Contents

Introduction

The Study of Microprocessors

As part of their second year, students studying computer science at COGS study a course called *Computer Systems Architecture*. As part of this course, they learn how to write in assembly language. Writing in assembly shows the student that:

1. High level languages and compilers are invaluable.
2. Processors have a fixed, small number of instructions they can be called to execute.
3. Data can be held in special registers on a processor. Different processors have different registers. Sometimes registers are assigned to specific tasks. Data not held in registers must be held in main memory.

Students wrote their programs in the assembly language of the MIPS R2000 processor. The R2000 is not the newest chip, but it is the epitome of the RISC principle[1]. Modern processors are too complex for anyone other than a specialist to study.

The R2000 and its descendants were powerful for their time, and were used in high end workstations. Nowadays the family is still used, albeit in less glamorous components such as unix terminals, games consoles and embedded devices.

1 A RISC chip is one with a very small set of instructions. This is in contrast to CISC chips, which have very many instructions, to aid the assembly programmer. The RISC principle assumes that software authors use a high level language and a compiler, and do not need lots of instructions. RISC is generally considered the better design, and high end computers almost exclusively use RISC chips (Sun, Silicon Graphics, etc). However, RISC chips never made it on the desktop market, dominated by the Intel 80x86.

The SPIM Simulator

Rather than being run on an actual R2000 chip, the programs were run on a simulator called SPIM. Running the program on a simulator was of great benefit. The program could be stepped though, in a similar manner to a C debugger. The values held in registers and memory were shown on the user's screen.

To have this sort of functionality on a real chip is not impossible, but requires expensive specialist hardware. In contrast, all the students on the course could run the simulator at the same time on the university's workgroup grade computer (Sun 450). One hardware setup per student would cost many times the amount that the COGS computer did. Students could also run a Windows version of the simulator at home, on their PCs.

SPIM was written in C, which I imagine would have been the natural choice at the time. C is fast, small, general purpose, and is the native tongue of the UNIX community.

A graphical version of SPIM called XSPIM is available for the X system on unix. Unless specifically stated otherwise, when I refer to SPIM, I mean XSPIM.

Problems with SPIM

SPIM is not a bad piece of software, but it does has some problems:

1) It is hopelessly unstable. If an assembler program accessed memory it did not own (What C programmers call a "segmentation fault"), the *whole simulator* would crash, returning the user to the command line with no apparent reason. It was difficult to know whether it was a bug in the user's program or a bug in the actual simulation which caused the problem. I think the simulator should handle faults like these, and convey some useful information to the user.

2) It is not intuitive. XSPIM uses a very old graphical widget set. One cannot tell easily what is a button and what just text. "Tool Tips" would be invaluable. The console-based SPIM is even worse. Some programs work well on text only modes (pine, vi), but there is simply too much information for SPIM to convey effectively.

3) It is inconsistent. Programs which work on the Windows version of SPIM might not work on the unix version. Psuedoinstructions varied between the two versions. The GUI is different for the two versions too. Strangely, the behavior of the unix version actually depended on the hardware of the host computer[2]. That means the same program (compiled from the same source) running on a Sun / Solaris system would behave differently to on an Intel / Linux system.

2 Whether it was 'Big Endian' or 'Little Endian' (which way round bytes are arranged into words).

A Replacement Simulator

I propose writing a replacement simulator. I will make it better than SPIM by including the following features:

1) To be able to execute all the instructions of the MIPS R2000 processor, so as to be an accurate simulation.

2) To support all the system calls of the original SPIM, so that COGS lecturers need not learn a new system. The most interesting of these to implement will be dynamic memory allocation. This would require some sort of memory manager.

3) (implied) To run programs written originally for SPIM (excluding pseudoinstructions, which are an optional extra).

4) To have a user interface which is *at least* as good as XSPIM's, and hopefully much better. The quality of a user interface is not something which can be quantitatively assessed, so I shall set myself this goal: Somebody comfortable with MIPS assembler and computer architecture should be able to use the program without referring to any printed manuals, other than to check assembler / system call syntax.

5) To be stable. Errors in the students' assembly code *must not* cause the simulation to crash. Indeed, the simulation must not crash *at all*. Useful error messages should be shown, and the user left in no doubt as to what happened.

6) To run on at least as many computer platforms as SPIM (i.e. Unix and Windows). I don't think it is required that a replacement has to run on the unix console, because I am yet to find anyone who actually used this version. The GUI should look as similar as possible on different platforms.

7) To be able to simulate as many processors as possible.

The last goal is particularly interesting, and quite difficult. It is what will set this project apart from a regular program like SPIM though.

Simulating Any System

The ability to simulate any computer type is not an easy goal. For it to work, it must be made certain at all times during development what parts are generic to all computer systems, and what is specific to a certain architecture.

Communication between the two groups must be kept within a defined set of method calls.

The Concept of a Plug-In Architecture

Once the generic simulation is made, we will know what parts specific to an architecture are required. A definition of an architecture 'plugin' can be constructed. It is important to realize that if *all* architectures are plug in objects, then the simulation will not function on its own; it will *need* a plugin to run at all.

As this simulator is designed to replace the SPIM simulator, the first plugin to be written will be an implementation of the R2000 processor. If a second implementation can be written in time, it will show how the simulation is generic, and act as proof that the plugin architecture works.

The Following Sections

The design and programming effort is split up into three distinct parts

1. Jukebox: Analysis and Design

This section describes the evolution of the generic parts of the simulation; those which are present no matter the architecture being simulated. This includes superclasses from which implementations extend, as well as the user's control system.

It also includes utility classes, which implementations are free to make use of. The utility classes are for such functions as binary arithmetic and graphical display.

2. The SPIM System

The first plug in architecture to be simulated is that for the R2000 processor, and the other parts of the SPIM simulator (memory layout, etc.).

Of course, since the simulator will only run with a plugin, this particular plugin must be developed alongside the generic parts of the simulator. This will allow both parts to be tested, and one part will not have to wait for development of the other to finish.

3. The DM0 System

A second plugin proves the ability of the system to simulate any architecture.

After these sections, we draw conclusions, and give appendices.

JukeBox: Analysis & Design

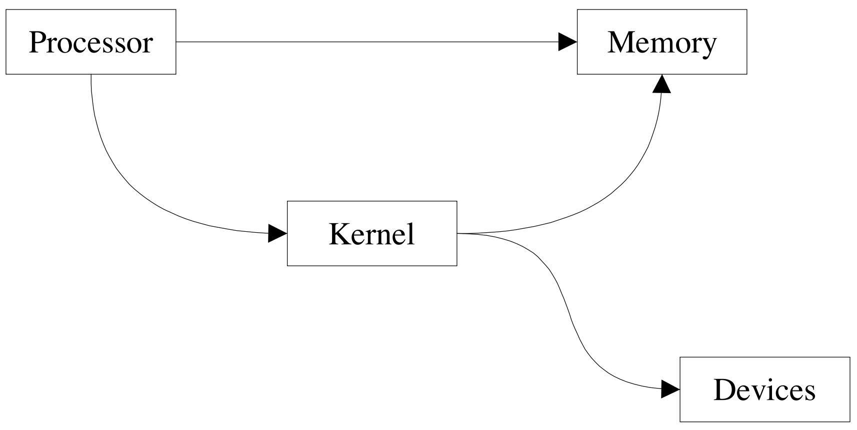
Contents

(Contents continued overleaf)

Design

The Structure of a Computer System

Any computer system we would want to simulate has a similar structure to the diagram below, shown from the point of view of a user's running program. The arrows point *from* a controlling component *to* a controlled component. We find that information actually tends to flow in the opposite direction to the arrows (e.g. from memory to the processor).



Structure of a Computer System

The Problem: Separating the General From the Specific

From the start,the goal of the simulation was that different processor implementations should be able to run. It is therefore important to separate details which are specific to any one implementation and details which are common to all systems.

That said, different implementations of the same conceptual component (e.g. two processors) should have some relation between them, so that they can be handled generically by the rest of the system.

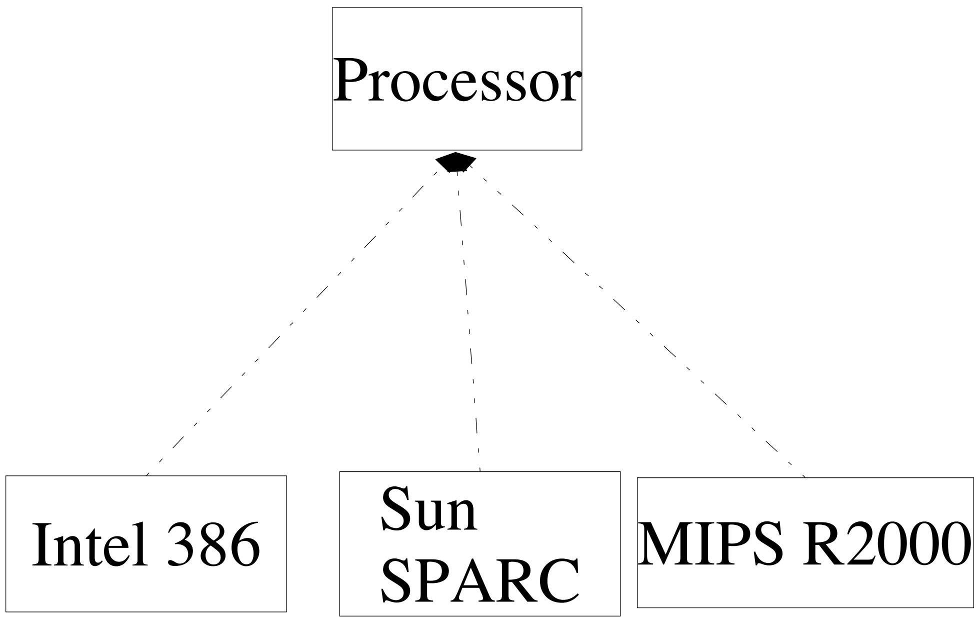
This chapter deals with the design of the general simulation. So, it does not consider any specific architecture. Such details are considered in the two plugin chapters later.

The Solution: Inheritance

This page should not be read lightly, it is important to realize how important the concept of inheritance is to this simulation.

Object oriented languages supply a good solution to this problem; inheritance. A generic superclass, such as **Processor** can be made, into which behavior common to all processors can be programmed. Then, for a specific *type* of processor, a class is made which *extends* the **Processor** class. This subclass can automatically do everything a generic **Processor** can. The implementation need only worry about what it specifically has to do.

A possible inheritance tree with several possible processor implementations is shown below:



Structure can be imposed to enforce that the generic superclass can never be instantiated directly. In Java, this is achieved with the keyword **abstract**.

The same system can be used for memory and kernel implementations.

The Following Sections

By designing the superclasses sensibly, we are in fact designing the structure of all implementations. The following sections describe the choices made in the design of the abstract superclasses.

The system components for which we design superclasses are:

1. Processor
2. Instructions
3. Memory

After these classes, which form parts of a real computer system, we shall discuss the design of some other

Superclass 1: Processor

The processor is the heart of a computer.

Its design determines the power of a computer system..

Requirements

There is only one thing we might want the system processor to do, and that is to execute a single step. A **void** method called **tick** would seem appropriate (named so because it is a pulse of the system clock that causes a real life processor to carry out a step).

For simple processors (in real life), a single clock pulse causes an instruction to execute. Newer processors (especially CISC ones) might take several clock pulses to execute a single instruction. We shall leave it up to each implementation to decide whether a call to **tick** should cause a whole execution of an instruction, or just a single clock pulse.

Since any number of exceptional situations might occur during the execution of a single call to **tick**, we declare that it throws the general **RuntimeSimException**[1].

Design

We can enforce the **tick** method, by declaring (but not defining) it in the abstract class. The method is declared **abstract** too. This means it *must* be overwritten by any superclasses. The compiler enforces this rule.

All computer systems have memory and a system kernel, which the processor needs access to (see diagram earlier in this paper). We can supply **protected** fields in our superclass. This promotes readability, as different processor implementations will have the same handle name as each other for their memory and kernel.

Later, we shall see that the memory is actually two objects (called **memory** and **program** below)

Here is the abstract class. This forms the basis of the design of any processor implementations.

```
public abstract class Processor[2] {


    protected InstructionMemory program;
    protected DataMemory memory;
    protected Kernel kernel;

    public abstract void tick() throws RuntimeSimException;


}
```

1 See "Exceptions" section for discussion of the **RuntimeSimException** class.

2 After reading the 'User Interface' section, we shall see that this class extends **VisibleSimulatorComponent**.

RegisterSet Class

`(uk.ac.sussex.cogs.peterpi.simulator.processor.RegisterSet)`

Practically every processor has a set of registers. Registers are identified by a name or number. Since a store of register values is something every processor implementation will need, it makes sense to supply one in the **processor** package. This is what the **RegisterSet** class is for.

A processor need only maintain one handle to a **RegisterSet** object, which stores the registers itself in a hashtable.

Each register is an instance of the **Value32** class[3]; the graphical equivalent of an int. The **RegisterSet** itself is a subclass of **JPanel**, and can be added to the processor as a graphical component (later we shall see that the processor is a **JPanel** too).

The register set must be able to return a handle to any one of the stored registers. The registers may be identified by their index (e.g. register number 8), or by name (e.g. "$t0"). To allow this to happen in a sensible way, the set will keep two hashtables:

```
private Hashtable registers;
private Hashtable indecies;
```

The main table 'registers' will hold the actual registers, and is keyed by that register's index (an **Integer** object). A secondary table 'indecies' will hold each register's index[4], and is keyed by the register's name. So, to retrieve a register from its name requires two hashtable requests; one from name to index, and one from index to register.

This is not as inefficient as it sounds: The **Integer** class overrides the **hashCode** method of **Object**, to return a hashcode that *is* that **Integer**'s value. This means that the 'indecies' table (indexed by **Integer**s) works practically as fast as an array. I could have (and indeed attempted to) used a **java.util.Vector**, but I have had previous trouble with that class. It cannot be trusted to execute its **ensureCapacity** method correctly. It gave unpredictable run time errors on my system, and I believe it may have a bug. With more time, I would investigate this further, and may well have ended up registering a bug with Sun.

3 See User Interface section for a description of this class. For now, read 'Integer'.

4 Question:	Why doesn't the 'indecies' table actually hold the registers, so as to reduce access time?
Answer:	Because sometimes it is useful to get the index of a register from its name.

Extending RegisterSet

The **RegisterSet** class is not abstract, but it is really *meant* to be extended. Classes extending **RegisterSet** would typically add visual hints, such as titled borders around related groups of registers (e.g. the temporary value registers on an R2000). If a processor implementation did not find it necessary to display its register set graphically, then it could use the base class.

The public interface to a **RegisterSet** is as follows:

```
public int lookup (String name) // gets the index of a named register
public Value32 fetchReg (int index) // gets the register at the specified index
public Value32 fetchReg (String name) // identical to: fetchReg (lookup (name))
```

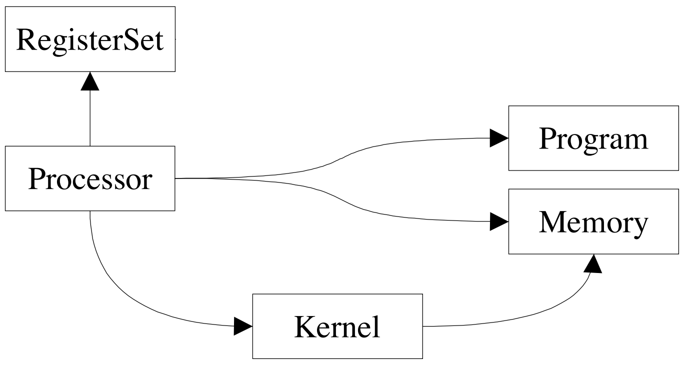
Subclasses also have access to the protected fields which add new registers to the hashtables:

```
protected Value32[5] makeRegister (String [] names, int index)
```

See how the names parameter is an array. There is nothing stopping one register being accessed by a number of different names. This is not uncommon in actual hardware.

