The Algebraic Specification of Multi-core and Multi-threaded Microprocessors

Sean Handley

May 2007

Abstract

In addition to decreasing latency by increasing clock speeds, modern microprocessor architects improve efficiency by employing techniques such as pipelining, multiple cores and multiple threads of execution in order to achieve superior throughput. Such processors can be modelled axiomatically using an algebraic specification language and a reductive term-rewriting system, such as Maude. In this project, I seek to create such models at varying levels of complexity. I begin with a programmer's view of the machine and proceed to develop pipelined and multi-core models.

Project Dissertation submitted to the University of Wales, Swansea
in Partial Fulfilment for the Degree of Bachelor of Science

Department of Computer Science
University of Wales Swansea

Declaration

This work has not previously been accepted in substance for any degree and is not being currently submitted for any degree.

May 10, 2007

Signed:

Statement 1

This dissertation is being submitted in partial fulfilment of the requirements for the degree of a BSc in Computer Science.

May 10, 2007

Signed:

Statement 2

This dissertation is the result of my own independent work/investigation, except where otherwise stated. Other sources are specifically acknowledged by clear cross referencing to author, work, and pages using the bibliography/references. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure of this dissertation and the degree examination as a whole.

May 10, 2007

Signed:

Statement 3

I hereby give consent for my dissertation to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

May 10, 2007

Signed:

This page is intentionally left blank.

Contents

1 Introduction

In addition to decreasing latency by increasing clock speeds, modern microprocessor architects improve efficiency by employing techniques such as *pipelining*, *multiple cores* and *multiple threads of execution* in order to achieve superior throughput. Such processors can be modelled axiomatically using an algebraic specification language and a reductive term-rewriting system, such as Maude.

In this project, it is my goal to use discrete mathematics in order to create algebraic models of microprocessors at varying levels of complexity. These models will then be used as templates for specification modules in Maude. Originally, I intended to produce pipelined, superscalar, multi-core and multi-threaded models. However, over time this proved too ambitious and I decided to focus on producing a simple programmer's view of the machine, a pipelined implementation and a multi-core model.

2 Background and Preliminaries

2.1 Modern Microprocessors

Since the first half of the 20th century, computer architecture has made many advances. Rather than a simple Von Neumann machine, running a single program sequentially until termination, machines of today encorporate many design tricks to speed up the process. *Pipelining* allows each stage of the fetch-decode-execute cycle to occur simultaneously - as one instruction is executed, the next is decoded etc. Multiple CPU cores and *multithreading* opens further avenues of parallelism which help to maximise throughput.[1]

2.1.1 Architecture Principles and Wisdom

Throughout the relatively short history of computer architecture, many important properties of systems have been observed. It is clear that, if a machine performs a certain type of operation more often than any other, it should be more heavily optimised. The common cases should be fast, and the uncommon cases should be correct.

Gene Amdahl formulated a specific law to model optimisation of a computer system. Amdahl's law can be summarised with the following formula:

$$\frac{1}{\sum_{k=0}^{n}(\frac{P_k}{S_k})}$$

where P_k is a percentage of the instructions which can be made faster/slower, S_k is a multiplier representing the speed change (where $1 =$ no change), k is a label for percentage and speedup, and n is the number of resulting increases/decreases in speed.

This sets the scene for improving the speed of CPU architecture as it shows that, as parts of a system are optimised, they will eventually be dominated by other suboptimal parts. This idea becomes increasingly important as parallelism is exploited in pipelined and multi-core systems.

2.1.2 Latency and Throughput

There are, typically, two methods to improve the efficiency of a system: decrease the system's latency, or increase the system's throughput. In processor design, this translates to clock speeds and the system's capacity for parallelism. Over time, clock speeds have increased substantially, decreasing the time it takes for a machine to execute a single instruction. Parallelism increases throughput because it allows multiple tasks to be executed simultaneously. Parallelism will be discussed in more depth later.

2.1.3 RISC vs CISC

RISC (Reduced Instruction Set Computer) is a design philosophy for microprocessors which was pioneered as a result of several emerging pressures on chip designers.

In the 1950s and 1960s, the majority of computer programs were written using machine code or assembler. This meant that instruction set designs were relatively complex, so as to make programmers' lives easier i.e. instructions which read in the values, calculated a result and stored them back in a single instruction. There were often several addressing modes and many systems had native support for polynomial and complex number data types. Such machines were later named *CISCs (Complex Instruction Set Compters)* in antithesis to their RISC counterparts.

Through the 1970s and 1980s, the use of compilers for high level languages became increasingly popular due to the advantages high level programming gave to the software community. Brevity and readability of programs meant increased productivity and easier debugging. However, a study into compiler generated machine code revealed that the compilers were ignoring many of the more complex instructions in favour of the smaller, simpler instructions. These instructions were being used in combination to achieve the same effect as the larger instructions and, in many cases, they would execute faster, too.

Another factor against CISC was the disparity between CPU speeds and memory speeds. It was becoming increasingly clear that the former were going to increase at a far higher rate than the latter. As a result, a machine's overall speed was to be limited by slow memory and so architectures had to be redesigned so as to minimise memory accesses. This led to the necessity of CPU caches and an increased set of registers.

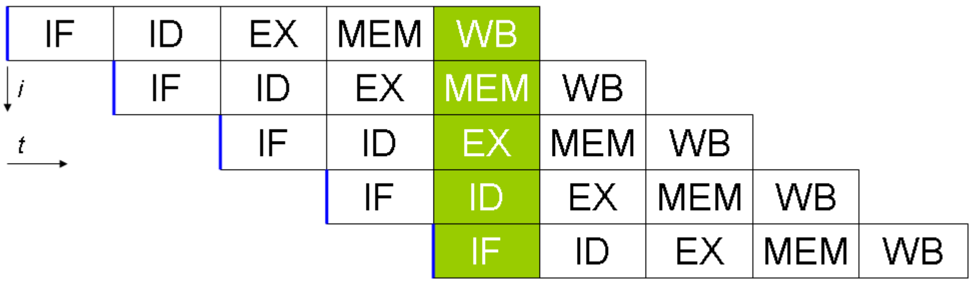
From these needs, the RISC movement emerged. RISC instructions were small with uniform execution times and single addressing modes. Also, RISC machines used a concept known as *load-store* whereby values needed for operations were loaded from memory into the CPU registers, operated on directly whilst there, and then written back to memory later. Usually, only integer (and perhaps floating point) data types were supported and the instruction sets were small, clean and orthogonal. This allowed for the development of *pipelining* which will be discussed shortly.

2.1.4 Parallelism

Typically, when discussing parallelism in a system, we talk about *Instruction Level Parallelism (ILP)* and *Thread Level Parallelism (TLP)*. The former involves having multiple instructions at different stages of execution at a given time. A pipelined processor is one which exploits ILP. The latter is where two independant instruction streams (known as threads) are executed simultaneously. The two methods of exploiting TLP which I will be discussing are *multiple CPU cores* and *multi-threading*.

2.1.5 Pipelining

Pipelining is an idea which has been around since the first half of the twentieth century. In a simple, non-pipelined system, instructions are processed in a sequential series where each section of the cycle must complete before another begins i.e. Fetch, Decode, Execute, Fetch, Decode, Execute,... A pipelined processor takes advantage of instruction-level parallelism and overlaps each stage as shown below.



KEY: *IF=Instruction Fetch, ID=Instruction Decode, EX=Execute, MEM=Memory Access, WB=Write Back to registers*[14]

This introduces many new *hazards*[1]. i.e. If a dependency exists between two instructions,

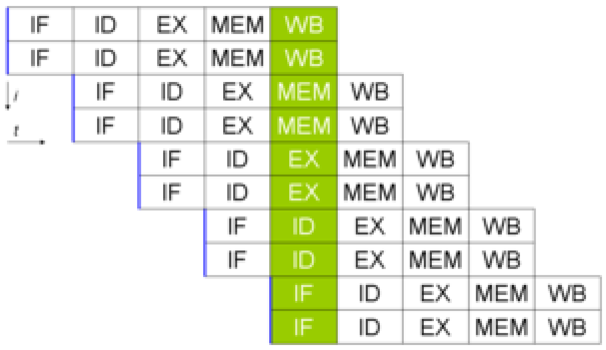
```
Add the value of R1 to the value of R2 and store the result in R3
Subtract the value of R4 from R3 and store the result in R5
```

The first instruction must be allowed to complete its execution before the second, since the second depends on the result of the first. Similar issues occur with branching instructions, where the result of a branch condition is not known. There exist many sophisticated branch-predicting strategies, but for the purposes of this project, I shall simply assume that all branches are taken.

2.1.6 Superscalar

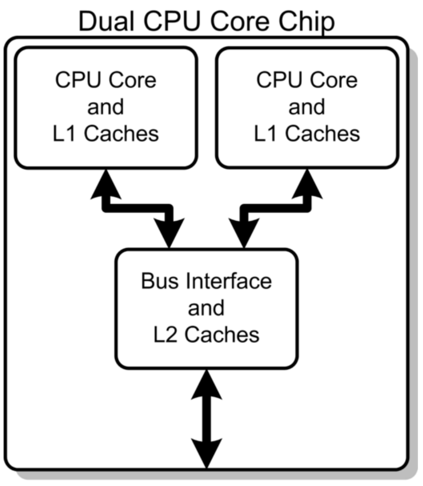
A *superscalar* processor is one which makes use of multiple pipelines in order to achieve superior throughput. In practice, this is achieved by duplication of several hardware components. As with pipelining, the use of superscalar processing introduces a number of complications.

