` are modified