Diagram

Our idea of the processor is now at this stage. The arrows show which object has a handle to another.



The Processor Model

Note:

The object oriented programming model supplies no distinction between what an object *has* and what it can *use*. Here, the processor *has* a register set, and can *use* memory. In code, both are treated equally as instance variables. If the reader prefers to visualize the register set *inside* the processor, this is fine.

5 Returns a handle to the register that has just been added to the hashtables. This is required if the subclass is going to add it graphically (e.g. add (makeRegister ("x", 2));)

Superclass 2: Instructions

The set of instructions a processor obeys is what defines it,
and how we go about programming it.

The Real World

In a real computer, instructions are encoded as binary words, just like data. This was a fundamental decision in the design of the first computer systems.

Instructions can be held in memory just as data can. Indeed, the memory itself is unaware of the difference.

During the course of execution, the processor reads in a word from memory, which is the next instruction to execute. It then examines the bits of the word to discover what actions it should perform. Finally it carries out those actions. This is known as the *fetch, decode, execute* cycle.

The Simulated World

In my simulation, I will not encode instructions as binary words, but as objects. This is to stop parts of the program from becoming inconsistent.

For example, when the processor executes an instruction, it performs a mapping from a *number* to a *concept*, and then performs an *action*. A completely separate part of the program, the assembler, performs a translation from a *string* to a *concept*, to a *number* to store in memory. An error in one of these components (but not the other) will create an inconsistency.

This might mean that the string symbolizing *addition* in the user's code actually ends up being carried out as *multiplication* on the chip. This will be unbelievably difficult to debug. Also, it will mean that chip definitions are likely to contain huge **switch** statements[6].

Instructions as Objects

Instead, instructions will be encoded as class definitions, extending a general **Instruction** superclass. These implementation classes will have the ability to encode *themselves* as strings and numbers, to give the user the impression that all those complicated transformations are actually being carried out.

Also, using **Instruction** objects as *visitors* to the processor can simplify a processor implementation, something discussed in a few page's time.

6 There is nothing especially *bad* about switch statements, but they do serve as a visual clue that a polymorphic solution might be more elegant.

A Naming Convention

On the previous page, we saw how instruction objects take care of their own mapping from concept to number, and to string. Once an **Instruction** object comes into being, it can masquerade as either one of these formats. One problem remains though. How can a mapping be made for the assembler; from the *string* of source code, to the *class* definition for that instruction?

We can solve the problem with a Java feature normally used for other purposes; the **Class** Class[7]. Objects of type **Class** represent a class definition. When we have an object of type **Class**, we can instantiate another object *whose type is that* represented by our original **Class** object.

For example, suppose object 'A' was of type **Class**, and its instance variables were such that it represented the **java.util.Vector** class definition. By calling a method of A, we can instantiate a new Vector *without* having to use the **new** keyword. Also, we do not need to know the *exact* type of the new object. It would be enough to treat it as any one of its superclass types (eg 'Object').

The only way to create a **Class** object is to call the static **Class.forName** method, passing in the full class name (e.g. `"java.util.Vector"`). So, at this stage, we know that if we have the full name of a class, we can instantiate objects of that class. We must however have a type (possibly a superclass) for the handle we will place on that new object.

Here's the trick; for each subclass of **Instruction** (each instruction of our processor), we can name the class *exactly* how the instruction is written in source code.

For example, the MIPS instruction for addition is written "ADD" in assembler. So, the class representing this instruction is also called "ADD". When the assembler comes across the string "ADD", it can automatically create an **ADD** object. Because all instructions inherit from the abstract **Instruction** superclass, we can treat it as an **Instruction**, and add it to memory. Without having known the exact type, the JVM would have constructed an instruction which is *automatically* of the correct subclass and added it to the text area of memory.

This trick allows parts of the assembler to be generic. By making these parts available, as part of the **uk.ac.sussex.cogs.peterpi.simulator.assembler** package, third parties will be encouraged to use this convention.

Hence the class **InstructionNameInstanciator**. It takes a string, which is the name of the class to instantiate, and a **Object []**, which are the arguments to be passed to the constructor. It then uses *reflection* methods such as **Class.forName** and **Class.getConstructor** to call the correct constructor for that class. All this is wrapped in a **try catch** block, with **AssemblerException**s thrown when something goes wrong.

There are two things that can cause the **InstructionNameInstantiator** to fail. The first is that it could be passed the name of a class which does not exist (remember, these strings come straight from the user), or has accidentally not been made **public**. The second is that the types of the arguments could be wrong. This can quite often occur, when the user misuses an instruction (for example, using an immediate value where a register name was required).

7 Yes, that *is* confusing! Read this page *very slowly and carefully*, as it is easy to get mixed up. Remember that actual class names and Java keywords are shown in **bold**. This page describes one of the most important design decisions of the project, and is worth taking time over.

Instructions as Visitors

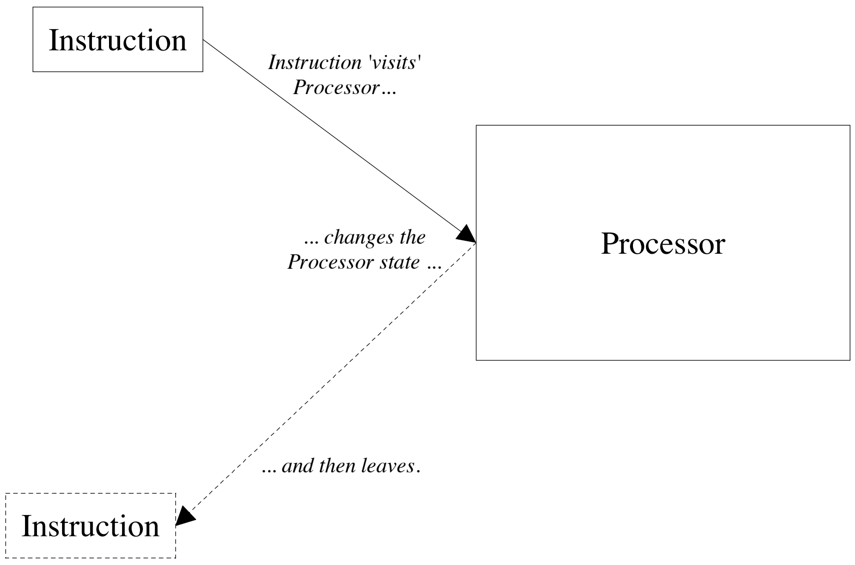
Consider a typical processor implementation. Somewhere along the flow of execution, there would have to be a decode stage. The processor would discover the identity of the **Instruction** it had just loaded from memory, and act accordingly.

Since chips can have a great many instructions, we might consider an implementation with a very large **switch** statement, defining the behavior that should occur for each type of instruction.

A better solution, perhaps, is to define the behavior for a particular instruction *as part of* that instructions class definition. The **Instruction** can then *visit* the processor, and change the processor's state accordingly. This way of doing things is known as the *Visitor Pattern*. The visitor pattern is interesting, because it is a complete reversal of the normal way of object oriented programming[8].

For example, suppose there is a class **Multiply extends Instruction**. Inside the class definition for **Multiply**, we declare a method **visit** whose job is to alter a **Processor** parameter. By the end of the method, the **Processor** will be in a state as if it had done the multiplication itself.

Diagram


Instruction
Instruction 'visits' Processor...
Processor
... changes the Processor state ...
... and then leaves.
Instruction

An Instruction visiting a Processor

8 Normally, Objects know how to change their own state. For example, an **Oven** will know how to **heat**. With the visitor pattern, objects exist only to change *another's* state.

All instructions for a particular processor have the same name for this special method, and it is declared as a method of the superclass for that instruction set. The processor can call this method, and the JVM automatically ensures that the correct version of the method is called at runtime[9].

The key advantage is that the work is delegated out amongst the class definitions. This results in smaller, easier to manage classes.

Theoretically, new instructions could be made for a particular processor without changing (or even having access to) the source code for that processor. In other circumstances, this is often a reason to choose the visitor pattern. This is unlikely to happen here though, as processor instruction sets do not change.

9 This is *polymorphism*, achieved through *late binding*.

Superclass 3: The Memory Class Set

The memory defines the state of the machine.

How the memory is organized determines the tasks a computer can perform.

In this secion,

A set of superclasses supply the interface to the rest of the system.

Other classes form a starting point for user implementations.

Introduction

The memory system is another part of the simulator (like the processor), for which it should be possible to write one's own implementation. As described previously, to enforce consistent behavior of implementations, and abstract class is made, from which all user implementations inherit.

Using int for the Address and Data

Different processors have different sized *address bus*es. The address bus is a set of bits shared by the processor and memory. During the course of a memory read, the address to read from is put onto the address bus, and the relevant data is placed on another bus called the *data bus* by the memory. During a write, both the address and data buses are set by the processor, and the memory writes the data.

The address and data busses need not be the same width[10]. However, we can expect that they *would* be the same on a RISC chip, as it stays with the RISC concept of simplicity and consistency.

I have decided to enforce that the simulated equivalents of both the address and data busses are represented as **int**s. Chips with busses of less than 32 bits (the size of a Java int) need not use the extra width. Chips with busses larger than 32 bits (e.g. the forthcoming Intel IA-64) will require this system to be reviewed. That said, it is unlikely that any students will need to simulate these chips.

10 The Motorola 68020, 68030, 68040 and 68060 CISC processors all have a 32 bit data bus, which is sensible as they are all 32 bits chips. However, they only have a 24 bit address bus. This means they can only access 2^24 memory locations, which makes for 16 Mb of memory. Nowadays, that's not much, but during the late 80's when the chips were at their commercial peak, 16 Mb was more than anybody could find a use for. The 680x0 family was a very successful line of processors, used in Apple Mac and Commodore Amiga computers.

The DataMemory Class

An interface for a system's memory could not be simpler. All that is required is a method to read a location of memory, and a method to write. In the Java convention, these are named **set** and **get**.

```
public int getData (int address)
public void setData (int address, int data)
```

It might be helpful for memory systems to be able to return their start address and size, and also whether they are byte or word addressed:

```
public int getStartAddress ();
public int getMemorySize ();
public int getUnitIncrement ();  // returns bytes per address unit
```

To ensure that the superclass has this information to return, it could be placed as a parameter to the constructor.

The InstructionMemory Class

The memory system for **Instruction** objects is practically the same. However, the read and write methods must return and take **Instruction** objects instead of **int** values:

```
public void setInstruction (int address, Instruction i)
public Instruction getInstruction (int address)
```

Because InstructionMemory and DataMemory share the size, startAddress and unitIncrement fields, those fields are put into a common superclass **Memory** (As are the **get** methods to retrieve them).

The Storage Problem

A processor using a 32 bit address bus (such as the R2000) can address 4Gb of memory. That is more than the simulator can hold[11]. It is therefore important to only store values that are actually required during execution.

With this in mind, memory implementations can start by not storing any values. When read requests come in for addresses that have not previously been written to, the implementation can return zero, or some other number, without having to put aside storage space for that data.

Only when a write request comes in for an address must storage be set aside for that memory location.

This memory system is fine for data that, once written, is relevant for the entire run of a program. Global constants declared at the assembly stage are examples of this type of value.

The **ManagedDataMemory** class described below allows areas of address space to be reserved, and then freed, in a similar fashion to the **malloc** and **free** routines of the C standard library. This makes it better suited for areas of memory which are to be used by dynamic allocation systems.

ManagedDataMemory Class

This class is designed to be accessed by memory management parts of a kernel. Blocks of memory can be reserved for use, and then declared finished with as appropriate. It would be a sensible idea for these blocks to use a memory-saving scheme like the one described above.

Since this class extends **DataMemory**, it already has the methods **setData** and **getData**. Since **ManagedDataMemory** is itself **abstract**, we do not have to give definitions for those methods. The class also has two more methods, both abstract.

```
public abstract void enable  (int startAddress, int size) throws MemoryException
public abstract void disable (int startAddress) throws MemoryException
```

The enable method tells the memory system that the specified range will be written to shortly, and any preparations for such writing should be carried out. A poor implementation of this class will set up an array of the relevant size, and instantiate all the entries. A good implementation will wait until write requests actually happen before instantiating values.

The disable method tells the memory that the specified range is no longer needed by the program, and that the memory object may choose to arrange for the storage classes to be garbage collected.

Both methods may throw a **MemoryException**, if the parameters do not make sense (e.g. an area is disabled which has not been enabled).

11 The Java Virtual Machine itself is a 32 bit computer.

DataMemory Reference Implementation: Block

The memory package supplies the **Block** class as an implementation of the **DataMemory** class, in the hope it may be useful.

An object of type Block keeps a handle to a **ComponentStore** object, which is basically a graphical hashtable. When read requests come in (as a call to **getData**), the store is referenced to see if there is a value already associated with that address. If there is, then that value is returned. If not, the value zero is returned.

Returning zero, rather than some other value, is a sensible choice. A byte with value zero is often used to terminate a string. If a programmer forgets to terminate a string, it is possible that my returning a zero will automatically terminate it[12].

When a call to setData occurs, the store again is searched for a value associated with that address. If there is one, then it is set to the specified word. If not, one is quickly instantiated and added to the store.

ManagedDataMemory Reference Implementation: Heap

The memory package also supplies the Heap class as an optional implementation of ManagedDataMemory, in the hope it may be useful.

A **Heap** object is actually a collection of **Block** objects. When a call to **setData** or **getData** occurs, the correct **Block** for the given address is found[13], and the call delegated to that object.

When a call to **enable** is made, the start address and size parameters are checked to ensure they do not overlap any existing blocks. If they do, and exception is thrown. Then, a **Block** object is instantiated, with the correct start address and size parameters sent to the constructor. The new object is placed in the **Heap**'s hashtable of blocks.