[1]To borrow a term from electronics.



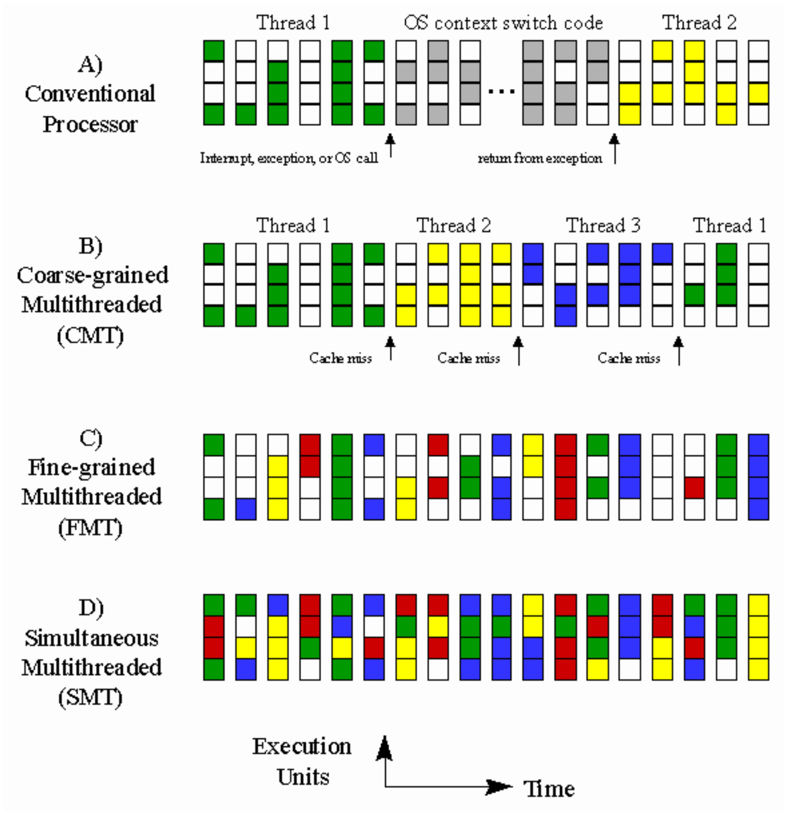
2.1.7 Multi-core

A *multi-core* microprocessor contains multiple processor cores which share some components, such as level two caches and the bus interface.



2.1.8 Multi-threading

Simultaneous Multi-Threaded microprocessors are single processors which behave as if they were multiple processors. An operating system will have access to these virtual processors and allocate tasks to them in parallel. In practice, this means that duplicated hardware introduced to aid superscalar processing is used when idle to run multiple threads of execution. Since threads are, mostly, independent of one another, this is relatively straightforward.


Thread 1
OS context switch code
Thread 2
A) Conventional Processor
Interrupt, exception, or OS call
return from exception
Thread 1
Thread 2
Thread 3
Thread 1
B) Coarse-grained Multithreaded (CMT)
Cache miss
Cache miss
Cache miss
C) Fine-grained Multithreaded (FMT)
D) Simultaneous Multithreaded (SMT)
Execution Units
Time

2.2 Algebraic Specification

Specifying a system in terms of an *algebra* is an elegant way to describe it. Using *interfaces* and *axioms*, rather than a series of imperative procedures, one achieves a mathematically sound representation, allowing for verification and proof of correctness. A system that can be verified is one which can be relied upon to perform its tasks predictably and safely. This is good news for manufacturers of aeroplanes, nuclear power plants and hospital equipment.

Both software and hardware may be modelled using an algebraic specification. In the case of this project, a working model of a microprocessor[2] may be constructed at any level of abstraction without actually making the machine itself, allowing easier exploration of concepts and ideas.

Dr. Harman and Prof. Tucker have produced several papers[2][3][6][7] which outline ways to use algebra to represent microprocessors.

2.2.1 Clocks

A *clock* is an algebra $(T|0, t+1)$ where $T = \{0, 1, ...\}$. Beginning at zero and using t+1 to refer to further clock cycles, time can be represented as a set of

[2]Albeit, one of a much higher level of abstraction than a realistic model of electronic components.

explicit, synchronous, discrete values where each clock cycle denotes an interval of time and time is defined by the occurence of events[3].

2.2.2 Streams

A stream, $s \in [T \to A]$, is a function from a clock T to a set A of data items. This gives us a formal representation of time-separated items.

$s \in [T \to A]$ is a stream.
$t \in T$ is a clock cycle.
$s(t)$ represents the data item on the stream at time t.

2.2.3 Iterated Maps

A microprocessor's behaviour can be described using an *iterated mapping function*:

$$F : T \times A \to A$$

which iterates as follows:

for $t \in T$, $a \in A$
$F(0, a) = a$
$F(t + 1, a) = f(F(t, a))$

Therefore $F(t, a) = f^t(a)$

This gives the following state-trace:

$$a, f(a), f^2(a), ..., f^t(a), ...$$

2.2.4 Decomposition

The iterated map function may be re-written to illustrate the dependencies between components in a system:

$$F_1(0, a_1, ..., a_n = a_1,$$

$$(...)$$

$$F_n(0, a_1, ..., a_n = a_n,$$

$$F_1(t + 1, a_1, ..., a_n) = f_1(F_1(t, a_1, ..., a_n), ..., F_n(t, a_1, ..., a_n)),$$

$$(...)$$

$$F_n(t + 1, a_1, ..., a_n) = f_n(F_1(T, a_1, ..., a_n), ..., F_n(t, a_1, ..., a_n)).$$

[3]This is an important concept. Time is not an external factor, independent of the microprocessor model. An interval is more usefully defined by the occurence of an event which is considered interesting. In this case, we will define such an event as a change of *state*.

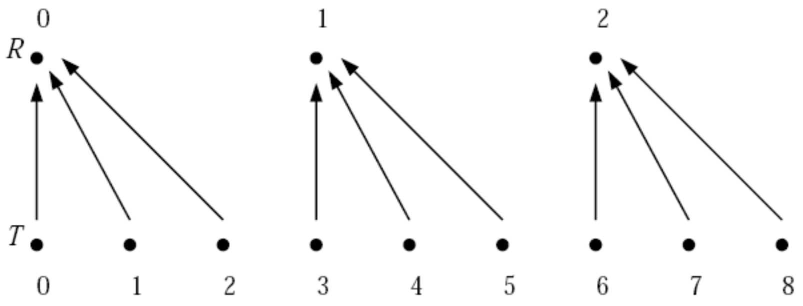
2.2.5 Comparing Iterated Maps

Iterated maps may be compared using the following commutative diagram:

$$\begin{tikzcd}
T \times A \arrow[r, "F"] & A \\
S \times B \arrow[u, "(\lambda, \psi)"] \arrow[r, "G"] & B \arrow[u, "\psi"]
\end{tikzcd}$$

2.2.6 Retimings

A microprocessor specification may contain multiple clocks of different speeds and, also, clocks may be irregular. Because of this, it is necessary to develop methods of mapping clocks to each other. This is know as a *retiming function.* A retiming $\lambda : T \to R$ is a surjective, monotonic map between a clock T and a clock R. The set of retimings from T to R is denoted with $Ret(T, R)$. The concept is illustrated visually below.



Every cycle of clock R corresponds with the same number $k \in N^+$ cycles of clock T. Clock T runs at exactly k times the rate of clock R, and $\lambda(t) = \lfloor t/k \rfloor$. We say such retimings are *linear* of length k.

For each retiming λ there is a corresponding *immersion* $\bar{\lambda} : R \to T$,

$$\bar{\lambda}(r) = (\mu t \in T)[\lambda(t) = r].$$

2.3 Maude

Maude is a *specification language* which allows a system to be modelled algebraically[11]. By *reductive term re-writing*, a module comprised of sorts, interfaces and axioms can be "executed" to produce an output. Modules are put together in such a way:

```
fmod BASIC-NAT is
   sort Nat .

   op 0 :  -> Nat .
   op s :  Nat -> Nat .
   op _+_ :  Nat Nat -> Nat .
```

```
    vars N M : Nat .
    eq 0 + N = N . *** Axiom 1
    eq s(M) + N = s(M + N) . *** Axiom 2
  endfm
```

This module presents a simple number system, represented by the constant 0 and the successor operation which gives us the next number. If we wish to perform a simple addition using this system (e.g. 2 + 3), we give Maude the `reduce` command.

```
  reduce s(s(0)) + s(s(s(0))) .
```

Maude responds with:

```
  rewrites:  3 in 0ms cpu (0ms real) (  rewrites/second)
  result Nat:  s(s(s(s(s(0)))))
```

Maude used the two addition axioms to re-write the terms:

```
  s(s(0)) + s(s(s(0))) = s(0) + s(s(s(s(0)))) [Axiom 2]
  s(0) + s(s(s(s(0)))) = 0 + s(s(s(s(s(0))))) [Axiom 2]
  0 + s(s(s(s(s(0))))) = s(s(s(s(s(0))))) [Axiom 1]
```

2.4 Relevant Literature

- *"Computer Architecture: A Quantitative Approach", Third Edition by John L. Hennessy, David A. Patterson.* A classic text in the field of computer architecture, H&P covers a wide variety of topics in great depth. General architectural concepts such as instruction sets, pipelining, branch prediction, superscalar processors and instruction level parallelism are documented thoroughly. However, the book does not seem to contain a great deal of information regarding multi-threading, therefore alternative sources of information on this topic must be sought.