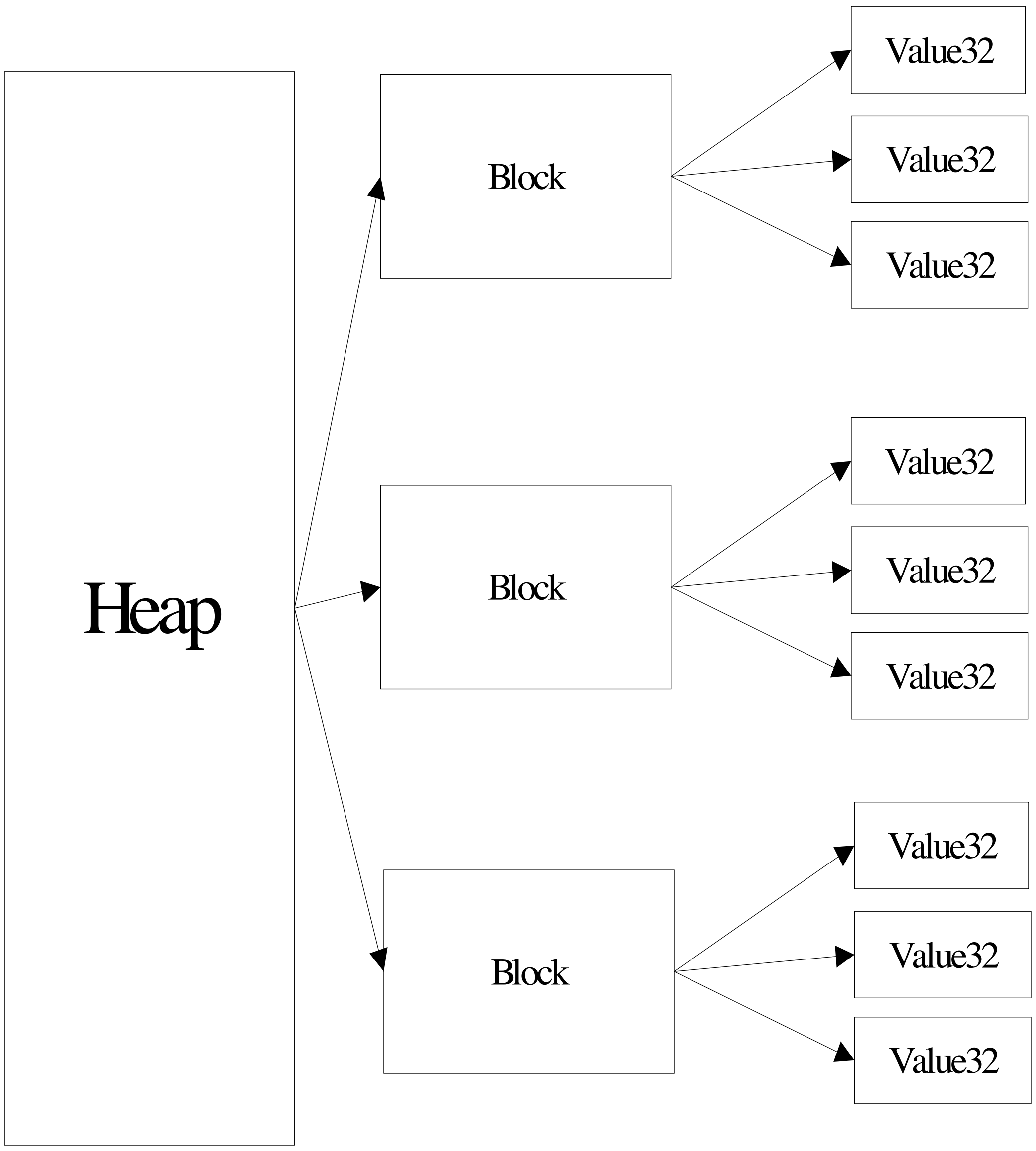
When a call to **disable** is made, the address parameter is checked to ensure that there is indeed a block starting at that address. If not, an exception is thrown. The relevant **Block** object is removed from the tables. Since there are now no handles to it, it is ripe for garbage collection. Indeed, the system garbage collector is explicitly called (System.gc), as the **Block** might have been very large.

12 Perhaps that is a bad thing; bugs might become intermittent, depending on how memory is allocated.

13 By serial search, which is a slow solution. More time would allow me to create a faster implementation, which stored the **Block**s in a way that made search faster. A storage system which allowed a binary search of the **Block**s would allow searches to be carried out in $O(\log_2 n)$ time, where n is the number of blocks stored.

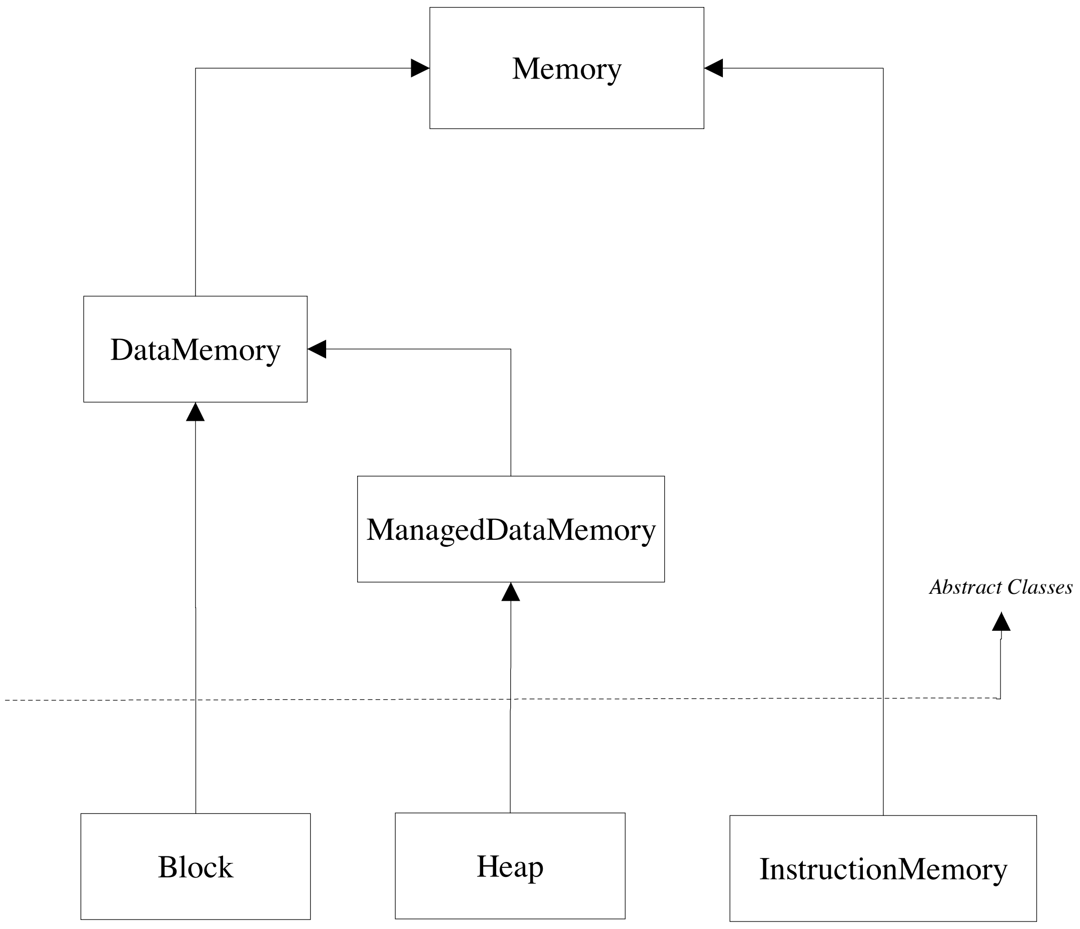
Composition Diagram

This diagram shows how a **Heap** object is made up of many **Block**s, and each **Block** is made of many **Value32** objects:



Inheritance Tree

So, we now have the inheritance tree of memory classes as shown below. The arrows point *from* a subclass *to* a superclass:



End of Superclasses

We now have a view of a complete, running computer system.
By designing abstract classes, we have defined how the components interact with each other,
and have allowed implementation subclasses to be written.

But this is only one part of a simulation;
The flow of information to the user is critical.
The component parts must be created and arranged correctly.

We shall continue to consider problems such as these.

Preparing the Plugin: The SimulatorSystem Class

We have considered the components of a running computer system,
but how can we allow a plugin system to prepare itself for use?

Introduction

The classes we have decided upon so far have all been concerned with the simulation at the point where it actually runs the users program. There are steps that need to be taken before this can happen. We can enforce that these steps are taken by requiring that for every use of the simulation, there is a **SimulatorSystem** subclass that can do the work for us. Which **SimulatorSystem** is used determines the identity of all other parts of the system. Therefore, the name of a **SimulatorSystem** should be passed in as a command line argument to the whole simulation.

Requirements

The processor, memory and kernel objects need to be instantiated. The user's program must be read and assembled. Values might have to be set that are specific to certain architectures. It is impossible for us to know what order these tasks must be carried out in for a certain combination of processor, kernel and memory.

After the configurations have taken place, the **SimulatorSystem** must be able to hand back to the rest of the simulation the processor, data memory and instruction memory. The main reason these are handed back to the simulation is for them to be placed on the user's screen.

Design

A **SimulatorSystem** object must have a method to call which makes it perform all the configurations necessary. Since part of the configuration task is to assemble the user's program, it makes sense for this method to take a **Reader** object which is connected to that file.

It makes sense for the constructor of each **SimulatorSystem** subclass to perform the configurations in its *constructor* method. That way, we can enforce that all configurations are carried out before the rest of the simulation gets the chance to call for the components.

The methods which return a handle to the processor, data memory and instruction memory should be called **getProcessor**, getMemory and getProgram respectively. The set/get naming convention is the accepted way in the Java community to name methods whose job is merely to retrieve or return a value.

Here is the code for the **abstract SimulatorSystem** class, with the methods we can considered.

By assuming that every system will have one processor, one area of data memory and one area of instruction memory, we can make these **protected** instance variables. This aids readability of the subclasses, as they will all have to use the same variable name for each component. It also allows us to define (not just declare) the **get** methods.

```
public abstract class SimulatorSystem {

    public SimulatorSystem (Reader input) throws SimulatorException {}


    protected Processor processor;
    protected DataMemory memory;
    protected InstructionMemory program;

    public Processor getProcessor () {return processor;}
    public DataMemory getMemory () {return memory;}
    public InstructionMemory getProgram () {return program;}

}
```

User Interface

The user interface is the most important part of the project,
as the program's goal is to show what's happening inside a computer.

Two Possible Designs

There are two ways in which a user interface could be incorporated into the system. Each has its pros and cons.

Design 1: The Interface as a Remote Object

In this design, the system componenents (processor, memory, etc) would be made without giving their user interface any thought. Once the system was working, the user interface could be written. Because the user interface would be a different set of objects to the simulator components, changes to each would not affect the other. This is the system I had in mind when I first started the project, as it is the most common method used for the small programs I had written up to this time.

Communication between the user interface and the rest of the system would be through a limited (and defined) set of method calls. For example, the processor might have a method an interface object could call to obtain the values of each of the registers.

Because one user interface cannot be made that is generic to all processors, each processor might have a method of the form:

```
public Component getUserInterface ()
```

Which would return a **Component** which was that processor's interface. **Component** is the top level object in the Java GUI inheritance tree. The object would not *actually* be a **Component**, but rather a subclass, such as a **Panel**, containing various other components. The Java toolkits (AWT & Swing) both form tree structures of container components at run time, each with possible sub-containers inside them. The leaf nodes of these trees are the actual buttons, labels and text components.

This design (the interface as a third party) has the advantage that large changes can be made to the user interface, without the underlying program having to be adjusted. Also, the source code for any one object will stay specific to what that object does. The code for a processor will not include any user interface work, and the user interface will not have any computer architecture terminology in it.

Design 2: A Completely Integrated UI

The second design that could be used is a little less obvious, but is quite neat. It takes the point of view that everything in the simulation should in fact *be* its own interface. Components as small as the individual registers on the processor would be able to draw themselves.

For this to work, every simulator component that is ever going to appear on the interface must extent **Component**, and be able to draw itself.

Simulator entities that are made up of smaller objects would become **Container**s, and literally have their component objects added to them, using the graphical **Container.add** method. For example, a Processor implementation might be a **Container**, and would have its register set added to it graphically. The register set itself would also be a **Container**, and would have each one of the registers added to it. Each individual register might be a **Container** too and perhaps have a text label for its name, and another for its value.

The disadvantage of this system is that it might become slow. The structure of the interface might become so deeply nested that it takes too long to redraw itself and becomes unresponsive. That said, the user would typically only be stepping through their program one instruction at a time, so it would only ever be one or two values which change per step.

This system has the big advantage that the user interface almost automatically becomes correct, and becomes very simple to write. Everything will be put in the right place. It will never happen, for example, that a processor's registers end up being placed graphically in with main memory, as it would be impossible to code it that way. Given that a processor's registers would typically be **private**, the only **Container** that could access them

For example, consider the **Processor** implementation mentioned above. It could instantiate its register set *and* add it as part of its own interface in just two lines:

(suppose the chip has an instance variable **registers** of type **MyRegs extends RegisterSet**)

```
registers = new MyRegs ();
add (registers);
```

The package javax.swing.border supplies borders which can be put around **Container** objects. One such border supplies titles, which help tell the user what a set of subcomponents are.

I decided to go with this design. The main reason was that it meant that each simulator component could keep more of its fields private, as it did not need to export their values to a separate interface object. I also liked the fact that it ties the interface layout to the structure of each class, as described above.

The Value32 Class: The Graphical Integer

```
(uk.ac.sussex.cogs.peterpi.util.Value32)
```

Once the design decision had been made that everything in the simulation was a graphical component, it became obvious that a class would be required whose job was to be a graphical equivalent of the humble **int**. A component which acted like (and displayed itself as) an **int** value would form a component part of nearly every other part of the system. This class was therefore the most important to look good, and be easy to work with.

This class is called **Value32**. Since its scope of usefulness extends beyond this simulation, it is placed in the package:

```
uk.ac.sussex.cogs.peterpi.util
```

Since a Value32 object stores an **int** value, the most obvious methods this class must have are a set / get combination:

```
public int getValue (); // returns the int this object represents
public void setValue (int newValue); // sets this object to represent 'newValue'
```

A **Value32** object would almost always have some name associated with it (e.g. "Stack Pointer"). This text would definitely be displayed on the screen, so we can guess immediately that it will be stored internally as some sort of graphical text component. However, the interface this object has with the rest of the system should not involve graphical objects. The name of a **Value32** should therefore be set and queried as a **String**:

```
public String getName ();
public void setName (String newName);
```

There are different ways a 32 bit quantity can be interpreted as. The most common way is as a signed integer, but other formats include unsigned values, floating point values and even 4 ACSII characters. Obviously, an object will only be able to take one of those guises at a time. There must be some method we can call to tell a **Value32** object hot to display itself:

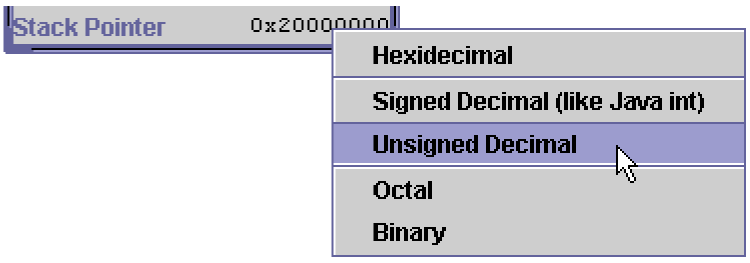
```
public void setMode (int mode);

(public static final fields might be used to refer to modes,
 eg UNSIGNED_INTEGER = 1)
```

The ValueFormatPopup Class

A popup menu is a sensible way to let the user choose the format of a **Value32** object. The ValueFormatPopup class achieves this by extending JPopupMenu from the Swing library.

Value32 objects are set to instantiate a new object of this class whenever they receive a right mouse click[14]. The popup menu contains a series of menu items, each one for a different formatting mode. Each menu item has an **ActionListener** which sets the mode of the **Value32**.

Here is a screenshot of a Value32 & ValueFormatPopup in use.



Failed use of the Singleton Pattern #1: The Popup Menu

The **ValueFormatPopup** gives us the opportunity to use the singleton design pattern. This pattern is used when there need not ever be more than one instance of an object. All constructors of the class are made **private**, and a **static** method provides handles to an instance. The **static** method always returns a handle to *the same* object, so ensuring only one is ever in existence.

There need only ever be one **ValueFormatPopup,** if it could be made to know which instance of **Value32** it is setting the mode of. The **Value32** just clicked upon could register itself with the popup, (e.g. **popup.register (this)**) so as to say "It's me you're adjusting".

This was done, but during testing, the menu would behave erratically. It would pop up in strange places, sometimes behind other windows.

As soon as the design pattern was removed (and popup instances created 'on the fly' as required), the menu behaved correctly. It would appear that graphical components must have internal fields which make this system impossible. This is a shame, because otherwise it would have been a textbook-example of the pattern, and would have saved time and memory[15].

14 A Swing bug stops this working on Solaris. A patch to the source code makes the menu appear for *any* mouse button.

15 Once Java objects are finished with, they stay in memory until the 'garbage collector' runs. The act of running the garbage collector is computationally expensive, but on the other hand, *not* running it is expensive on memory. The garbage collector can be explicitly called to run, but normally the JVM decides when it will run.

Failed use of the Singleton Pattern #2: Layout Managers

Another place where the singleton pattern might have been used is for the layout managers. Each graphical component which is a C**ontainer** (holds other components inside itself) is associated with an object called a LayoutManager. The AWT and Swing libraries supply a set of useful layout managers.

Instances of layout manager classes can be unique, such as leaving a certain space between the child components, a setting a grid of a certain size. Most of the time though, the defaults work find. Because of this, it is usual to create and assign a layout manager in a single line, leaving the layout manager anonymous:

(inside the constructor for a container)

```
setLayout (new BorderLayout ());
```

Here, we can see that a **BorderLayout** (a type of layout manager) is instantiated and assigned, all in one line. If several of these **Container** objects are made, they will all have their own unique manager.

This appeared to be a poor design; a waste of memory. A class called **Layouts** was placed in the '**util**' package, which had **static** methods to return oft-used layout managers.

As with the other failed use of the singleton pattern, odd behavior occurred at run time. Containers would lay out their components in the wrong place and at the wrong time. There was little order to the errors.

Like with the popup menus, there must be some structure in the layout manager system which objects to manager objects being assigned to more than one container.

The VisibleSimulatorComponent Class

`(uk.ac.sussex.cogs.peterpi.simulator.VisibleSimulatorComponent)`

How can we *enforce* that the processor, instruction and data memories are all graphical components?

As ever, inheritance is the key. We can make some class extending a suitable graphical component (JPanel) *from which* **Processor** and **Memory** both extend (Remember that **DataMemory** and **InstructionMemory** both extend **Memory**, and so will be included in the arrangement).

This superclass is **VisibleSimulatorComponent**.

There is nothing really that can be put into the definition of the class. It only exists because it is more intuitive to write:

```
class Processor extends VisibleSimulatorComponent
```

... than it is to write:

```
class Processor extends JPanel
```

That said, one useful thing we might want an object of this class to do is to return a string description of itself. This is quite important, because a **Processor** or **Memory** might come from any author from any organization, and represent any processor or memory system. This description could form the title of a surrounding window for a simulator component:

```
public abstract String getDescription ();
```

Controlling the System: The Controller Class

When should a program start, and when should it stop?
The user must be allowed to control the system.

Introduction

A window (or other area) containing actions that the user might want to carry out must be present. This area might also contain other information which does not refer to any one simulator component. If the number of actions that can be carried out by the user is small, it makes sense for each one to be a button in this area. If the number of user actions is high, a drop-down menu might be better.

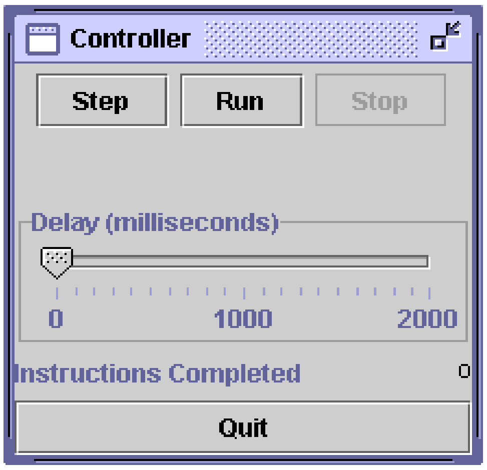
User Actions

When we consider the number of things a user might want to do, we see that the choice is small:

- Execute a single instruction
- Execute many instructions consecutively
- Stop a running program

These actions can be put into a class without any trouble. The resulting class is called **Controller**.

Here is how the Controller object ends up on the screen:



The Controller

We can see that there are few enough components for the area not to be too confusing. However, it is reaching the limit. If any other functions were added, the area might need to be organized in a better way, perhaps with pull down menus.

Since a controller must be able to call the processor to **tick** (as a result of pressing 'step' or 'run'), it needs a handle to the processor. This is a constructor parameter.

MultiThreading the Controller

In a graphical application, it is important to know how the program is running. In Java, and most languages for that matter, once a graphical application is up and running, it becomes *event driven*. This means that some *event* will occur (typically a mouse click on a button), and as a result, some *actions* will be carried out. Once the actions are completed, the program returns to a sleeping state, waiting for the next event.

If we stop to think about most desktop applications, such as word processors, web browsers and paint programs, we can imagine the events occurring, and the program returning to its dormant state after it has reacted to the events. In the word processor, the events are key presses. In the browser, they are mouse clicks. See that the actions for an event are processed before the next event is fired.

This system is fine when the set actions to carry out for each event is finite. Almost always, that *is* the case. However, consider the 'run' button on the controller. When we press it, we place the program in an infinite loop, continuously calling the **Processor.tick** method. It is quite possible that this loop will *never* exit.

While this loop is running, mouse presses on other buttons (including 'stop') are not responded to.

The answer to this problem is to use *threads*. A complete discussion on Java threads is enough for a book alone (*Java Threads*, O'Reilly). Here, it is enough to say that the program execution seems to split in two. One half enters that infinite loop, the other returns, ready to respond to subsequent events.

If the 'Stop' button is pressed, it is responded to by the thread that *is not* in the loop. The thread that *is* in the loop is made to finish, and it disappears just as magically as it came into being.

The 'Stop' and 'Run' buttons are mutually exlusive. When the program is running (and there are two threads) it is impossible to press 'Run' again. If the program is not running, it is impossible to press 'Stop'.

An early prototype had the 'Run' button magically turn into the 'Stop' button when it was pressed, so only one of them was on screen at any one time. However, It was found that users did not notice the switch. As they searched for a 'Stop' button, the last place they looked was the space they just clicked on as the 'Run' button! This is interesting, because I imagined the system might be advantageous, as it took up less screen space (1 button instead of 2).

Handling the Unexpected: The Exception System

Sooner or later, an unexpected situation will occur.
How will the simulation react?

Introduction

Almost every action a computer system carries out makes some assumptions. Normally, these assumptions turn out to be true, and all is fine. But sometimes our assumptions are wrong. If an error goes unreported, it can have very bad effects for the running system. We call these exceptional circumstances *exceptions*. A processor will have a set system for detecting and responding to exceptions, and this must form part of any implementation.

The most well known and easily understood exception that can occur is *arithmetic overflow*. This occurs when the result of some arithmetic function (typically addition) is too big to fit in a processor's register. What should happen next depends on the circumstances of the user program. If the large value can be acceptably approximated by a smaller one, this might be made to happen. If the value is needed in exact form, the system must be alerted to the problem. However, these choices are at the user level; the processor must *always* be made aware of exceptions.

Arithmetic overflow is by no means the only exception that occurs. Others include trying to read or write to a memory location which does not physically exist, or jumping the flow of execution to an address holding no instructions.

The Java Exception System

Java has an elegant system for dealing with exceptional circumstances. The system enforces that action is taken when an error occurs, but does not result in the paranoid coding style required in C[16]. The Java system is too involved to explain here. Instead, the reader is referred to chapter 9 of *Thinking in Java* for a well written explanation.

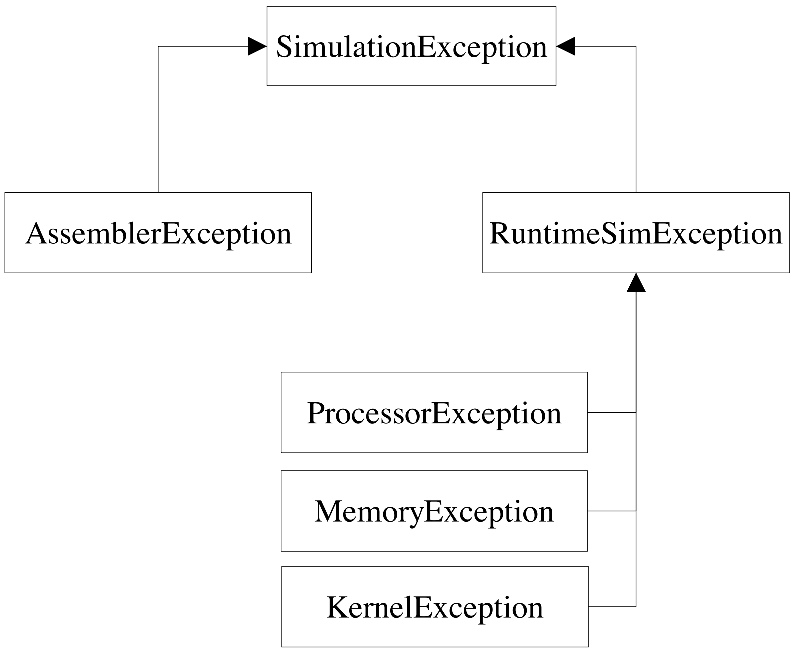
An inheritance tree of **Exception** objects that are to do with the simulation is required. Some of these exceptions have been mentioned already in this paper. Generally, for each section of the simulation, there is a corresponding type of exception. For example, objects to do with the memory system have methods which may throw a **MemoryException**. Processors may throw **ProcessorException**s during their **tick** calls. The name of the exception indicates what system component caused it to occur.

The types of exceptions can be split into two families; those are thrown at run time, such as a 'segmentation fault' or arithmetic overflow, and those that are thrown during assembly, and are therefore problems with the syntax of the user's program or with ***Jukebox***'s initialization routines.

16 ... where you effectively have to assume that an error *did* happen on each and every procedure called. When reading a C program, it's difficult to see which lines are error checking, and which lines actually *do* something. Java splits the lines up into a **try** block, where the action takes place, and a **catch** block, where recovery takes place.

Inheritance Diagram

Here is an inheritance tree of the exceptions used:



When the Exceptions are Thrown

Fortunately, the two types of exception are thrown at different times during the course of execution, which makes them easy to distinguish.

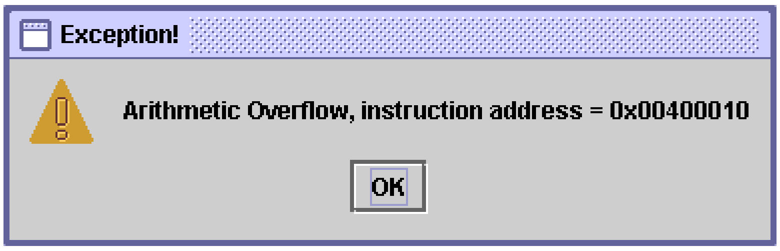
AssemblerExceptions are thrown before anything becomes visible on screen. If an **AssemblerException** is thrown, then the user's program does not make sense, and it is therefore impossible to carry on with the simulation. The simulation exits, and the exception's *detail message* is printed. An example of a detail message for an **AssemblerException** is 'Unknown Instruction', along with the relevant line of the user's text.

Subclasses of **RuntimeSimException** are thrown during the execution of a user's program, once the simulation is active on the user's screen. These exceptions represent the *real* exceptions that happen in hardware. **RuntimeSimException**s are much more interesting, as they are not *always* fatal.

For example, a half-written (see 'Future Enhancements') feature of the R2000 class is that if a **RuntimeSimException** is thrown during a call to **tick**, the processor may choose to *catch* the exception, and set some of its own registers accordingly. This is an imitation of how the *real* processor would deal with a *real* exception (as opposed to a Java one)

The tick method call, and any it may call, are all further down the call stack[17] than the method calls of the parent simulation (such as **Controller**). If the processor chooses to catch the exception, then unless it explicitly rethrows the exception, then the **Controller** and the rest of the simulation will be unaware that an exception even occurred.

If a RuntimeSimException occurs which is not caught be any implementation classes, then it is caught by the **Controller**, which displays a warning message to the user. Here is a screen shot of a warning message caused by arithmetic overflow:


Exception!

Arithmetic Overflow, instruction address = 0x00400010

The fact that the program does anything *at all* is a huge improvement over SPIM, which would typically crash.

17 When exceptions are thrown, they travel back up the call stack, until they are 'caught' by a **catch** block. If an exception is caught, methods further up the call stack are unaware that the exception ever happened.

The Simulator Class

*We have seen how the **SimulatorSystem** prepares a plugin for use.*
*The **Simulator** class boots & prepares the entire simulation.*

So far, most of our discussion has been on the system as it is during execution of a users program. We have seen how a **SimulatorSystem** implementation can instantiate components, but even *it* is not actually an executable class (with a **main** method) which can be called at the command line.

This completely initial startup is supplied by the Simulation class. This class is not overridden for different implementations.

The steps the simulation class takes are simple, and do not require a detailed design:

1. The second command line argument will be the name of the program to load and run. This file is opened, and wrapped in a **BufferedReader**.

2. The first command line argument will be the name of the **SimulatorSystem** to use. This class is loaded and instantiated. The file opened in the previous step is passed in as a parameter.

3. Once the **SimulatorSystem** constructor returns, we know that all components are ready, and the program has been assembled. Windows are created holding the processor, program and data memory. These objects are obtained through methods such as **getProcessor**, called on the newly instantiated **SimulatorSystem**. A final window containing a controller panel is made.

4. These windows are placed inside a single virtual desktop which helps the user relate them all to this program. The virtual desktop is maximised, and the simulation is ready to run.

As we can see, all that this class basically does is to instantiate a **SimulatorSystem**, and arrange the initial appearance on screen. Such administrative tasks are not very exiting to program, but are nonetheless required.

We will execute the simulator like this:

```
java Simulator <name-of-system> <name-of-user-program>
```

On a UNIX system, we can simplify this by giving a shell script for **SimulatorSystems** we know about. For example, in the next section, we will encounter a **SimulatorSystem** called **SpimSystem**. We could make a shell script **SpimSim** with the content:

```
java Simulator uk.ac.sussex.cogs.peterpi.simulator.spimSystem.SpimSystem $1
```

$1 is replaced with the first argument to the script. So to execute the simulation running a user program **programFile**, we might type:

```
SpimSim programFile
```

At the command prompt.

Solving Common Problems: The BinaryMath Class

The same problems tend to appear again and again in any program.

It makes sense to supply a common solution.

The need for the BinaryMath Class

It is immediately obvious that in a simulation of a processor, a great amount of bit manipulation is going to be done on Java's numerical data types. Java ints are used to hold data items in memory and registers (through the graphical **Value32** wrapper). Java ints are useful for this task, because they are *guaranteed*[18] to be 32-bits wide.

The fact that a Java int is always 32 bits is an advantage, as we can safely say that an int *will* hold a 32 bit signed value, or 4 ascii characters, or whatever other type we can store in 32 bits.

The fact that a Java int is *always signed* proves to be a great pain though. When the simulation is required to imitate a processor instruction which acts on *unsigned* data, we must trick the Java runtime into thinking the int values are unsigned. It turns out that this can be done by or-ing the int with a long (64-bit) zero, and working on the resulting long.

It would be useful to provide solutions to these common problems in a class, so that they need not be repeated across the program. Apart from being difficult to read, bit manipulation in Java is also difficult to write, as you can never be sure whether you can trust an operator like '>>' to do what you had in mind.

The Structure of the Class

For conventional mathematical problems of any complexity, we can normally find a method in the **java.lang.Math** class which computes it for us. These methods are normally static, so we can just call **Math.(method)**, rather than actually having to make a **Math** object.