- *"Algebraic Models of Microprocessors: Architecture and Organisation" by N. A. Harman and J. V. Tucker, Acta Informatica 33 (1996).* This is an early paper on the topic of modelling microprocessors algebraically. It contains a lengthy section of algebraic "tools" which will form the basis of my models.

- *"Correctness and Verication of Hardware Systems Using Maude" by N. A. Harman, Swansea University.* This paper discusses representing hardware systems algebraically using Maude, making it a useful link between the algebraic theory and creating Maude modules which put the theory into practice.

- *"High Performance Microprocessors" by N.A. Harman and A. Gimblett, Swansea University, 2006.* These are the course notes for the high performance microprocessors course, as taught at Swansea University. They discuss a broad selection of topics in the field, providing the reader with a firm grounding in computer architecture.

- *"System Specification" by N. A. Harman and M. Seisenberger, Swansea University, 2006.* These are the course notes for the system specification course, as taught at Swansea University. They discuss the specification of software and hardware using the algebraic specification language, Maude. As well as providing a background for specification in general, they describe algebras, sorts, axioms and other relevant topics and proceed to outline useful examples of microprocessor specification. The coursework submitted for this module last year will provide a basis for the initial programmer's model.

- *"Algebraic Models of Simultaneous Multi-Threaded and Multi-Core Microprocessors" by N.A. Harman, Swansea University, 2006.* Highly relevant material concerning details of how multi-core and multi-threaded processor function and how this functionality is modelled algebraically.

- *"Algebraic Models of Computers" by N.A. Harman, Swansea University, 2006.* Neal Harman's partially completed book on the field. It covers all related topics in detail.

- *A selection of papers published on Intel.com.* The world leading Intel Corporation publishes a helpful selection of white papers and demonstrations on their public FTP server. These cover some general topics, such as multi-cores and multi-threading, to a moderate level of detail and often include animated demonstrations to aid understanding of potentially complex interactions. Other papers chronicle individual architectures and innovations for which Intel is directly responsible, such as NetBurst and the Core Duo. The papers used will be included on the project's accompanying CD-ROM.

3 Specifying Basic Microprocessors

Before producing implementations which use parallelism, I shall discuss how a simple, sequential processor can be modelled which matches the programmer's perspective i.e. each instruction completes before the next is executed, therefore data dependencies do not interfere with results.

3.1 The binary number system

The heart of the microprocessor logic in these models is contained within a module which defines the binary number system and the operations which may be performed on binary values. Whenever an instruction is executed, it uses an operation which has been defined here over the natural numbers, as implemented in base two.

Maude provides an ideal mechanism for specifying the binary number system:

```
sorts Bit Bits .

subsort Bit < Bits .

ops 0 1 : -> Bit .
op __ : Bits Bits -> Bits [assoc prec 1 gather (e E)] .
```

We begin by defining the sorts *Bit* and *Bits* and declaring the former to be a subsort of the latter. The constants, zero and one, are defined as being of sort Bit. The __ operation is a recursive definition of a binary sequence of any length, defined as some Bits appended to some Bits to give some Bits. Since a Bit is a subsort of Bits, a sequences is recursively evaluated until each part is recognised as a separate Bit - either the constant one or the constant zero. In addition to this, some information describing the behaviour of such a binary sequence is given in square brackets. Binary sequences are declared to be associative, evaluated from right-to-left and given order of precedence of 1, the highest precedence in the system. This means that binary sequences are evaluated using the __ operation before any other operators acting on the sequences[4].

Now that a simple, rescursive concept of binary numbers has been established, axioms which act upon these numbers can be defined. The instruction set for this simple, RISC model involves only a few logical and mathematical operators, none of which return any negative numbers. Below, the axioms defining the logical operators *and*, *or* and *not* are defined.

```
op _and_ : Bits Bits -> Bits [assoc comm prec 2 gather (E e)] .
op _or_ : Bits Bits -> Bits [assoc comm prec 2 gather (E e)] .
op _sl_ : Bits Bits -> Bits [prec 2 gather (E e)] .
```

[4]In Maude, the operator __, defined by two underscores, represents the placement of the function inputs. Infix operators are typically defined by _ op _. Since there is no operation symbol, the inputs are simply defined as single *Bit*s, separated by spaces i.e. 0 1 0 1 1 0 ...

```
vars S T : Bits .
vars B C : Bit .

eq not (S T) = (not S) (not T) .
eq not 0 = 1 .
eq not 1 = 0 .

eq B and 0 = 0 .
eq B and 1 = B .
eq (S B) and (T C) = (S and T) (B and C) .

eq B or 0 = B .
eq B or 1 = 1 .
eq (S B) or (T C) = (S or T) (B or C) .
```

3.2 Machine Words and Instruction Formats

In this 32-bit microprocessor model, all instructions have uniform length and format. A single instruction is exactly one 32-bit word, the first byte defines the op code[5] and the second, third and fourth bytes define operands. The notion of 32-bits being equal to a word and 8 bits being equal to a byte is easily conveyed using Maude's built-in *mb()* operation.

```
vars B1 B2 B3 B4 B5 B6 B7 B8 : Bit .
vars B9 B10 B11 B12 B13 B14 B15 B16 : Bit .
vars B17 B18 B19 B20 B21 B22 B23 B24 : Bit .
vars B25 B26 B27 B28 B29 B30 B31 B32 : Bit .

mb (B1 B2 B3 B4 B5 B6 B7 B8) : Byte .

mb (B1 B2 B3 B4 B5 B6 B7 B8
    B9 B10 B11 B12 B13 B14 B15 B16
    B17 B18 B19 B20 B21 B22 B23 B24
    B25 B26 B27 B28 B29 B30 B31 B32) : Word .
```

Now the Maude parser will recognise 32-bit strings as words and 8-bit strings as bytes. However, for a given 32-bit word, it is necessary to define a set of functions which extract the op code and the three operands. This requires the use of a function, *bits*, which takes a bit sequence and two index values and returns the subsequence defined by the indices.

```
op bits : Bits Int Int -> Bits .
op opcode : Word -> OpField .
ops rega regb regc : Word -> OpField .
```

[5]A number which statically represents an operation.

```
var S : Bits .
var B : Bit .
vars I J : Int .
var W : Word .

eq bits(S B,0,0) = B .
eq bits(B,J,0) = B .
ceq bits(S B,J,0) = bits(S, J - 1,0) B if J > 0 .
ceq bits(S B,J,I) = bits(S,J - 1,I - 1) if I > 0 and J > 0 .

eq opcode(W) = bits(W,31,24) .
eq rega(W) = bits(W,23,16) .
eq regb(W) = bits(W,15,8) .
eq regc(W) = bits(W,7,0) .
```

This allows instructions to be stored as 32-bit strings and further acted upon by the microprocessor. It should be noted that all the mathematical and logical operations need to be performed both upon bytes (adding register addresses together, for example) as well as on full words in memory. For this reason, operations are defined twice - once for bytes, once for words.

3.3 Memory and Registers

Since this is a 32-bit system, the number of addressable memory locations is 2^{32}. As registers are addressed using a single byte, this gives 2^8 general purpose registers.

Memory and registers are modelled in Maude in exactly the same manner. Since Maude is not an imperative language, it is not possible to assign a value to a variable. Since this is precisely what we wish to do with our memory and registers, it is necessary to represent storage and retrieval in a different manner.

```
sort Mem .

op _[_] : Mem Word -> Word .
op _[_/_] : Mem Word Word -> Mem .

var M : Mem .

var A B : Word .
var W : Word .

eq M[W / A][A] = W .
eq M[W / A][B] = M[B] [owise] .
```

The _[_] operator defines a "read" operation which takes a memory location as input and returns the value at that location, if one exists. The _[_/_] operator "writes" the given value to the given memory location. Obviously, there is no actual reading or writing occurring in a traditional programming sense. The axioms defining the behaviour of the memory read and write in a more mathematical sense. Instead of a variable being replaced, new information is just added to the existing structure. The first axiom states that, if a write has given W as the value at address A and the contents of A are requested, W is yielded. The second axiom is marked by *[owise]*. This is shorthand in Maude for *"apply this axiom if none of the others may be applied"*. If B is given as the index, and no corresponding W value exists, the operation yields M[B], a recursive call to search other entries. This defines the read behaviour. The write behaviour is defined constructively within the two read axioms.