It seems a good idea to copy this system for a **BinaryMath** class. The class contains many **static** methods which perform tasks to do with bit manipulation and the like. The class is placed in a package outside the scope of the simulator, because it is completely general to all applications requiring it.

18 Unlike in C, where the int datatype (and all the others, for that matter) can be whatever size is most efficient for the hardware.

Testing

No Testing can be Done!

At the current stage, there is now way that any testing can actually be done, as all of the classes we have considered have been abstract, and are meant as a template for an actual implementation. There is no way we can start the program (not even in the debugger).

The testing of the first implementation; the Spim system, includes testing of the template classes introduced in this section.

In an object oriented language, the compiler does so much type checking for you, that a lot of errors to do with type mismatches are solved at compile time. In C, which is much less fussy about type, finding such errors (e.g. returning a character instead of an int) would form a definite percentage of testing efforts.

In Java, there is a garbage collector, which automatically reclaims heap memory (i.e. objects) that is finished with. In C++ (and C), this must be done explicitly. Forgetting to do this leads to *memory leaks*. Finding and repairing memory leaks is another large part of the typical testing phase.

It is not arrogant to say that if a program *compiles* in Java, then it will almost definitely *run*. There might be logical errors in what the program actually *does*, but these are normally obvious at run time (e.g. an addition instruction carrying out multiplication).

End Of Chapter

We have considered all parts of the generic simulation,
but an architecture plugin is required for it to run.

The next two chapters discuss two very different plugins;
The SPIM plugin simulates a real life processor.
The DM0 plugin simulates a fictional system used to help compiler designers.

System 1: The SPIM System

A Plugin for the ***Jukebox*** Simulator

Contents

Introduction

The SPIM System forms a plugin for the ***Jukebox*** generic simulator described in the previous chapter. Therefore, it is not an executable program in itself.

This chapter makes use of terms introduced in the previous one. Do not read about the SPIM System until you have read about the *Jukebox* simulator.

This system will supply the functionality my simulator requires to behave like the SPIM simulator I hope to replace. In this sense, it represents the majority of my efforts.

It also includes an implementation of a real processor; the MIPS R2000.

Requirements

The requirements of this module are the same as those for the whole project; to supply an accurate simulation of the R2000, and the rest of the SPIM simulator.

Processor: MIPS R2000

The R2000 is a real life processor, which the SPIM system simulates.
The accurate imitation of this chip forms the core of this plugin

Introduction

The implementation of the R2000 processor is the core of this project. By completing this part, I will show my ability to simulate a real world processor.

Requirements

The requirements of this module are simple. The simulated processor should behave as closely as possible to a real R2000. It would also be useful for a **SimulatorSystem** implementation to be present, which users of the R2000 can extend.

The R2000 chip is described extensively in the book *Computer Organization and Design; The Hardware Software interface*. A publicly available appendix lists all the R2000's instructions, the register use, the memory use, and everything else required of this project.

Design

To make the processor, we declare a class **R2000 extends Processor**. The chip can be defined by the contents of its registers, so these must be placed as instance variables in the class.

Maintaining the Registers

The R2000's registers can be split into two semantically similar groups; the user registers and the system registers.

The user registers are the ones which can be accessed by instructions directly. There are 32 of these. Some of those registers have special uses to the assembler programmer. For example, the register named "$sp" is used by the stack pointer. These conventions allow assembly subroutines from different authors to be used together. At the simulation level, we care not for these conventions, but we do need to supply the register names for the user interface.

The system registers are the ones whose values are set as a side effect of instruction execution. The most obvious of these is the program counter. Most instructions[1] cannot read or write to the program counter directly. Other examples are the registers HI and LO, which are used to store results of multiplication and division. They cannot be used by regular instructions, only specific ones.

The floating point unit (FPU) also has user registers and system registers. Unfortunately, it became obvious quite early on in the project that there would simply not be enough time to model the FPU. A description of how one might go about adding the FPU is given later in this paper. Thanks to the clean design of the processor model, adding the FPU would not be difficult, only time consuming.

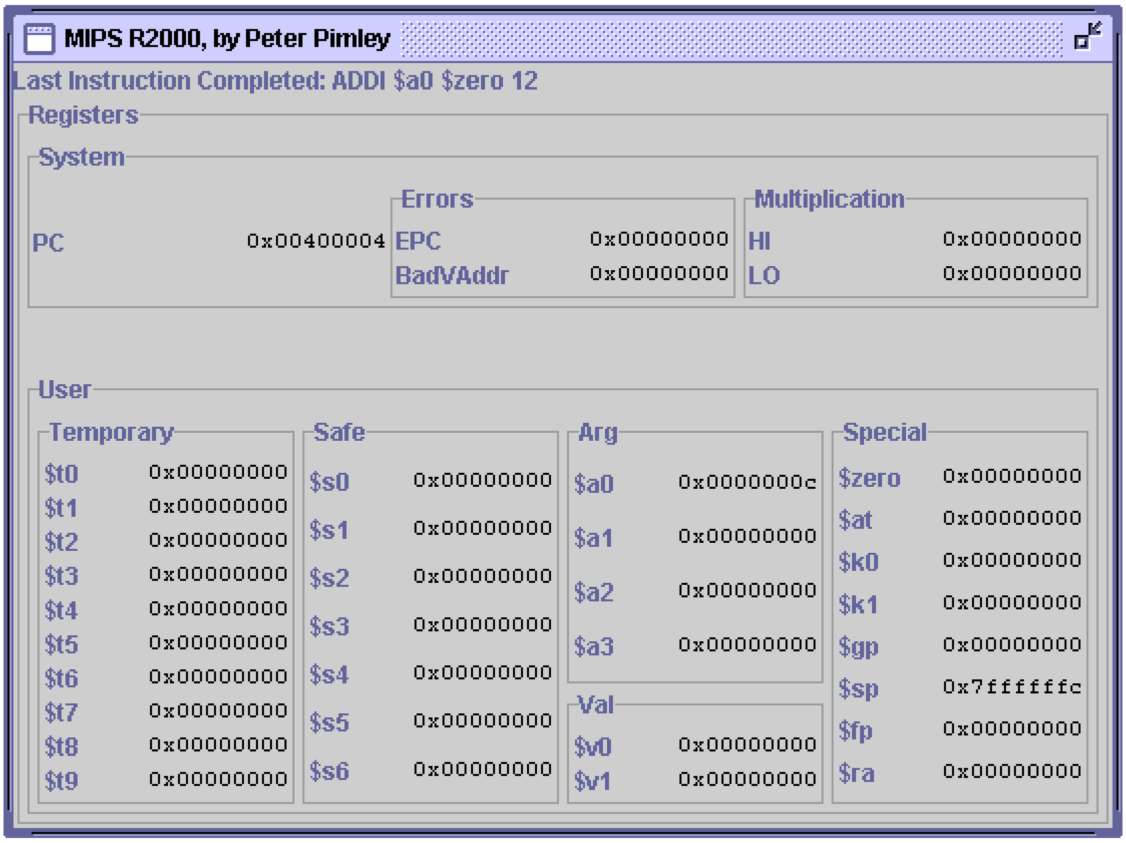
Instead of adding all the registers immediately to the processor, we make use of the **RegisterSet** class, described earlier with the **Processor** superclass. If we had not done this, there would be too many instance variables of **R2000** for it to be maintainable. The user registers are placed in one instance, the system registers in another.

The user registers are actually held in a subclass of **RegisterSet**. Although the registers are normally referred to by their names in assembler, they may also be referred to by their number. For example, the strings "$t0" and "$8" both are names for the same register. The **UserRegisters** class overrides the **protected makeRegister** method to add this extra name automatically.

Even after splitting the register set into two groups, each group still holds a lot of registers. This is not a problem for the internal hashtables, but it would make for a cluttered interface. The registers can be put into **JPanel**s with titled borders during the set's constructor. This makes their on screen appearance much clearer. A screenshot of the two register sets is shown overleaf. This is running

1 With the exception of jump and branch type instructions.

on a Windows system, but thanks to Swing, it looks identical on any platform.


MIPS R2000, by Peter Pimley

Last Instruction Completed: ADDI $a0 $zero 12

Registers

System

PC	0x00400004

Errors

EPC	0x00000000
BadVAddr	0x00000000

Multiplication

HI	0x00000000
LO	0x00000000

User

Temporary

$t0	0x00000000
$t1	0x00000000
$t2	0x00000000
$t3	0x00000000
$t4	0x00000000
$t5	0x00000000
$t6	0x00000000
$t7	0x00000000
$t8	0x00000000
$t9	0x00000000

Safe

$s0	0x00000000
$s1	0x00000000
$s2	0x00000000
$s3	0x00000000
$s4	0x00000000
$s5	0x00000000
$s6	0x00000000

Arg

$a0	0x0000000c
$a1	0x00000000
$a2	0x00000000
$a3	0x00000000

Val

$v0	0x00000000
$v1	0x00000000

Special

$zero	0x00000000
$at	0x00000000
$k0	0x00000000
$k1	0x00000000
$gp	0x00000000
$sp	0x7ffffffc
$fp	0x00000000
$ra	0x00000000

The R2000 Registers

Because **RegisterSet** objects are themselves **JPanel**s, they can be arranged as required. Here, the two register sets (User & System) have been placed one above the other, inside on parent **JPanel**. The parent panel carries the titled border "Registers".

See how both register sets subdivide themselves visually into **JPanels**, grouping registers into manageable sized sets. This deep nesting of **JPanel**s is the most effective way of maintaining a tidy user interface in Java. The Swing component set is small and fast enough to stop the interface slowing down due to excessive nested calls during display.

Also, see the top field; "Last Instruction Executed". This has nothing to do with the registers, but is worth noting. It serves to help the user see where they are in the program. At each call to**tick**, this field is filled with the result of **i.toString**, where **i** is the instruction executing on that step. Here, we can see that the ADDI instruction has just been carried out. Sure enough, the value 12 is in register $a0.

Implementing the tick Method

In order for the **R2000** class to extend **Processor**, there are certain methods it *must* define. The most important of these is the **tick** method. It defines what steps the processor must take to execute an instruction.

A real life R2000 would carry out the traditional *fetch, decode, execute* cycle[2]. However, as we shall see later, our simulated version uses the visitor pattern[3]. Because each instruction takes care of the execution, and polymorphism takes care of the decode stage, all that the R2000 itself has to do is the fetch.

The fetch stage is simple. It only requires a few lines of code. The value of the program counter is obtained, and used as the address parameter for a **getInstruction** method call on the instruction memory. The result of this call is an **Instruction** object. This can be safely cast to an **R2000Instruction**. Then, we simply call the **visit** method on that instruction, and the decode and execute stages happen automatically.

There is only a little housekeeping to be done. During the execution of the instruction, the program counter retains its value as the address of that instruction[4]. The program counter is increased *after* the call to visit, unless it has been adjusted *during* the call (i.e. a jump occured). The address of the program counter before the call is retained, so that it and the program counter value afterwards can be compared. If the two differ, a jump occurred, and we should leave the program counter alone.

Also, the R2000 has a special register **BadVAddr**, which is set to be the address of the last instruction, *if* that instruction caused an exception to be thrown. The existing exception system suits this perfectly. The call to **instruction.visit** is surrounded by a **try** block. If a run time exception is thrown during the execution of that instruction, control immediately returns to the corresponding **catch** block in **R2000.visit**. This block sets the value of **BadVAddr**.

The behavior after that depends on the instance boolean variable **doTrapHandling**. If this is true (the default is false), the program counter jumps to the *trap hander*, located at a reserved address. Otherwise, the exception is thrown again, and caught by the simulator, which will alert the user.

Another register, **EPC**, holds bits which determine the cause of the exception. Its behavior has not been implemented, but would be put in the same place to where **BadVAddr** is set.

2 In the later R3000 model, the stages were separated with data buffers, allowing them to be carried out *simultaneously*, each stage working on separate instructions. This is called *pipelining*, and allows for considerable speed increases. Pipelining was a major step forward in processor design.

3 Described in the section "Instructions as Visitors"

4 As opposed to incrementing before the call to visit, and holding the address of the *next* instruction.

The Instruction Set

In the previous section, we saw how the chip expects
each instruction to define its own behavior.

To arrange the instructions into a sensible structure will help in this task.

Introduction

The R2000 is a RISC processor, which means it does not have many instructions compared with other processors. However, it still has about 50, so managing the behavior of all of them inside the R2000 class is impossible; the class would get to big to maintain.

With this in mind, it was decided to use the visitor pattern, as described earlier in this paper. Also, the convention of naming instruction classes exactly as they appear in code was used. This would make the assembler simpler.

Instruction Hierarchy

Arranging the instructions into an inheritance tree will help reduce repeated code. The goal is that instructions that are similar will be near each other in the tree, and can call code of a common superclass.

Layer 0: The Root Superclass

A class **R2000Instruction extends Instruction** can form the root of our inheritance tree.

To be sure that all instructions for the R2000 had a **visit** method, it could be added as an **abstract** method of the **R2000Instruction** class. Fields that were common to all instructions are also added. Most notably, all instructions have a field called an opcode (operation code) which serves as the main form of identification.

The **toNumber** and **toString** methods of **Instruction** would have to be overwritten. We shall consider this later.

So far, our class definition might look like the following:

```
abstract class R2000Instruction extends Instruction {

  private int opcode;

  R2000Instruction (int opcode) {
    this.opcode = opcode;
  }

  abstract void visit (R2000 chip) throws RuntimeSimException;
}
```

Indeed, the actual class is not much different. It has a slightly different constructor, and **toNumber** and **toString** methods. We discuss all these later.

Layer 1: Hardware Format

The real R2000 helps us form the first layer of instructions. It groups instructions into three types:

1. J-Type: These are jump instructions. They hold an offset[5] to jump the program counter by, as a 26 bit field. The opcode occupies the other 6 bits to make a total of 32.

2. IType: These are instructions which hold an immediate value. An example is the ADDI instruction, which adds an immediate onto a register. The immediate is held in the lower 16 bits of the instruction. It holds 2 register indexes ("source" and "target") and the opcode in the upper 16 bits to make a total instruction length of 32 bits. Immediate values larger than 16 bits wide simply cannot be stored.

3. RType: These instructions act on three user registers. Typically, two registers "source" and "target" are used as inputs to an arithmetic function (like addition), and the result stored in the third "destination" register. All R-Type instructions have the same opcode, and use a second "function" field for identification. R-Type instructions are 32 bits wide. A special 6 bit field called the "shift amount" (shamt) is used for bit shift instructions.

See how all three instruction types are 32 bits wide. Not all processors have fixed instruction lengths, but it is part of the RISC goal of simplicity and consistency.

These types give us the first levels of the inheritance tree. We shall make classes **JType**, **R-Type** and **I-Type.** Since R and I type instructions share a common pair of registers; source and target, we shall put them in a class called **NotJType**. R and I type classes both extend **NotJType**.

5 Since the program counter's value is always on a word boundary (i.e. a multiple of 4), the offset can be shifted right 2 bits before being stored in the instruction. This allows for jumps up to 2^28 bytes, which is good enough for most programs. The JR instruction allows jumps of 2^32 bytes; the entire address space.

Layer 2: Written Format

The constructors for the classes we have decided on so far require *all* the fields that type of instruction has. For example, the constructor for an RType requires values for all three register indexes, the function code, and the shift amount.

Few instructions are written using all the fields for that type of instruction. For example, the R-Type ADD instruction does not require us to explicitly state the default shift amount of zero. So, somewhere between the call to instantiate ADD and the call to instantiate R-Type[6], these fields must be created from known defaults.

The creation of these default fields is something which will be shared between all instructions written in a particular way (e.g. all R-Types written `<instruction> <dest> <source> <target>`) It makes sense for this work to be centralized into classes in between the hardware format layer (R-Type, I-Type, J-Type) and the actual instruction objects.

For example, many R-Type instructions are written . There could be a class extending R-Type for which the constructor takes in the dest, source and target fields, creates a value of zero for the shift amount, and then calls the R-Type constructor.

It makes sense for those classes to be named in a way which makes their format obvious. I have used the following naming scheme for this layer of classes:

D = dest register index

S = source register index

T = target register index

Shamt = Shift amount

Imm = immediate value.

From this key, the name of the class is simply the fields required, in the correct order. The name of our example class taking dest, source and target fields would therefore be **DST**.

Fortunately, the written format of every R2000 instruction is available in appendix A of *Computer Organization & Design*.

6 Every constructor must call a constructor of the superclass in its first line.

Not many programmers appreciate this fact, because if a call to the superclass' constructor is *not* made, the compiler secretly puts one in, using the constructor which takes no arguments. The only time this becomes a problem is when the superclass *does not have* a constructor that takes no arguments, and the compiler is then stuck, and complains.

If a class is defined with *no constructor at all*, then it is given a default one by the compiler. The default constructor takes no arguments, and returns immediately.

Layer 3: Semantic Grouping

So far, the layers of our tree have all been to do with reducing saved code during the constructor for each type of instruction. In the next layer, instructions that might share code and fields during their execution (that is, their **visit** method), are grouped[7]. We shall see why this is advantageous in the later section "Using the Hierachy during Execution".

Layer 3 classes need not be introduced if they are not required. Examples of level 3 groups are:

Branch extends TSImm

This class has the protected method **branch**, which accesses the program counter of the chip, and offsets it by the amount held as that instruction's immediate.

LoadStore extends TSImm

This class has the protected field **address** which is the result of addition of the source and the immediate. Instructions use this during memory access.

Layer 4: Actual Instructions

The final layer of the tree is the set of instructions themselves. Each instruction takes arguments in its constructor in the same order as when the instruction is written. Normally, these fields can be passed straight to the superclass constructor. Each instruction has an opcode or function code, which is wrapped in an **Integer** and passed to the superclass' constructor.

It is not strictly required that the opcode or function code be wrapped in an **Integer**. Pass-by-reference is certainly not required[8]. However, all the other objects in the constructor of real instructions *must* be objects. If the ints are not wrapped, It gets confusing as to what is an **int** from a previous inheritance level, and what is an **Integer** from the **InstructionNameInstantiator**.

7 Fortunately, instructions that we can group in this layer all tend to be in the same groups in previous layers. For example, nearly all the branching instructions are subclasses of the SImm layer 2 class.

8 Java has a similar set of primitves to C++. These values (int, float, boolean, etc) are stack variables, and are always passed by value. When pass-by-reference is required, one uses the 'wrapper classes' Integer, Float and Boolean. These are full objects which are, of course, passed by reference.

Implementing Behavior

We have arranged the instructions into a neat tree structure.
Now, we must populate this tree by defining the behavior of each instruction.

Implementing the toString Method

Each instruction must be able to generate a string representation of itself. At first, it seems a good idea for each layer 4 (i.e. 'actual') instruction to have its own toString method. For example, the toString method of the SRL (Shift Right Logical) instruction might look something like this:

```
public String toString () {
  return "SRL " + rdIndex + " " + rsIndex + " " + rtIndex;
}
```

Whilst this looks reasonable enough, a similar method would have to be put into every instruction, making for a lot of repeated code. Also, the user expects to see the instruction written in *exactly* the same way as they wrote it. A user might prefer to use lower case letters for the "SRL". They might choose to use the optional commas that separate registers. It is impossible to generate a string automatically which will suit every user.

Instead, we can store the line that was *actually written* by the user, and use that as the string representation. The line can be passed in as a parameter to the constructor of every instruction. If this approach is taken, then the instruction will appear on the screen in *exactly* the same way as the user wrote it.

This has the added advantage that if the user declares a label on the same line as an instruction, the label appears as well.

Since every R2000 instruction object will have this representation system, the string can be stored in the superclass **R2000Instruction**. The **toString** method can also be defined in that class. By requiring a **String** object in its (only) constructor, the **R2000Instruction** class ensures that every subclass can supply a String.

The class definition given in the section 'Layer 0' now becomes:

(new fields & methods in bold)

```
abstract class R2000Instruction extends Instruction {

  private int opcode;
  private String command;

  R2000Instruction (String command, int opcode) {
    this.command = command;
    this.opcode = opcode;
  }

  abstract void visit (R2000 chip) throws RuntimeSimException;

  public String toString () {return command;}
}
```

Implementing the toNumber Method

As mentioned during the introduction to the class earlier in this paper, all **Instruction** objects must also be able to give a numerical representation of themselves. This value is requested via the toNumber method.

We have seen with the **toString** method, that it is nearly always best to put the method calls as far up the inheritance tree as possible. Since objects of type **JType**, **RType** and **IType** have all the fields required in the 'real' equivalent, we can implement the **toNumber** method at their layer.

In doing this, we use the inheritance tree that we have so far to full effect: In the root class **R2000Instruction**, we define a **toNumber** method which takes an initial zero, and performs a shift and OR to put the opcode in the correct place[9]. In the I, J and R type classes, we override this method, but in each, the first call that is made is to **super**.**toNumber**. This returns a value that is all zeros, apart from that the opcode has been filled in for us. On the following lines, the rest of the number can be constructed, again using shifts and ORs. No two classes share any repeated code in their **toNumber** methods!

Since no subclasses will ever need to overwrite **toNumber** or **toString**, we can declare them **final**. A **final** method cannot be overridden; the compiler enforces this. Calling a **final** method incurs slightly less overhead than a non-**final** method, as the call can be bound at compile time (and even inlined, if the compiler so wishes).

If the reader found the previous paragraph interesting, he should read the page 'Polymorphism in Java and C++' in the 'Notes' section of this paper.

9 All types of instructions have their opcode in the same high end 6 bits. This is quite fortunate for the toNumber implementation, but is not surprising considering that the R2000 is a RISC chip, and aims for simplicity.

Implementing the visit method

This page describes one of the most important parts of the project, and should be read with care.

Every Layer 4 instruction should have a visit method, defining its behavior upon the processor. As we have seen in the section 'Instructions as Visitors', the processor will call this method when it requires a certain instruction to execute.

In a similar way to how all the Layer 1 instructions call super.toNumber in their own toNumber methods, every instruction calls super.visit during its own visit method. This allows behavior common to all of a certain type of instruction to be centralized, and gives me just one chance to make a coding mistake.

It turns out that almost all behaviour of any one class is shared throughout some of its peers. For example, consider the class ADDU. The inheritance relationship goes like this:

```
ADDU extends DST extends RType extends NotJType extends R2000Instruction
```

The behaviour that needs to take place during an ADDU is:

1. Read the source register
2. Read the target register
3. Add them together
4. Save the result in the destination register

See how it is only step 3 that is unique to ADDU. All RType instructions are likely to require the values of the registers. All are likely to have to be able to save registers too.

So, for each Layer 3, 1 or 0 instruction, a **visit** method is written, which sets up as many protected variables as may be useful. The most obvious examples of such variables are the values of each register. visit methods are not requied for Layer 2 classes, as they just serve to rearrange constructor arguments.

Overleaf is the entire execution flow for the execution of **ADDU.visit**

```
ADDU.visit:                super.visit (chip);
RType.visit:                   super.visit (chip);
NotJType.visit:                    super.visit (chip);
R2000Instruction.visit:                // return immediately


NotJType.visit:                    rsValue = rs.getValue ();
                                   rtValue = rt.getValue ();


RType.visit:                   rdValue = rd.getValue ();


ADDU.visit:                rdValue = rsValue + rtValue;
                           saveRD (chip);


RType.saveRD:                  rd.setValue (rdValue);
```

Control Flow of ADDU.visit

The method **saveRD** is highlighted just to point out that it is *not* a second call to **visit**.

See how the work is done on the way *back up* the call stack. As execution progresses, the steps taken become more and more specific, until flow returns to the actual **ADDU** class, where the actual addition takes place.

The values rdValue, rsValue and rtValue are all protected. As well as a **visit** method, RType supplies a method **saveRD**, which centralizes saving the destination register. This allows the *actual* register (variable name **rd**) to be made **private**. This in turn means that calling the **saveRD** method is the *only* way an **RType** subclass can save the destination register. This stops any instructions from saving the destination register incorrectly (e.g. accidentally saving the system time, or something equally silly).

Also, see how easy it would be to create the similar **SUBU** instruction (unsigned subtraction). The source code for **ADDU** (and there's not much of it) can be copied almost word for word! The only change required is in the function code and, of course, the replacement of '+' with '-'.

The result is that there are hardly *any* shared actions of more than 1 line of code between *any* two classes in the whole of the **mipsr2000** package.

This has another advantage. If the ADDU instruction was thoroughly tested, then we could almost guarantee that SUBU would work, since we only change one character, and the 'correctness' of the '-' operator can be assumed.

R2000System

The design of the R2000 processor is now complete.

However, a complete system requires two more parts;
A memory system for data,
and an assembler to load programs

Introduction

A working class extending SimulatorSystem must be written for the R2000, otherwise people wanting to use it will not know how it is to be instantiated.

To implement a **SimulatorSystem**, all that is required is that we write a constructor which takes in a **Reader** object, which is the source code of the program to run.

The R2000System class in itself is, therefore, not very interesting at all. It does make use of two other classes which are worth discussion though; **R2000Assembler** and **R2000Memory**

R2000Assembler

This is the assembler for the R2000 implementation. It is arranged as a collection of private methods, each of which deal with a logical amount of program code, like a single line, or a single argument. It makes use of the **InstructionNameInstantiator** class to generate the instructions.

It reads the text through a **MIPSStreamTokenizer**, which is a class extending the **StreamTokenizer** class in the standard libraries. When a **MIPSStreamTokenizer** is called to read the next token from the source, it analyses the text first, to see whether the next token is an immediate number, a register, or a label. If it is any of these, then relevant action is taken. For a label, that means resolving the label into an address using an instance of **LabelStore**. For a register, the actual **Value32** object representing that register is fetched from the processor.

This assembler implementation is not as complete as the assembler on the original SPIM. In the original, there are directives to place data in various formats into the static segment. My assembler does not do this. Also, the SPIM assembler can handle a variety of formats for addresses on load and store instructions. Mine can only handle one: (register) (offset) (e.g. $sp 8).

As a result, my goal of being able to run programs written for the original SPIM is *not* met. however, there is no technical barrier stopping the R2000Assembler class becoming more complete, and becoming compatible with the SPIM assembler. The only thing stopping me from doing it is time. It is important to realize that my project is *not* an assembler; it is a generic simulator. It would have been easy to waste a great deal of effort making my assembler compatible with that on SPIM, only for the rest of the project to suffer (e.g. the user interface).

The assembler is a tidy class whose development could easily be taken up by a maintenance programmer. I estimate that making it fully compatible with the SPIM assembler would make for a complete (if uninspiring) BSc. dissertation.

R2000Memory

This class is an implementation of the **DataMemory** abstract superclass. As instance variables, it holds three other **DataMemory** implementations, to which it delegates calls to read and write depending on the address. These three objects represent the static area, heap and stack used on an R2000 implementation.

These memory areas are shown in the figure below, taken from *Computer Organization & Design*. The heap is marked "Dynamic Data"

file:///Dl/gateway/shared/work/Project/docs/r2000-mem-use.jpg

R2000 Memory Use

Unfortunately, the class needs to know what addresses correspond to each memory area. This meant that a limit needed to be placed on how far the stack can travel down the address space. Addresses below this value are automatically considered part of the heap. Likewise, there must be a definite address separating the static segment from the heap.

The exact value of this limit makes for a 32 KByte stack, and a 32 KByte static segment. This ought to be enough for most student's programs. Indeed, it wouldn't surprise me if the Java runtime ran out of memory long before the stack fills up.

These values are named constants in the source code, and could be changed at the user's desire.

SpimKernel

For this plugin to replace SPIM,
it must include an equivalent set of system calls.

The SpimKernel package supplies this.

Introduction

The SPIM simulator included a small set of system calls that could be used to request services such as I/O and memory allocation. The set of system calls is given below, and is taken from *Computer Organization and Design*

service	**code**	**arguments**	**results**
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string address	
read_int	5		$v0 = int
read_float	6		$f0 = float
read_double	7		$f0 = double
read_string	8		$v0 = string address
sbrk	9	$a0 = bytes required	
exit	10		

The SpimSystem implementation for my simulator must include a kernel with these system calls.

This can be achieved by implementing the **Kernel** abstract class described earlier. To make the system more maintainable, each particular type of service (e.g. 'Output') is placed in a separate class, and a single **Kernel** implementation **SpimKernel** delegates the system calls to these component parts. For example, a single class **Printer** handles the printing of values. The delegation is a simple **switch** statement, a few lines are shown below:

```
switch (callId) {

    case (PRINT_INT) :
        printer.printInt (args [0]);
        return null;

    case (PRINT_STRING) :
        printer.printString (args [0]);
        return null;

    case (MALLOC) :
        return new Integer (memMan.malloc (args [0]));
```

Most of those parts are too simple to warrant discussion here. The memory management system however, is interesting, as it follows a similar algorithm to the malloc call in the C standard library. Indeed, the class **MemoryManager** is based on the description of malloc in the definitive C book *The C Programming Language* (Kernhigan, Ritchie).

The Memory Allocation Algorithm

The heap can be considered to be like a long tape of paper. A block of memory that is available for the user is shown below. The picture is taken from *The C Programming Language*.

file:///DI/gateway/shared/work/Project/docs/malloc

We can see that each block of memory that is made on the heap has two 'secret' fields in front of the address that the user program considers to be the start. When there are many blocks active on the heap, these two fields form a linked list structure of used blocks. Because each block knows its own size, it is possible to calculate the free space between any two blocks. The last block on the heap points to the first, so the memory manager knows when it has searched the entire heap.

When a request comes in for more memory, the list can be searched, and a free space found. Some algorithms try to find the best fit (that is; smallest free space that will do the job) for memory efficiency, while others search for the first fit for sakes of speed. My **MemoryManager** class uses the first fit, for speed and simplicity of the algorithm.

The SPIM kernel does not supply any call to return finished memory to the system[10]. My **MemoryManager** does though. When a call to **free** comes in, the parameter is the start of a block's user space. If the parameter is valid, the block can quickly be found on the linked list structure of blocks, and removed. This removal requires the previous block to adjust its 'Address of next block' field. In a real system, the work would be finished here. In the simulated system, the manager calls the heap to **disable** the memory area, which removes it from the screen, and allows it to be garbage collected.

Since the original SPIM does not support this service, it is not listed in the key of call numbers. **In my SpimSystem, system call 11 provides the *free* function. The $a0 register is set to the start of the space to be freed.**

An unused block at address 0x01000000 supplies the head of the list, and cannot be removed.

10 ... and neither does UNIX. A UNIX process can never become smaller, even if it is a C program and uses the **free** call. In UNIX, **malloc** and **free** are both utility functions of the C programming language; *not* operating system. UNIX itself only supplies the **sbrk** call for more memory.

Testing

Because of its modular design, parts of the Spim system (and indeed the whole simulator) could be tested as they were written.

How Testing was Performed

Typically, after each MIPS instruction was written, small programs were made which implemented that instruction in each of its possible behaviors. For example, the BLEZ (branch if less then or equal to zero) instruction has two behaviors. If its named register is greater than zero, then the instruction does nothing. Otherwise, the branch occurs. Therefore, two small programs would be written to test this instruction; one for each possibility.