3.4 Instruction sets

Instruction sets are defined very simply with a set of constants.

```
ops ADD32 MULT AND OR NOT : -> OpField .
ops SLL LD32 ST32 EQ GT JMP : -> OpField .

eq ADD  = 0 0 0 0 0 0 0 0 .
eq MULT = 0 0 0 0 0 0 1 0 .
eq AND  = 0 0 0 0 0 0 1 1 .
eq OR   = 0 0 0 0 0 1 0 0 .
eq NOT  = 0 0 0 0 0 1 0 1 .
eq SLL  = 0 0 0 0 0 1 1 0 .
eq LD   = 0 0 0 0 0 1 1 1 .
eq ST   = 0 0 0 0 1 0 0 0 .
eq EQ   = 0 0 0 0 1 0 0 1 .
eq GT   = 0 0 0 0 1 0 1 0 .
eq JMP  = 0 0 0 0 1 0 1 1 .
```

As this is a load-store model, *LD* and *ST* are the operations to read from and write to memory. Also defined are the arithmetic operators, *ADD*,*MULT* and *SLL*[6]; the logical operators *AND*, *OR* and *NOT*; the comparison operators *EQ* and *GT*; and the branch instruction *JMP*. For the purposes of simple algebraic modelling, this instruction set provides a reasonable amount of functionality without introducing unnecessary complications.

3.5 Modelling Progress With Streams and an Iterated Map

Iterated maps are used in these models to show change of state of time. These are modelled in Maude in the following manner.

```
op spm : Int SPMstate -> SPMstate .
op next : SPMstate -> SPMstate .
```

[6]Shift left.

```
var SPM : SPMstate .
var T : Int .

eq spm(0,SPM) = SPM .
eq spm(T,SPM) = next(spm(T - 1,SPM)) [owise] .
```

The *spm* operation shows the state of the microprocessor at a given time using the next-state function. As shown earlier, the algebraic representation for an iterated map is as follows.

for $t \in T$, $a \in A$
$F(0, a) = a$
$F(t + 1, a) = f(F(t, a))$

Due to the way in which Maude evaluates terms, it is necessary for us to interpret this as

for $t \in T$, $a \in A$
$F(0, a) = a$
$F(t, a) = f(F(t - 1, a))$

This gives exactly the same behaviour, as expected, but must be written as such to account for a subtlety of the language.

The next-state function is then laid out in the form of conditional equations, each dependant on the opcode of the current instruction being executed.

```
ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
    REG[REG[rega(MP[PC])] + REG[regb(MP[PC])] / regc(MP[PC])])
    if opcode(MP[PC]) == ADD .
```

The terms *MP*, *MD*, *PC*, and *REG* represent program memory, data memory, program counter and registers. The separation of program memory and data memory is somewhat artificial, since the Von Neumann architecture of modern machines uses the stored program concept whereby programs and data occupy the same memory. However, at a lower level, instructions and data do inhabit separate caches, therefore the separation isn't all that unrealistic and can be justified by the overall simplicity of the memory model and the reduction in complexity that a division provides.

3.6 Running Test Programs

Test programs are written for the micrprocessor model in a manner similar to assembler language. Instruction words are carefully written by hand with each of the four bytes separated by a line and commented to make for easier reading.

```
    ops Md Mp : -> Mem .
    op Rg : -> Reg .
    op Pc : -> Word .

    *** Set the PC to zero
    eq Pc = 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 .

    *** R1 = 1
    eq Rg[0 0 0 0 0 0 0 1] =
            0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 1 .

    *** R2 = 3
    eq Rg[0 0 0 0 0 0 1 0] =
            0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0
            0 0 0 0 0 0 1 1 .

    *** INST 1 [ Add RG1 and RG2 and store in RG6 ] -> RG[6] = 4
    eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] =
            0 0 0 0 0 0 0 0 *** ADD
            0 0 0 0 0 0 0 1 *** RG1
            0 0 0 0 0 0 1 0 *** RG2
            0 0 0 0 0 1 1 0 . *** RG6

    *** INST 2 [ Store RG6 in Mem[RG1+RG2] ] -> Mem[4] = 4
    eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0] =
            0 0 0 0 1 0 0 0 *** STORE
            0 0 0 0 0 0 0 1 *** RG1
            0 0 0 0 0 0 1 0 *** RG2
            0 0 0 0 0 1 1 0 . *** RG6
```

This section of code adds the contents of registers 1 and 2 and then stores the result in the address indexed by that result.Once a program has been written in this style, the state of the microprocessor over time is revealed using Maude's *reduce* command.

```
reduce spm(1, (Mp,Md,Pc,Rg)) .
reduce spm(2, (Mp,Md,Pc,Rg)) .
.
.
.
```

This gives a state trace in the following format:

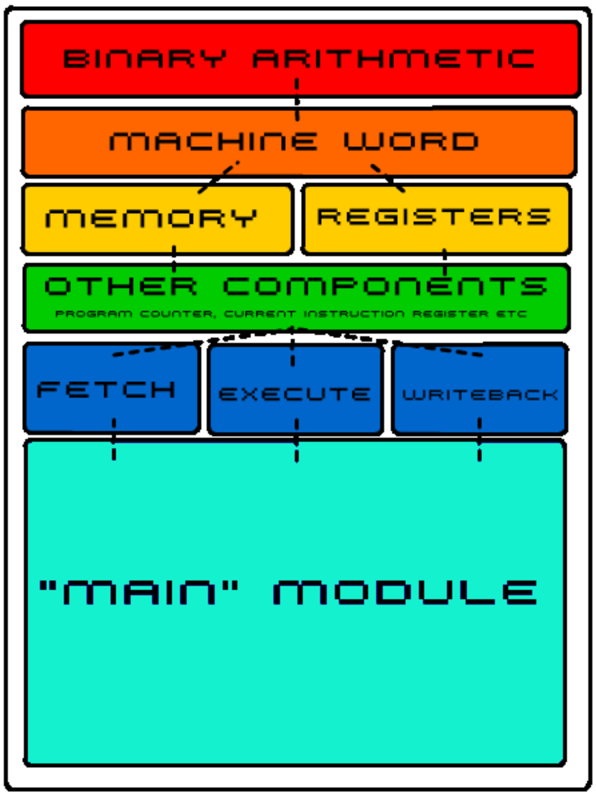
```
result SPMstate: Mp,Md,0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0,
(Rg[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 / 0 0 0 0 0 0 0 1]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 / 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 / 0 0 0 0 0 1 1 0])
result SPMstate: Mp,
Md[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 /
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0],
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0,
(Rg[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 / 0 0 0 0 0 0 0 1]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 / 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 / 0 0 0 0 0 1 1 0])
```

4 Specifying Complex Microprocessors

Specifying microprocessors which take advantage of instruction-level parallelism and thread-level parallelism requires a different approach to the simple, sequential models. This involves the use of mutiple iterated maps and functions which coordinate them.

4.1 Pipelined Processors

The pipelined implementation uses three iterated maps (one for each of the functional units), all of which are coordinated by a "main" iterated map. An overview of the module structure is given below.



4.1.1 Pipeline Hazards

A hazard, in terms of microprocessor architecture, is defined as a set of circumstances which facilitate an action that leads to an incorrect state or a damaging performance penalty. In a sequential machine, one instruction may not be executed until another has finished. This is inefficient but it is non-hazardous. A pipelined microprocessor, such as this, has several instructions in the processor simultaneously, each at different stages of execution. This introduces a level of instruction level parallelism which introduces the possibility of hazards in three distinct ways:

- *Resource Conflicts*: Two instructions both require access to the same component at the same time.
- *True*[7] *Data Dependency*: An instruction takes as one if its operands the result of a previous instruction which has not yet completed.
- *Branching*: A branch instruction sets the program counter to a new address which is not the next sequentially. This means a new address must be calculated, often pending the result of a conditional branch instruction. Naturally, this causes a performance penalty as the program counter must be updated to show the next instruction address. Before this is calculated, the Fetch Unit must be stalled as simply fetching the next sequential instruction will result in the wrong instruction being executed.

Resource conflicts have no other solution than to stall for a cycle, unless the particular resource is duplicable. In this simulation, the pipeline is small and simple, therefore such conflicts aren't a threat as each stage is mutually exclusive in the resources it will require i.e. The Fetch Unit will only need to access program memory and the current instruction register; The Execute Unit will only need to read from the registers; and the Writeback Unit will need to write to data memory, the program counter and the registers.

This does pose a form of resource conflict with respect to the registers and this is the Read After Write hazard. Such hazards can be removed by the compiler, often by placing non-related instructions between the two offending ones. However, this cannot be relied upon and RAW hazards are generally unavoidable. Their dependency is known as a true dependency because the second instruction is unable to proceed without the result of the first. It's not a simple name dependency, but a dependency on the computed value of the instruction. In such circumstances, there is little choice but to stall the pipeline. The easiest way to implement a safeguard against this hazard in my 3 stage pipeline is to keep a record of the previous instruction, a Previous Instruction Register. The result register given in this instruction can be checked against the operand registers of the current instruction. If a Read After Write hazard is detected, the fetch unit is stalled by setting a "stall" flag, a simple boolean value which the Fetch Unit checks before fetching the next instruction. If it is set, the Fetch Unit unsets it and does nothing for that cycle.