The BLEZ is perhaps a bad example, because in actual fact *three* tests would be carried out on that instruction. The first two are the immediately obvious ones; a test would be carried out with values either side of the special separating value of zero. Then, a test would be carried out when the named register actually *was* zero.

This is quite a common approach to take with actions that depend on the value of a number. Inputs safely clear of each limiting value are tested, and when they are passed, inputs actually on each limiting value are tested. For example, a function which took in a percentage integer might be tested first with the value 46 (a completely valid input), then 1405 and -62 (completely invalid), and then 99, 100, 101, -1, 0 & 1; inputs on the limit values.

The Power of the Forte Debugger

Normally, to test every piece of code as it is written would be a huge drain on resources. It is difficult to see where a running program actually *is*, and often special *debug code* has to be inserted to tell the programmers extra information (e.g. "I am in this method.", "The value of *x* is ....").

Due to the strength of the *Forte*[11] debugger, such testing is easy. One can hop straight to a new or troublesome spot in the program, and see exactly what's going on. Primitive values can even be altered while the program runs!

Most errors could be spotted before they even happened, as I tended to read each line before allowing it to be executed. A screen shot of the *Forte* debugger working on the **BLEZ** instruction is shown overleaf. The 'Debugger Window' can be seen, showing local variables. The source editor can also be seen. The red line is a breakpoint, and the blue one is the current position of the program.

11 *Forte for Java* is a development tool I used. It is available free of charge from *Sun*. It includes a source editor which automatically highlights keywords such as for and if. It also includes a debugger which is integrated with the source editor. One can place the cursor at any line of code, and command the debugger to run to that point. When the debugger pauses, all instance variables and local method variables can be viewed.

Though *Forte* does include tools for auto-generating code (for GUI classes), *I did not use them*. The source code they produced was messy.

file:///DI/gateway/shared/work/Project/docs/debugger.jpg

In the picture, we can see the source code for the **visit** method of **BLEZ**. It is only two lines. Importantly, the visit method of BGTZ differs *only* in that the <= operator has been replaced by >. Once the **visit** method of **Branch** (the shared superclass), and the **branch** method (which is also shared) have been tested, BGTZ need not be tested at all. All that is needed is a quick sanity check to ensure that it was correct to use the > operator.

The huge extent to which repeated code has been eliminated from the system means that for any new addition, most of the code has already been written and tested. All that is required is that the new way *in which the existing code* is called makes sense.

Summary

The combined classes of the mipsr2000 and spimSystem packages make for a possible replacement to SPIM, and could easily be developed further.

The fact that the assembler is so simple is a slight limitation, but useful programs can still be written. For example, the following recursive factorial subroutine works perfectly:

```
main:
ADDI $a0 $zero 12
JAL Factorial
BREAK

Factorial:  BLEZ $a0 zeroArg

            NonZero:

            # save the $ra to the stack
            ADDI $sp $sp -4
            SW $ra $sp 4

            # decrement the arg, and JAL
            ADDI $a0 $a0 -1
            JAL Factorial

            # recover the $ra
            LW $ra $sp 4
            ADDI $sp $sp 4

            # get the arg back to its original value
            ADDI $a0 $a0 1

            # multiply the current arg by the previous result, and return
            MULT $a0 $v0
            MFLO $v0
            JR $ra



            zeroArg:
            ADDI $v0 $zero 1
            JR $ra
```

The system looks impressive on screen too. Overleaf is a screenshot of the system executing the program above. The stack can be seen filling up with return addresses.

file:///Dl/gateway/shared/work/Project/docs/running-spim.jpg

System 2: The DM0 System

We have seen the Spim plugin provide the ability to simulate a real processor.

The DM0 System simulates a fictional processor,
but is of real use for students studying compiler design.

It also proves the generic nature of the simulator classes.

Contents

Introduction

By writing a second processor implementation, I will show how my program is extendable. It would be nice to be able to write another implementation of a real processor, but I did not have time. By the time the R2000 was nearing completion, there were only a few weeks left.

There is a processor simple enough for me to write a complete implementation though. In their second year, students at COGS study a compiler design course. The assignment of the course is to write a compiler, from a small C-like language to assembler. The chip that the programs run on is called DM0. The details of DM0 are shown on the next page. *The next page is not my work.* It is the work of, and remains copyright of, Dr. Des Watson of COGS. The page is publicly available at the following URL:

`http://www.cogs.susx.ac.uk/local/teach/langcomp/q00.html`

Requirements

My simulator must be able to run a DM0 implementation without modification. This will give a working example of its extendibility. The DM0 implementation must follow exactly the specification given on the following page.

DM0

DM0 is a stack-based machine, supporting a total of just 9 instructions. It has two distinct internal storage areas. The first is the *stack*, used by the individual instructions as detailed below and the second is an area used for the storage of variables. The effects of each instruction are defined in an algorithmic notation where the variable storage area is represented by the array `v` indexed by the `variable number' (where, for example, the first variable declared has variable number 0, the second has variable number 1 and so on). The stack is represented by the array `s` indexed by the stack pointer `sp`. Initially, `sp` has the value `-1` indicating that the stack is empty, and `sp` is incremented by one each time a value is pushed onto the stack. DM0 is defined as follows:

LDC k

load the integer value `k` onto the top of the stack.

```
sp:=sp+1; s[sp]:=k
```

LDV k

load the value of variable `k` onto the top of the stack.

```
sp:=sp+1; s[sp]:=v[k]
```

STO k

store the value at the top of the stack to variable `k`, decrement the stack pointer.

```
v[k]:=s[sp];sp:=sp-1
```

OUT

output the value at the top of the stack and then decrement the stack pointer.

```
writeln(s[sp]); sp:=sp-1
```

NEG

negate the value at the top of the stack.

```
s[sp]:=-s[sp]
```

ADD

add the top value on the stack to the next-to-top value, decrement the stack pointer and place the result in the location now pointed to by the stack pointer.

```
s[sp-1]:=s[sp-1]+s[sp]; sp:=sp-1
```

SUB

subtract the top value on the stack from the next-to-top value, decrement the stack pointer and place the result in the location now pointed to by the stack pointer.

```
s[sp-1]:=s[sp-1]-s[sp]; sp:=sp-1
```

MUL

multiply the top value on the stack by the next-to-top value, decrement the stack pointer and place the result in the location now pointed to by the stack pointer.

```
s[sp-1]:=s[sp-1]*s[sp]; sp:=sp-1
```

DIV

divide the next-to-top value on the stack by the top value, decrement the stack pointer and place the result in the location now pointed to by the stack pointer.

```
s[sp-1]:=s[sp-1]/s[sp]; sp:=sp-1
```

Memory

On the last page, we see that the fictional DM0 system has two areas of memory. These are referred to as the stack and the store. In the description, it is inferred that both areas of memory start at address zero, and work upwards.

In my simulation, I have only allowed one area of memory for a processor (see **SimulatorSystem** class description). Therefore, I must change the description of the chip slightly. Either the stack or the store must be adjusted so that it starts not at zero, but at some other value.

It makes sense for the stack to start at another value. This is because none of the DM0 instructions are actually concerned with the exact value of the stack pointer, whereas they are concerned with addresses into the store. Existing compilers assume the store starts at zero, but they do not make such assumptions about the stack

Where to Start the Stack

Now that it has been decided the stack shall start at a nonzero value, we must decide exactly where it should start.

In a real computer system, it is important to make full use of all memory available. So, it would make sense for the stack to start right at the top of the address space, at value 0xffffffff, and work *down* the address space as it fills. Then, both the stack and the store would have as much space as possible before any collisions occur. Remember, this is how the R2000 works, with the stack at the top, and the heap at the bottom of the address space.

However, a DM0 program hardly ever uses more than about 10 variables, and never more than about 20 values on the stack, so efficiency is not our concern. Besides, having the stack the 'wrong' way round is just another potential cause for confusion for the second year student, who is already puzzled enough with the workings of his compiler!

Instead, I decided that the stack should start at 0x10000000, and work upwards in the usual manner. This is a good second-best to it starting at zero, as the student can just ignore the first character of the address[1].

Dr Watson confirmed that my method was a sensible way to solve the problem.

1 That is, assuming they stay in hexadecimal mode on the GUI. Unfortunately, the only values that are actually written the same in decimal and hex are the numbers 0-9. I could have chosen to start at a value like 1000000_{decimal}, but a familiarity with hexadecimal notation is something no computer scientist should be without.

The DM0Memory Class

The DM0Memory class extends **DataMemory**, and contains two **DataMemory** objects, representing the stack and store. Calls to read and write are delegated to the relevant object, depending on the address.

This class is similar to **R2000Memory**, and **Heap**, which also pass their calls on to smaller **DataMemory** objects.

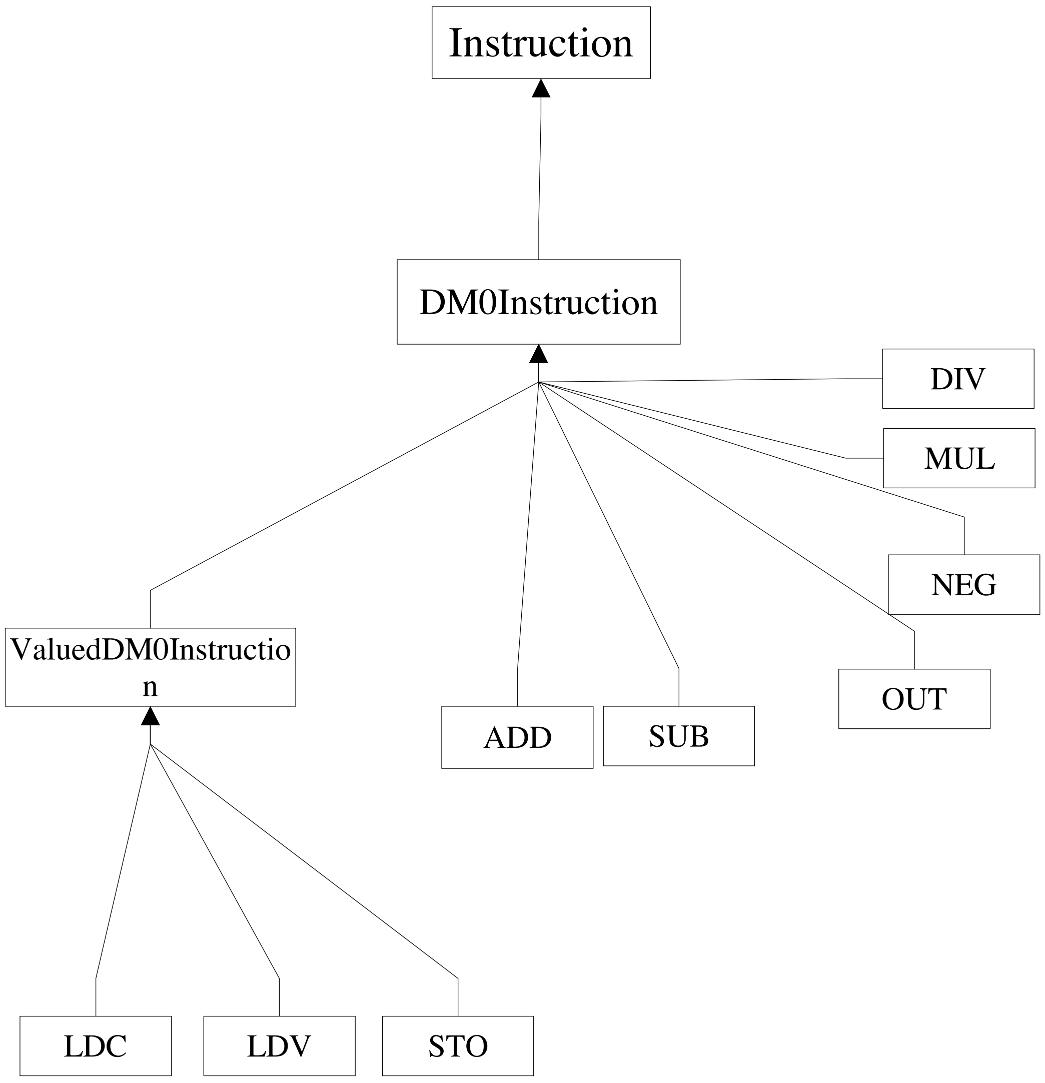
With more time, this similarity could be integrated into an inheritance tree. It was not apparent during the design stage of either one of the two systems (this and Spim).

Instructions

We have also seen how we can name the instruction classes to allow easy parsing of source code. This was demonstrated by the MIPS R2000 implementation. We have also seen from the R2000 how instructions can be used as visitors to the processor, simplifying code. Both these techniques are used in the DM0 implementation.

The instructions can be immediately put into two groups; those that take a numerical argument, and those that do not. If there were more instructions, we might also split the instructions that do have an argument into two further groups; one group for which the argument is an immediate value, and one for which it is an index into the store (eg LDV). As it turns out, there are not enough instructions for this to be a useful task.

So, we obtain the following inheritance tree.



Processor

The **DM0** class extends **Processor**. It so small it is self explanatory. It keeps a program counter and stack pointer as instance variables, which are also displayed on screen. At each call to **tick**, it loads the relevant instruction, executes it by calling **visit**, and increments the program counter.

To centralize stack management away from instruction classes, it supplies **push** and **pop** methods. This in turn allows the stack pointer to be made **private**.

There really is nothing else to it.

Assembler

`(uk.ac.sussex.cogs.peterpi.simulator.dmo.DM0Assembler)`

The DM0 Assembler is very simple. It just reads in lines of code and passes the tokenized values to the **InstructionNameInstantiator**. So, it is not very robust to errors.

This is not too much of a problem though, as the code it reads is likely to be of a high quality, as it has come from a compiler. Students do not need to write their own DM0 assembler by hand. Their only need for my simulation is to check *how* a program works, rather then *if it works at all.*

Kernel

DM0 does not need a kernel, as there are no services it requires. One could argue that the OUT instruction requires the service of printing, but this is so trivial that the processor itself can keep a handle to the stream it outputs to (typically standard out on a unix system). The OUT command does not require any formatting.

System

The class DM0System performs all the startup required by DM0. That is, to run the assembler, and then instantiate the processor. To run a DM0 program would therefore require you to write:

```
java Simulator uk.ac.sussex.cogs.peterpi.simulator.dm0.DM0System <program-file>
```

A shell script would make this easier

Screen Shot

Overleaf is a screenshot of a DM0 system, executing the compiled equivalent of:

```
x = 9;
print ((5 + 3) * 6) / x
```
(The code can be seen in the program window).

file:///Dl/gateway/shared/work/Project/docs/running-dm0.j

A Running DM0 System

We can see that the processor is just about to perform a multiplication on the two stacked values 8 and 6. The store holds the value of 'x' as 9 in address zero. The program counter and stack pointer are visible components of the processor. All of these are **Value32** objects, and so could have their format changed at a single click by the user. This would probably come in handy after the multiplication, as the values would become larger than 10, and the hex difficult to read.

Because the components are small, the virtual desktop (bordered by a Microsoft window) need not take up the entire screen. The image has been cropped, but the virtual desktop occupied only about a quarter of my screen when the screenshot was taken.

Keen readers will spot that this is a development version of DM0Memory; the stack starts at 0x20000000 in the picture, instead of the correct 0x10000000!

Testing

Since DM0 is such a small system, it was possible to run tests that would ensure that every line of the system was executed at least once.

For each of the instructions a program was written which contained just that instruction. Where instructions required 2 values on the stack, these were placed with the LDC or LDV instructions. So, to test the MUL instruction, a program similar to the following would have been run:

```
LDC 5
LDC 8
MUL
```

The program was stepped through, and if the answer was not correct, then the instruction could immediately be dismissed, and the source code checked.

As well as checking the answer, it was also important to check that the stack pointer and program counter were in the correct position. This only had to be done once though, as each instruction called the same **push** and **pop** methods to adjust the stack pointer, and the program counter is only adjusted in one place.

If each class had lots of lines of code, and (more importantly) lots of instance variables, I might have considered also doing *White Box* testing. This involves running the program and checking (normally with the debugger) that each and every value in the program is altered correctly.

This is not required for DM0 as the whole system is so small that the sort of bugs that white box testing usually spots (enigmatic or intermittent ones) have nowhere to hide.

Conclusions

This module for the simulator accurately simulates the DM0 processor. It will help students see how a simple stack based machine can be used to evaluate expressions of arbitrary length. It is fully compatible with programs already written for the chip.

Also, it shows how generic the simulator tool is, and how easily plugins can be written.

Conclusions

A General Simulator

The fact that I could write two plugins, for radically different architectures, shows that I achieved my goal of creating a completely general simulation.

It would be interesting to see how well another person could do in writing a plugin for a different chip. I'd be especially interested in a plugin for a very early processor.

Modular Design aids Future Work

I find it particularly encouraging that for each of the future enhancements listed in the next chapter, I already know exactly where new objects would be inserted into the system, and I can be sure that they won't break any existing parts.

The simulation obvious has a lot of potential for growth yet.

Summary

I am extremely pleased with the end result of this project. It is nearly up to a commercial quality, and really could be used in a teaching environment.

The SPIM Plugin

A simulator equivalent to the original SPIM is made by combining the ***Jukebox*** simulator with the SpimSystem plugin.

We can draw conclusions about this system by recalling its original goals:

"To be able to execute all the instructions of the MIPS processor"

My simulation **does not** implement all instructions. I have not implemented the floating point instructions, nor the floating point registers. *Only time has stopped me from doing so*. Adding FPU functionality would not be difficult, and is described in the 'Future Enhancements' section.

"To support the system calls of the original SPIM"

The **SpimKernel** class, together with the **SYSCALL** R2000 instruction supply exactly the same functionality as the kernel on the original SPIM. The way the memory management works is different, so identical requests for memory on the two systems may result in different addresses being returned. This is not of concern, as one can never guarantee address values in a dynamic memory allocation environment anyway.

"To run programs written for the original SPIM"

My simulator does not do this, as the **R2000Assembler** class is not as complete as (and not compatible with) the assembler built into SPIM. I did not realize at the start of the project how good SPIM's assembler actually is. It supports a great many directives, which appear to be able to be interleaved with each other throughout a piece of source code Writing an assembler of its complexity is almost enough for a BSc. project in itself. I could very easily have wasted a great amount of time making my assembler compatible, at the expense of the rest of the simulation.

As for psuedoinstructions, at first it would appear that they would be impossible to implement, as the assembler is hardcoded to instantiate one instruction per line. However, it is possible, and would not interfere with the rest of the assembler at all. It is discussed in the 'Future Enhancements' section.

"To have a user interface which is at least as good as SPIM's"

I hope the screenshots speak for themselves. My program separates the components of the system, and clearly groups related objects by used of titled borders. Buttons are easily identified, and tool tips help explain what may not be intuitive.

"To be stable"

I cannot see any way to crash my *simulator*, although there are plenty of ways to write an incorrect *user program*. Apart from silly errors found straight after compilation, I have not been able to cause it any upset. The main place where the original SPIM crashed was during memory access. My simulator simply does not care if a memory location is invalid. An exception is thrown, and gracefully handled by the simulator after the relevant method calls have returned. The user's program is considered to have crashed (and rightly so), but the simulation remains running, and the user can examine all values to see what went wrong.

Besides, when Java programs *do* crash, the JVM gives the exact type of the exception (e.g. Null Pointer), and the line number in the source code where it occurred. Often, bugs can be repaired without even having to load the debugger. This is a much more helpful scenario to the unhelpful advice given when a C program crashes:

```
segmentation fault (core dumped)
```

"To run on at least as many platforms as SPIM"

As it is written in Java, the program will run on any system for which there is a Java runtime. At present, this includes my three 'target' operating systems; Windows, Solaris and Linux. Most flavors of unix have Java runtimes.

The program requires the swing library, which is a standard component of Java runtimes versions 1.2 and higher (1.3 is the current release at time of writing). On Solaris, a bug in Java 1.2 means that 1.3 is required.

Since the program has a large run time footprint[1], it is possible that smaller computers will struggle to run it , but it has been tested on a Pentium 166Mhz class computer running Windows, and runs smoothly.

In the paragraph explaining this goal, one sentence reads that "*The user interface should be as similar as possible across platforms*". Because the Java swing library is portable, the program looks **identical** on all architectures.

1 The exact size of which seems to depend on the JVM. On a Pentium II running Linux, it was about 40 Mb. On a Sun 450, it was 68 Mb. Such large footprints are not uncommon in Java programs. It is important to realize that only a fraction (about a quarter) of that size actually seems to be needed after the program has loaded, and the rest gets shifted off to swap space without any degradation of performance.

(This data was collected using the **top** UNIX command)

The DM0 Plugin

The DM0 plugin is important for several reasons:

Firstly, it proves that the simulation truly is general to any architecture, and the the plugin concept works.

Secondly, it shows that the utility classes supplied with the simulation, particularly the **InstructionNameInstantiator**, are general purpose.

Finally, it provides an effetive simulation of the DM0 machine! This will help the second year students studying *Langauges and Compilers* for their degree.

I think this plugin is a definite sucess!

Acknowledgments & References

Acknowledgments

The following people have been of great help during the course of this project.

- Dr. Adrian Thompson: Project Supervisor

  Dr Thompson explained parts of the MIPS system to me, and oversaw the writing of this paper.

- Dr. Des Watson: Lecturer

  Dr Watson gave advice on the DM0 architecture, and answered a lot of questions about the structure of an assembly level program a high level language compiler might generate

- Sue Chatterley: Student

  Sue's project was a study of optimization techniques concerned with reducing the size of a program. Most optimizers put their efforts into increasing the speed of a program. Optimizing for size is a relatively new area of research.

  By making her reference implementations of her ideas run on MIPS assembly programs, she could use my simulator to check execution times. It is a pity that this was only possible very late on in the project.

- Dr. Ian Wakeman: Lecturer

  Dr Wakeman taught the *Computer Systems Architecture* course which inspired this project. I considered using some of his code for my representation of the R2000 processor.

- Dr. Guy McCusker: Lecturer

  Dr McCusker gave advice on the class naming techniques used in the MIPS and DM0 assemblers.

References

The following books were of great help during the course of this project.

- ***Computer Organization & Design*** Patterson & Hennessy

  This is the definitive guide to the MIPS processor, and computer organization in general. Its appendix gives a description of the SPIM simulator and the MIPS instruction set. Literally *every* question I had about computer system design could be answered by this book.

- ***Thinking in C++*** Eckel

  This book was published when object oriented programming was new to most commercial programmers (it had been around for some time in the academic world). It gives excellent descriptions of how the object oriented programming model is made, and how to use it best. It is especially useful for Java programmers, as it helps them see what their language does for them.

- ***Thinking in Java*** Eckel

  Instead of merely listing the Java API, which is available online anyway, *Thinking in Java* explains the *why* questions as well as the *how* ones. It serves as a good introduction to object oriented programming. Also, it gives an introduction to the Swing component set, and makes comparisons between it and the previous AWT component set.

- ***The C Programming Language*** Kernhighan, Ritchie

  For years this book *was* the definition of the C language. It gives an example implementation of the *malloc* and *free* routines, which formed the basis of the **SpimKernel**'s memory management routines.

Appendix: Future Enhancements

Contents

FPU Functionality to the MIPS R2000

An FPU should be added to the MIPS R2000 chip. This would not be difficult to do, but it might be time consuming. One would proceed as follows:

- Make a class extending **RegisterSet**, called something like **FPURegs**. In the constructor, use the **protected** method **makeRegister** to instantiate all the FP registers. These should be placed inside **JPanels** with **TitledBorders** to make it look clear.

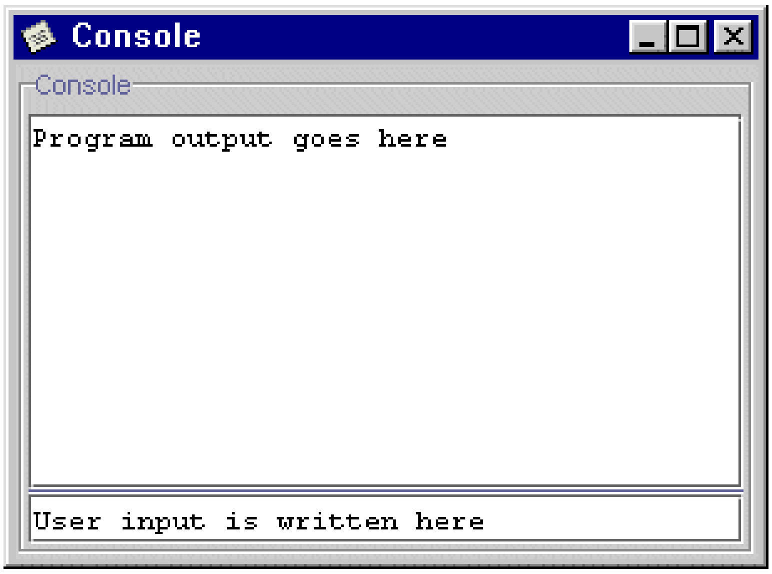
- Integrate the floating point instructions into the existing inheritance tree of instructions. This will probably require a branch quite high up in the tree, as FPU instructions share little with regular instructions.

- Additional methods may need to be placed into the BinaryMath utility class, to convert between the **int** value of a **Value32** object, and the **float** which that **int** *actually* represents.

- An additional display mode could (but does not need to) be added to **Value32**, to allow it to display itself as a floating point number.

A Graphical Console

At the moment, input and output to and from the program is done via the console or DOS box which launched the program. This is not convenient, as the simulation's virtual desktop typically covers that console. A graphical component should be made, looking like the mock one below, from which a **Reader** and **Writer** can be retrieved. The text components themselves would be private, so access to them would *only* be via the **Reader** and **Writer.**



```
Program output goes here
```

```
User input is written here
```

A Future Graphical Console

Text in the output area would be automatically wrapped when it reaches the sides. A vertical scrollbar on the output area would allow users to see previous text.

PseudoInstructions for the MIPS Assembler

PseudoInstruction support could be introduced to the MIPS assembler with hardly any change to current code.

At the moment, the program is read line by line from a **BufferedReader**, which is 'directly' attached to the source file. I propose writing a filter, which would sit in between the **BufferedReader** and the file. Whenever the filter spots a PsuodoInstruction, it would replace it with the complete set of real instructions. It could use the reserved register $at for this purpose.

For example, suppose the filter spotted the multiplication pseudoinstruction, acting on arbitrary registers *x, y* and *z*:

```
mul x y z
```

It would translate that to the equivalent real code:

```
mult x y   # user's line = mul x y z
mflo z     # user's line = mul x y z
```

By including the user's original line as a comment (which would be visible on screen), the filter makes its presence known to the user, who may otherwise be confused by this unexpected tampering.

Some **String** manipulation routines might be required (This class would be 1 line in Perl!), as would a lookup table of known pseudoinstructions and their real equivalents. A pseudoinstruction / real instruction pair could be made into a *class*.

To sit between the file and **BufferedReader**, the filter class would have to extend **Reader**. This is not a problem. The filter would read lines in one at a time, and store them as a String instance variable. Calls to read would be delegated to a **StringReader** instance variable, which reads from the current line.

Instructions as BreakPoints

Breakpoints are instructions which, when executed, cause a debugger to stop and let the user view the current situation before continuing. In a high level language debugger, breakpoints are typically put just before unstable code. The author can run quickly to the breakpoint, then step through the program 1 line at a time.

Already, the simulation lets the user run and step through the program, but it does not supply breakpoints.

Breakpoint code could be introduced to the **InstructionMemory** class. When an instruction on the screen is right-clicked, a popup menu could appear, and the user could set that instruction to be a breakpoint. The **InstructionMemory** could remember the address of the breakpoint instructions in some sort of random access structure (to allow for fast checking of a given address).

When a request to **getInstruction (address)** comes in, the **InstructionMemory** could first check to see if that address is a breakpoint. If it was, then an exception could be thrown, with the detail message "Breakpoint reached at address ..... ". Once the user is happy to continue, they could do so by pressing the 'step' or 'run' buttons.

Some breakpoints could work just once and then be forgotten about, and others could be persistent.

An ASCII mode for Value32

It would be useful for **Value32** objects to be able to display themselves interpreted as a sequence of 4 ASCII charatcters. An ASCII character is 8 bytes, and is a common thing to be stored in memory, especially the static segment. For example, the character 'A' is represented by the value 0x41 (unsigned 65). The 32 bit hex pattern 0x41424344 should be displayed by a **Value32** as "ABCD".

This would not be too difficult. an extra constant; ASCII_DISPLAY would be added to **Value32** (as a parameter for **setMode**). Inside **Value32**.**setMode**, the **switch** statement which discovers the mode would have a new clause added to it. A **private** method would take care of the string manipulation required.

A new menu item would have to be introduced into the **ValueFormatPopup** class. This is not difficult.

Some values do not have corresponding characters, or they have ones which cannot be printed (e.g. 13, carriage return). Some suitable character could denote an unprintable character (perhaps the ?

character).

Trap Handling for the R2000 & Assembler

The R2000 already has some code in place for handling exceptions, such as arithmetic overflow. The assembler should be extended to allow for users to write their own trap handler routines, as part of their assembly code.

Scrollbars for the Memory System

The **ComponentStore** class stores components and displays them on screen. It is a useful class, and normally looks good on screen. However, if there is not enough space on the screen for all the components, they start to overlap and the interface becomes unusable.

The Swing graphical library supplies classes for implementing scrollable areas. I tried to implement this in the **ComponentStore** class, but unpredictable behavior occurred when new components were added to the store. Sometimes, the store would shrink to a tiny size, even when its surrounding window would allow it to grow. I'm not sure if the Swing **JScrollPane** class I was using expects its inside component to change size.

Appendix: Notes

Polymorphism in Java and C++

In some places in the project, we have declared methods to be final, which means they cannot be overridden.

This is in complete contrast to how it works in C++, and shows the fundamental difference of the two languages: They have completely different goals.

In C++, the programmer must explicitly declare that a method *can* be overridden, through use of the **virtual** keyword. In his book *Thinking in Java*, OOP guru Bruce Eckel explains:

*The reason **virtual** exists in C++ is so you can leave it off for a slight increase in efficiency (or, to put it another way, "If you don't use it, you don't pay for it").*

The statement "If you don't use it, you don't pay for it" sums up the C++ philosophy perfectly. The C++ goal (like C) is always efficiency. Unfortunately, a lot of C++ programmers either forget to declare methods as **virtual**, or just don't understand what polymorphism and method overriding are, and see objects as selectively private **struct**s

In Java, the goal is simplicity, readability and complete object orientation. You really need to be familiar with the language to tell it to stop helping you (as we have done with the **final** keyword).

From *C++ for Java programmers*, Timothy Budd:

Bjarne Stroustrup, the developer of C++, has even stated:
"*Within C++, there is a much smaller and cleaner language trying to get out.*"
Some would argue that that smaller, cleaner language is Java.

Though Java may be smaller and cleaner, it is no replacement for C++. To consider it so would be a mistake.