Branching strategies are a vital part of any efficient pipelined processor. A branch target buffer holds a number of previously computed branch targets which can be used directly without recalculation. In conditional branches, however, it isn't known if a branch will be taken or not. This means a level of prediction is necessary. A 2-bit branch predictor records whether the last 2 executions of a particular branch were taken or not. This make prediction easier, since loop conditions often evaluate true many times in sequence. Results predicted in such a manner must be treated as speculative, however, and recalculated if the prediction is later proven wrong. In this simulation, however, it is

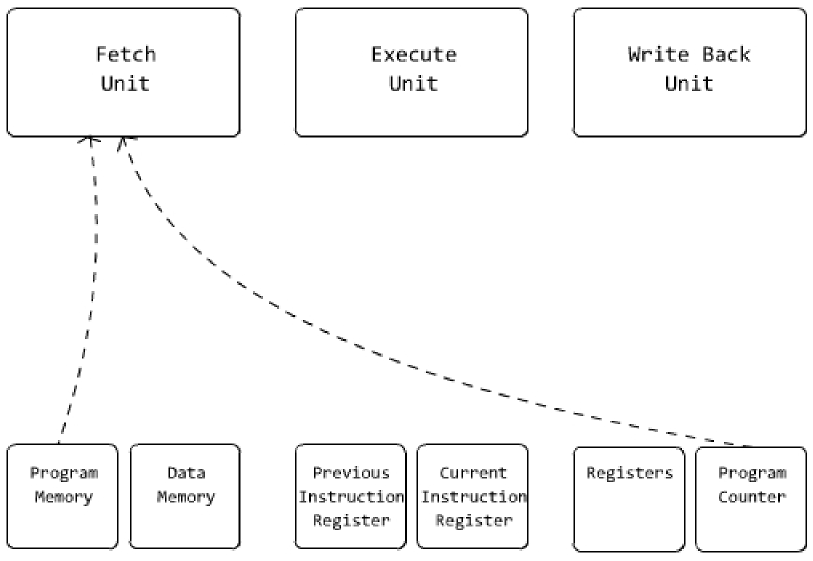
[7]There are three generally recognised sorts of data hazard. *Read After Write*, also known as *True Data Dependency*, is the sort we shall concern ourselves with in a pipelined processor. The others, *Write After Read* and *Write After Write*, become problems in superscalar machines due to the issues arising from out of order execution.

easier just to have the Execution Unit manually change the current instruction register and program counter as soon as it detects that a branch has been taken. This is known as *bypassing* or *forwarding*.

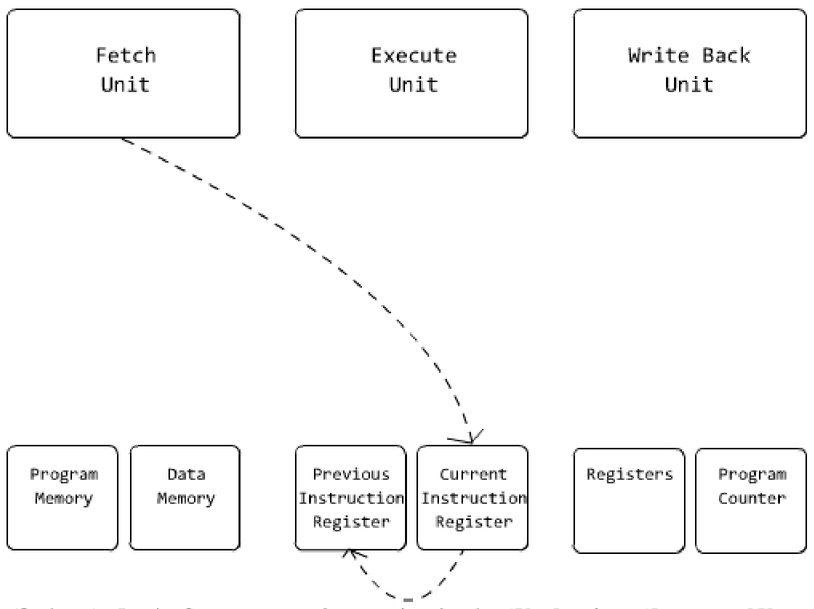
In this implementation, we assume a MIPS style pipeline (See Appendix) i.e. one without interlocks, therefore resolution of data dependencies is given over to the compiler and all incoming code should (hopefully) be free of data dependencies.

4.1.2 Abstract Model

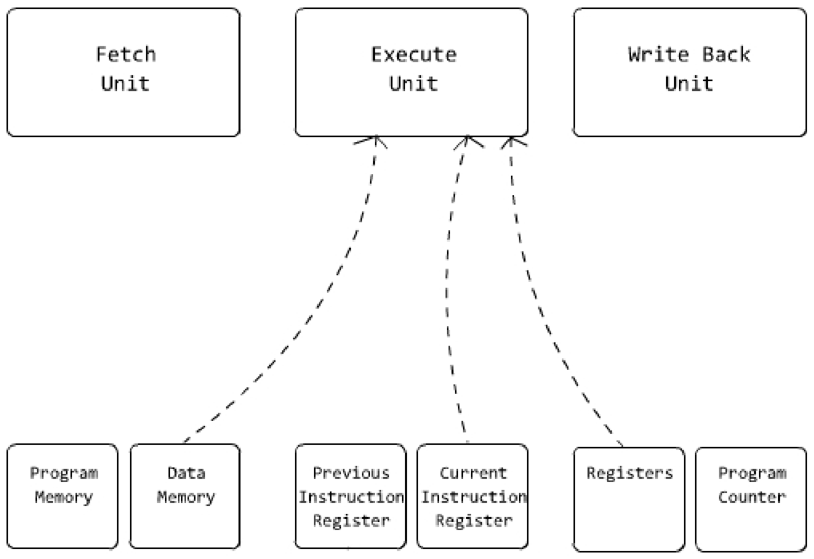
Here is a diagramatic representation of the pipeline implementation.



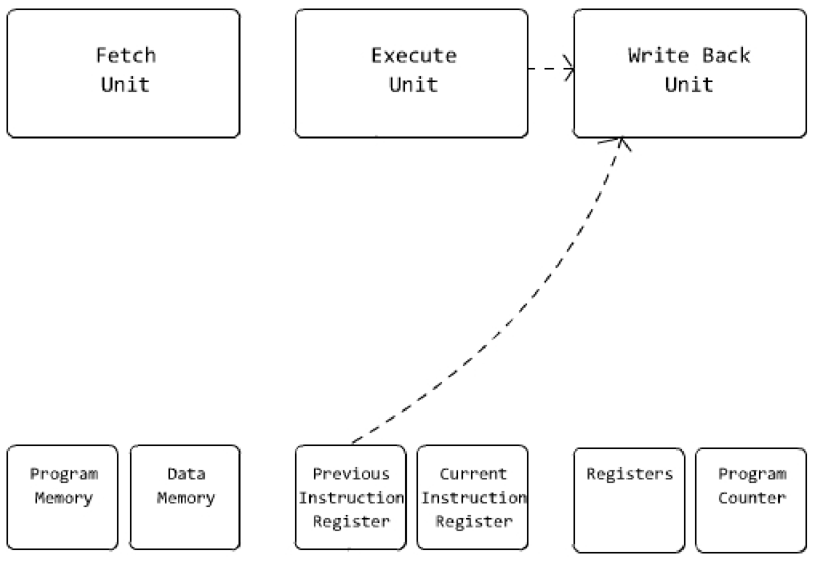
Clock = 1: Fetch unit reads the current instruction.



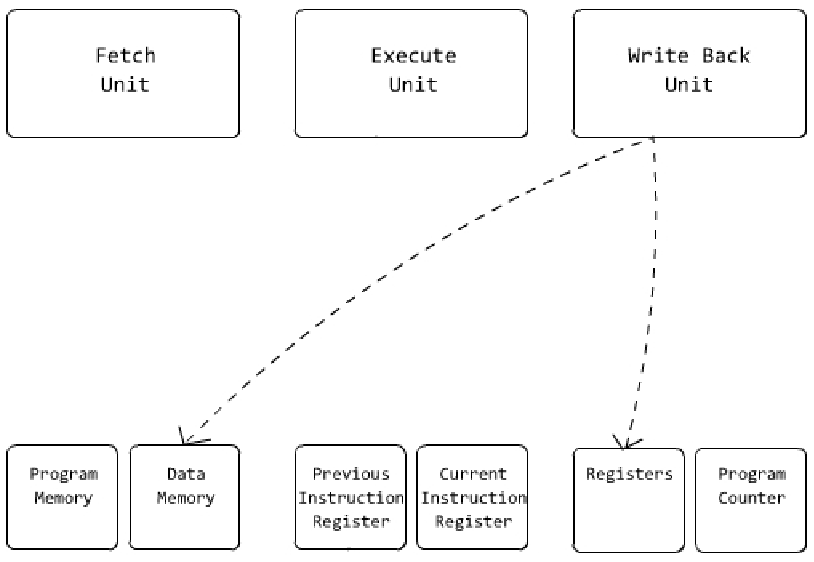
Clock = 1: Fetch places current instruction in the CIR. Previous CI goes to PIR.



Clock = 2: Execute reads the current instruction and any operands from registers/memory.



Clock = 3: Writeback obtains the result from Execute and reads the PIR.



Clock = 3: The PIR reveals where (if anywhere) the result is to be stored.

4.1.3 Coordinating The Functional Units

Coordinating the functional units is a rather delicate process. The structure of the specification must allow each functional unit to "see" the others in order to acquire the necessary state information to carry out its assigned tasks. The relationship of the functional units is as follows.

- **Fetch Unit**: Needs access to the execute unit in order to detect branches and update the program counter accordingly.
- **Execute Unit**: Needs access to the fetch unit in order to retrieve the current instruction. Also needs access to the writeback unit to retrieve operands because the writeback unit coordinates access to the memory and registers.
- **Writeback Unit**: Needs access to the execute unit in order to determine whether the current result needs to be stored and, if so, where.

This leads to the following natural placement of system components:

- **Fetch Unit**: Contains the Program Memory, the Program Counter, The Current Instruction Register and the Previous Instruction Register.
- **Execute Unit**: Contains the Result; a Branch Taken flag; a Writeback flag specifying register-writeback, memory writeback or neither; a memory location for writeback if needed; a register location for writeback if needed.
- **Writeback Unit**: Contains the Data Memory and the Registers.

With regard to data processing, each unit operates thusly:

- **Fetch Unit**: Checks whether execution has branched and updates the program counter and current instruction register accordingly.
- **Execute Unit**: Gets the needed operands, computes the result of the instruction and sets flags for the writeback unit.
- **Writeback Unit**: Gets the result from the execution unit and writes it back to storage accordingly.

In terms of algebra[8], a functional unit is organised in the following manner:

$F_{State} = PM \times PC \times CIR \times PIR$

$F : T \times F_{State} \times E_{State} \times W_{State} \to F_{State}$
$fnext : F_{State} \times E_{State} \times W_{State} \to F_{State}$

$F(0, f, e, w) = f$
$F(t+1, f, e, w) = fnext(F(t, f, e, w), E(t, f, e, w), W(t, f, e, w))$

$fnext(PM, PC, CIR, PIR) = ...$

This structure allows each functional unit to access the data it needs at each stage. Here are the relevant Maude implementations with comments:

[8]No pun intended.

```
***
*** Functional Units
***
*** The functional units have all been moved into one module to resolve
*** issues with visibility
***
fmod FUNCTIONAL-UNITS is
    protecting MEM .
    protecting REG .
    protecting INSTRUCTION-SET .

    sort FeState .
    sort ExState .
    sort WbState .

    *** FETCH OPS

    *** Program Memory, Program Counter,
    *** Current Instruction Register, Previous Instruction Register
    op (_,_,_,_) : Mem Word Word Word -> FeState .

    op pm_ : FeState -> Mem .
    ops pc_ cir_ pir_ : FeState -> Word .

    op feu : Int FeState ExState WbState -> FeState .

    op fenext : FeState ExState WbState -> FeState .


    *** EXECUTE OPS

    *** Result, Taken Flag, WBFlag, MemWBLoc, RegWBLoc
    op (_,_,_,_,_) : Word Bool Int Word OpField -> ExState .

    op exu : Int FeState ExState WbState -> ExState .

    op exnext : FeState ExState WbState -> ExState .

    ops result_ memwbloc_ : ExState -> Word .
    op taken_ : ExState -> Bool .
    op wbflag_ : ExState -> Int .
    op regwbloc : ExState -> OpField .

    *** WRITEBACK OPS

    *** Data Memory, Registers
    op (_,_) : Mem Reg -> WbState .

    op dm_ : WbState -> Mem .
    op reg_ : WbState -> Reg .
    op wbu : Int FeState ExState WbState -> WbState .

    op wbnext : FeState ExState WbState -> WbState .

    *** FETCH VARIABLES

    var PM : Mem .
    var PC PIR CIR : Word .

    var feS : FeState . *** state
    var T : Int .     *** time

    *** Only one time variable is needed since all units should remain
    *** in step due to the lack of interlocking and stalling

    *** EXECUTE VARIABLES

    var TAKEN : Bool .
    var WBFLAG : Int . *** 0 = no writeback, 1 = mem, 2 = reg
    var RESULT MEMWBLOC : Word .
    var REGWBLOC : OpField .

    var exS : ExState . *** state

    *** WRITEBACK VARIABLES

    var wbS : WbState . *** state
    var DM : Mem .
    var REG : Reg .

    *** FETCH EQUATIONS

    eq pm(PM,PC,CIR,PIR) = PM .
    eq pc(PM,PC,CIR,PIR) = PC .
    eq cir(PM,PC,CIR,PIR) = CIR .
    eq pir(PM,PC,CIR,PIR) = PIR .

    *** iterated map
    eq feu(0,feS,exS,wbS) = feS .
    eq feu(T,feS,exS,wbS) = fenext(feu(T - 1,feS,exS,wbS),
                                   exu(T - 1,feS,exS,wbS),
                                   wbu(T - 1,feS,exS,wbS)) [owise] .

    *** If the instruction isn't a branch, increment PC as normal
    ceq fenext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG))
        = (PM,PC + Four,PM[PC],CIR)
        if opcode(cir(PM,PC,CIR,PIR)) =/= JMP .
    *** If the instruction is a branch and is not taken, increment PC as normal
    ceq fenext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG))
```

```
    = (PM,PC + Four,PM[PC],CIR)
    if opcode(cir(PM,PC,CIR,PIR)) == JMP
    and taken(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) == false .
*** If the instruction is a taken branch, jump to location and store return address
ceq fenext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG))
    = (PM,REG[regc(CIR)],PM[REG[regc(CIR)]],CIR)
    if opcode(cir(PM,PC,CIR,PIR)) == JMP
    and taken(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) == true .

*** EXECUTE EQUATIONS

eq result(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) = RESULT .
eq taken(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) = TAKEN .
eq wbflag(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) = WBFLAG .
eq memwbloc(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) = MEMWBLOC .
eq regwbloc(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) = REGWBLOC .

*** iterated map
eq exu(0,feS,exS,wbS) = exS .
eq exu(T,feS,exS,wbS) = exnext(feu(T - 1,feS,exS,wbS),
                               exu(T - 1,feS,exS,wbS),
                               wbu(T - 1,feS,exS,wbS)) [owise] .

*** define instructions

*** NOP (opcode = 0)
ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
    (NOPWORD,false,0,NOPWORD,NOP)
    if opcode(cir(PM,PC,CIR,PIR)) == NOP .
*** ADD (opcode = 1)
ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
    (REG[rega(CIR)] + REG[regb(CIR)],false,2,NOPWORD,REG[regc(CIR)])
    if opcode(cir(PM,PC,CIR,PIR)) == ADD .
*** MULT (opcode = 10)
ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
    (REG[rega(CIR)] * REG[regb(CIR)],false,2,NOPWORD,REG[regc(CIR)])
    if opcode(cir(PM,PC,CIR,PIR)) == MULT .
*** AND (opcode = 11)
ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
    (REG[rega(CIR)] & REG[regb(CIR)],false,2,NOPWORD,REG[regc(CIR)])
    if opcode(cir(PM,PC,CIR,PIR)) == AND .
*** OR (opcode = 100)
ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
    (REG[rega(CIR)] | REG[regb(CIR)],false,2,NOPWORD,REG[regc(CIR)])
    if opcode(cir(PM,PC,CIR,PIR)) == OR .
*** NOT (opcode = 101)
ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
    (!(REG[rega(CIR)]),false,2,NOPWORD,REG[regc(CIR)])
    if opcode(cir(PM,PC,CIR,PIR)) == NOT .
*** SLL (opcode = 110)
```

```
ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
    (REG[rega(CIR)] << REG[regb(CIR)],false,2,NOPWORD,REG[regc(CIR)])
    if opcode(cir(PM,PC,CIR,PIR)) == SLL .
*** LD (opcode = 111)
ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
    (DM[REG[rega(CIR)] + REG[regb(CIR)]],false,2,NOPWORD,REG[regc(CIR)])
    if opcode(cir(PM,PC,CIR,PIR)) == LD .
*** ST (opcode = 1000)
ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
    (REG[rega(CIR)] + REG[regb(CIR)],false,1,REG[regc(CIR)],NOP)
    if opcode(cir(PM,PC,CIR,PIR)) == ST .
*** EQ (opcode = 1001) [RA == RB]
ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
    (constzero32,false,2,NOPWORD,REG[regc(CIR)])
    if opcode(cir(PM,PC,CIR,PIR)) == EQ and REG[rega(CIR)] == REG[regb(CIR)] .
*** EQ (opcode = 1001) [RA =/= RB]
ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
    (constminus1,false,2,NOPWORD,REG[regc(CIR)])
    if opcode(cir(PM,PC,CIR,PIR)) == EQ and REG[rega(CIR)] =/= REG[regb(CIR)] .
*** GT (opcode = 1010) [RA > RB]
ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
    (constzero32,false,2,NOPWORD,REG[regc(CIR)])
    if opcode(cir(PM,PC,CIR,PIR)) == GT and REG[rega(CIR)] gt REG[regb(CIR)] .
*** GT (opcode = 1010) [RA <= RB]
ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
    (constminus1,false,2,NOPWORD,REG[regc(CIR)])
    if opcode(cir(PM,PC,CIR,PIR)) == GT and not REG[rega(CIR)] gt REG[regb(CIR)] .
*** JMP (opcode = 1011) [branch not taken]
ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
    (constzero32,false,0,NOPWORD,NOP)
    if opcode(cir(PM,PC,CIR,PIR)) == JMP and REG[rega(CIR)] =/= REG[constzero8] .
*** JMP (opcode = 1011) [branch taken]
ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
    (PC + Four,true,2,NOPWORD,REG[regb(CIR)])
    if opcode(cir(PM,PC,CIR,PIR)) == JMP and REG[rega(CIR)] == REG[constzero8] .

*** WRITEBACK EQUATIONS

eq dm(DM,REG) = DM .
eq reg(DM,REG) = REG .

eq wbu(0,feS,exS,wbS) = wbS .
eq wbu(T,feS,exS,wbS) = wbnext(feu(T - 1,feS,exS,wbS),
                               exu(T - 1,feS,exS,wbS),
                               wbu(T - 1,feS,exS,wbS)) [owise] .

ceq wbnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
    (DM[result(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC)
    / memwbloc(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC)],REG)
    if wbflag(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) == 1 .
    ceq wbnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
        (DM,REG[result(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC)
        / regwbloc(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC)])
        if wbflag(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) == 2 .
    ceq wbnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
        (DM,REG)
        if wbflag(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) == 0 .

endfm
```

Finally, a top-level iterated map coordinates the processor as a whole and enters some initial values to each of the units.

```
    sort PState .

    op (_,_,_) : FeState ExState WbState -> PState .
    op pmp : Int PState -> PState .
    op pnext : PState -> PState .

    op fetchunit_ : PState -> FeState .
    op executeunit_ : PState -> ExState .
    op writebackunit_ : PState -> WbState .

    var FETCHUNIT : FeState .
    var EXECUTEUNIT : ExState .
    var WRITEBACKUNIT : WbState .

    var PMP : PState .
    var Time : Int .

    eq fetchunit(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT) = FETCHUNIT .
    eq executeunit(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT) = EXECUTEUNIT .
    eq writebackunit(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT) = WRITEBACKUNIT .

    eq pmp(0,PMP) = PMP .
    eq pmp(Time,PMP) = pnext(pmp(Time - 1,PMP)) [owise] .

    eq pnext(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT) =
        (fenext(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT),
        exnext(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT),
        wbnext(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT)) .
```

The implementation can then have reductions performed upon it, as with the programmer's model.

```
    reduce pmp(1, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
```

4.2 Multi-core Processors

Multi-core processors can build upon two pipelined processors. Each is given its own program to run, since multi-core is an idea to take advantage of TLP. They do share data memory, though, and care must be taken to safeguard data integrity.

4.2.1 Consistency Issues

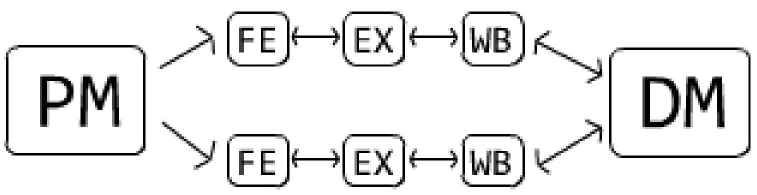
When two processor cores share the same data memory, there is always the possibility that each will execute interdependent instructions simultaneously. This could lead to a compromise of data integrity due to inconsistencies and mistakes in program execution. A processor which executes instructions more efficiently is useless if it makes computational errors. Therefore data consistency in a multicore processor is a key concern and steps must be taken to prevent these issues from happening.

Data hazards in multi-core machines are as described earlier when discussing pipelined machines. The likelihood of them occurring is hopefully much less, since the operating system schedules two distinct processes to each core. However, processes may share memory and communicate, so the potential for inconsistency still remains.

4.2.2 Coordinating The Two Cores

Building upon the pipelined implementation of the previous section, a dual-core processor can be described thusly.

$\sum_{pmp^2} = (Fetch \times Execute \times Writeback)^2 \times Mem^2$



Each pipeline retains its own program counter and registers but the contains of memory are now shared between the pipelines[9].

PMP^2 is loosely defined as follows:

[9]Since PM will never change, it would be perfectly possible to duplicate it into PM_1 and PM_2 and place each in the Fetch state of the respective pipeline.

$PMP^2 : S \times \sum_{PMP^2} \to \sum_{PMP^2}$,
$PMP^2(0, \sigma) = \sigma$,
$PMP^2(s + 1, \sigma) = next(PMP^2(s, \sigma))$,

where $next : \sum_{PMP^2} \to \sum_{PMP^2}$ is defined by

$$\begin{aligned} next(p1, p2, dm, pm) = (&fetch(p1, dm, pm), execute(p1, dm, pm), writeback(p1, dm, pm), \\ &fetch(p2, dm, pm), execute(p2, dm, pm), writeback(p2, dm, pm), \\ &dm(p1, p2, dm, pm), pm) \end{aligned}$$

5 Thoughts and Conclusions

Overall, the algebraic methods employed for describing the changes of state over time in complex microprocessors are rather effective. Provided that the system is properly broken up into smaller conceptual entities, the process of describing a system's state evolution tends to be successful. Although the models developed in this project are nowhere near as complex as a hardware example, the concepts used are extensible enough to be able to describe any given system. The real challenge, it seems, is coordinating the level of abstraction carefully.

Microprocessors which have been specified using the algebraic tools laid out here can be proven mathematically correct using one-step theorem provers. Therefore the continued research in this area has the potential to yield better hardware in practice. The ARM6 microprocessor was developed using similar methods[9]. Although this project has not been specifically aimed at proving correctness of specifications, it aims to show that machines can be specified elegantly in algebra, reduced by Maude and ultimately proven correct.

This project began in September of 2006 as an ambitious quest to explore specification of several types of machine: Pipelined, Multicore, Multithreaded and Superscalar. In the months that followed, it became increasingly clear that developing each of these would not be feasible and focus was shifted to the development of the Pipelined and Multicore models with significant emphasis on the former. Overall, I am rather pleased with the way these have turned out.

In future work, it would be a good idea to model a processor with pipeline interlocks. MIPS processors are used in industry but there are plenty of other commercial processors which implement their own locking mechanisms. Although MIPS may be simple, it requires the programmer to know more about the underlying architecture than should really be necessary. Therefore future endeavours in this field should focus (amongst other things) upon the development of effective dependency resolution.

References

[1] “Computer Architecture: A Quantitive Approach”, Third Edition by John L. Hennessy, David A. Patterson.

[2] “Algebraic Models of Microprocessors: Architecture and Organisation” by N. A. Harman and J. V. Tucker, Acta Informatica 33 (1996).

[3] “Correctness and Verification of Hardware Systems Using Maude” by N. A. Harman, Swansea University.

[4] “High Performance Microprocessors” by N. A. Harman and A. M. Gimblett, Swansea University, 2006.

[5] “System Specification” by N. A. Harman and M. Seisenberger, Swansea University, 2006.

[6] “Algebraic Models of Simultaneous Multi-Threaded and Multi-Core Microprocessors” by N. A. Harman, Swansea University, 2006.

[7] “Algebraic Models of Computers” by N. A. Harman, Swansea University, 2006.

[8] “Modelling SMT and CMT Processors: A Simple Case Study” by N. A. Harman, Swansea University, 2007.

[9] “Formal specification and verification of ARM6.” by A. C. J. Fox. In D Basin and B Wolff, editors, TPHOLs 03, volume 2758 of Lecture Notes in Computer Science, pages 2540. Springer-Verlag, 2003.

[10] Intel’s White Papers, The Intel Technology Journal (1998 - 2006).

[11] “The Maude 2.0 System” by Manuel Clavel, Francisco Durn, Steven Eker, Patrick Lincoln, Narciso Mart-Oliet, Jos Meseguer and Carolyn Talcott. Published in Proc. Rewriting Techniques and Applications, 2003 , Springer-Verlag LNCS 2706, 76-87, 2003.

[12] “The Java(TM) Language Specification”, 3rd Edition, by James Gosling, Bill Joy, Guy Steele, Gilad Bracha, 2005.

[13] “Programming Ruby: The Pragmatic Programmer’s Guide”, by Dave Thomas, Chad Fowler, Andy Hunt, 2004.

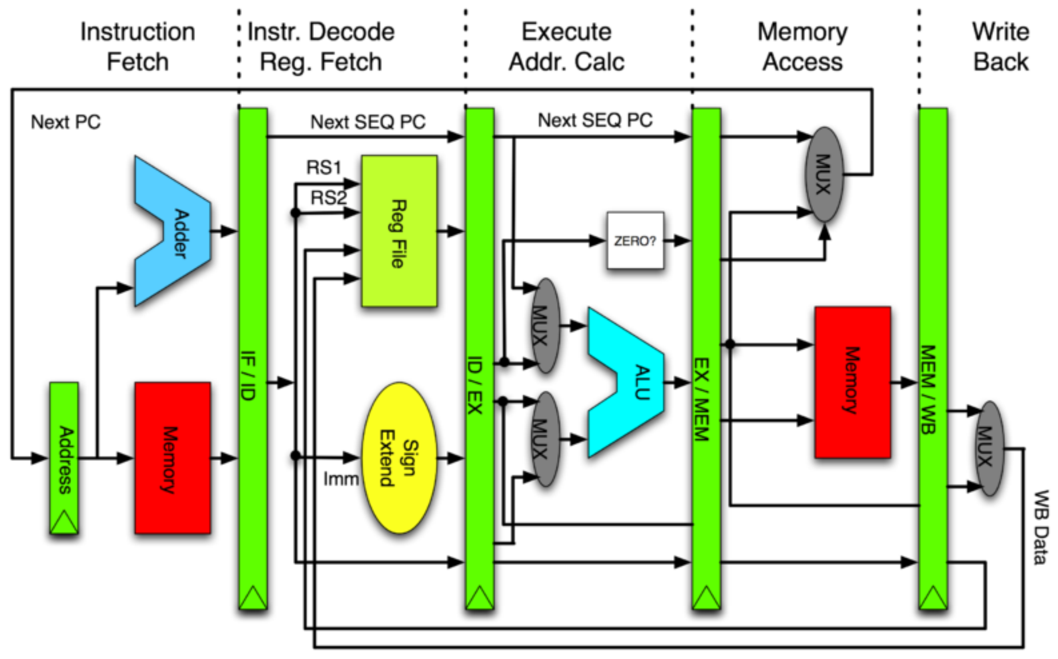
[14] Microprocessor diagrams acquired under a free usage license from the public domain.

A Case Studies

A.1 The MIPS Processor

The MIPS processor (Microprocessor without Interlocked Pipeline Stages) was developed at Stanford by a team lead by Dr. John L. Hennesy. As the name suggests, the processor was designed without pipeline stage interlocking. This means there is no built-in protection from data hazards and any name or value dependencies must be resolved by the compiler before runtime. In terms of architecture, the MIPS processor is a very clean RISC style machine. Despite the desktop market being dominated by x86 architecture, MIPS processors are currently thriving in specialised devices such as gaming consoles, set-top boxes and home entertainment systems.

The MIPS pipeline uses 5 stages - Fetch, Decode, Execute, Memory Access, and Writeback. Here is a graphical representation of the flow of data in the pipeline.



The MIPS architecture is a useful base for the design of microprocessor specification because it aims to be clean and simple. The lack of interlocks can be viewed as a downfall but this simplification of the pipeline allows for faster prototyping and fewer errors in development of the machine.

B Source Code

```
***
*** "A 32-bit Generic RISC Microprocessor Specification"
*** Sean Handley, sean.handley@gmail.com
*** 2006-08-02
***

***
*** Definitions for binary arithmetic
***
fmod BINARY is
    protecting INT .

    sorts Bit Bits .

    subsort Bit < Bits .

    ops 0 1 : -> Bit .
    op __ : Bits Bits -> Bits [assoc prec 1 gather (e E)] .
    op |_| : Bits -> Int .
    op normalize : Bits -> Bits .
    op bits : Bits Int Int -> Bits .
    op _++_  : Bits Bits -> Bits [assoc comm prec 5 gather (E e)] .
    op _**_  : Bits Bits -> Bits [assoc comm prec 4 gather (E e)] .
    op _>_  : Bits Bits -> Bool [prec 6 gather (E E)] .
    op not_ : Bits -> Bits [prec 2 gather (E)] .
    op _and_ : Bits Bits -> Bits [assoc comm prec 2 gather (E e)] .
    op _or_ : Bits Bits -> Bits [assoc comm prec 2 gather (E e)] .
    op _sl_ : Bits Bits -> Bits [prec 2 gather (E e)] .
    op _-- : Bits -> Bits [prec 2 gather (E)] .
    op bin2int : Bits -> Int .

    vars S T : Bits .
    vars B C : Bit .
    var L : Bool .
    vars I J : Int .

    op constzero32 : -> Bits .
    op constzero8 : -> Bits .

    *** define constants for zero^32 and zero^8
    eq constzero32 =    0 0 0 0 0 0 0 0
                       0 0 0 0 0 0 0 0
                       0 0 0 0 0 0 0 0
                       0 0 0 0 0 0 0 0 .

    eq constminus1 =    1 1 1 1 1 1 1 1
                       1 1 1 1 1 1 1 1
                       1 1 1 1 1 1 1 1
                       1 1 1 1 1 1 1 1 .

    eq constzero8 =     0 0 0 0 0 0 0 0 .

    *** Binary to Integer
    ceq bin2int(B) = 0 if normalize(B) == 0 .
    ceq bin2int(B) = 1 if normalize(B) == 1 .
    eq bin2int(S) = 1 + bin2int((S)--) .

    *** Length
```

```
eq | B | = 1 .
eq | S B | = | S | + 1 .

*** Extract Bits...
eq bits(S B,0,0) = B .
eq bits(B,J,0) = B .
ceq bits(S B,J,0) = bits(S, J - 1,0) B if J > 0 .
ceq bits(S B,J,I) = bits(S,J - 1,I - 1) if I > 0 and J > 0 .

*** Not
eq not (S T) = (not S) (not T) .
eq not 0 = 1 .
eq not 1 = 0 .

*** And
eq B and 0 = 0 .
eq B and 1 = B .
eq (S B) and (T C) = (S and T) (B and C) .

*** Or
eq B or 0 = B .
eq B or 1 = 1 .
eq (S B) or (T C) = (S or T) (B or C) .

*** Normalize supresses zeros at the
*** left of a binary number
eq normalize(0) = 0 .
eq normalize(1) = 1 .
eq normalize(0 S) = normalize(S) .
eq normalize(1 S) = 1 S .

*** Greater than
eq 0 > S = false .
eq 1 > (0).Bit = true .
eq 1 > (1).Bit = false .
eq B > (0 S) = B > S .
eq B > (1 S) = false .
eq (1 S) > B = true .
eq (B S) > (C T)
= if | normalize(B S) | > | normalize(C T) |
    then true
    else if | normalize(B S) | < | normalize(C T) |
        then false
    else (S > T)
    fi
fi .

*** Binary addition
eq 0 ++ S = S .
eq 1 ++ 1 = 1 0 .
eq 1 ++ (T 0) = T 1 .
eq 1 ++ (T 1) = (T ++ 1) 0 .
eq (S B) ++ (T 0) = (S ++ T) B .
eq (S 1) ++ (T 1) = (S ++ T ++ 1) 0 .

*** Binary multiplication
eq 0 ** T = 0 .
eq 1 ** T = T .
```

```
    eq (S B) ** T = ((S ** T) 0) ++ (B ** T) .

    *** Decrement
    eq 0 -- = 0 .
    eq 1 -- = 0 .
    eq (S 1) -- = normalize(S 0) .
    ceq (S 0) -- = normalize(S --) 1 if normalize(S) =/= 0 .
    ceq (S 0) -- = 0 if normalize(S) == 0 .

    *** Shift left
    ceq S sl T = ((S 0) sl (T --)) if bin2int(T) > 0 .
    eq S sl T = S .
endfm

***
*** Module for dealing with machine words and instruction formats.
***
fmod MACHINE-WORD is
    protecting BINARY .

    *** 32-bit machine word, 1 byte per opcode/reg address
    *** Opfields and register addresses are both 1 byte so they share a name

    sorts OpField Word .

    subsort OpField < Bits .
    subsort Word < Bits .

    op opcode : Word -> OpField .
    ops rega regb regc : Word -> OpField .

    op _+_ : Word Word -> Word .
    op _+_ : OpField OpField -> OpField .

    op _+8_ : Word Word -> Word .
    op _+8_ : OpField OpField -> OpField .

    op _*_ : Word Word -> Word .
    op _*_ : OpField OpField -> OpField .

    op _&_ : Word Word -> Word .
    op _&_ : OpField OpField -> OpField .

    op _|_ : Word Word -> Word .
    op _|_ : OpField OpField -> OpField .

    op !_ : Word -> Word .
    op !_ : OpField -> OpField .

    op _<<_ : Word Word -> Word .
    op _<<_ : OpField OpField -> OpField .

    op _gt_ : Word Word -> Bool .
    op _gt_ : OpField OpField -> Bool .

    vars B1 B2 B3 B4 B5 B6 B7 B8 : Bit .
    vars B9 B10 B11 B12 B13 B14 B15 B16 : Bit .
    vars B17 B18 B19 B20 B21 B22 B23 B24 : Bit .
    vars B25 B26 B27 B28 B29 B30 B31 B32 : Bit .
    vars V W : Word .
    vars A B : OpField .

    *** 8 bits = opfield
    mb (B1 B2 B3 B4 B5 B6 B7 B8) : OpField .

    *** 32 bits = word and/or memory address
    mb (B1 B2 B3 B4 B5 B6 B7 B8
        B9 B10 B11 B12 B13 B14 B15 B16
        B17 B18 B19 B20 B21 B22 B23 B24
        B25 B26 B27 B28 B29 B30 B31 B32) : Word .

    *** 1 byte per opcode/reg address
    eq opcode(W) = bits(W,31,24) .
    *** eq opcode(W) = bits(W,7,0) .
    eq rega(W) = bits(W,23,16) .
    *** eq rega(W) = bits(W,15,8) .
    eq regb(W) = bits(W,15,8) .
    *** eq regb(W) = bits(W,23,16) .
    eq regc(W) = bits(W,7,0) .
    *** eq regc(W) = bits(W,31,24) .

    *** truncate the last 32 bits/8 bits resp
    eq V + W = bits(V ++ W,31,0) .
    eq A + B = bits(A ++ B,7,0) .
    eq V gt W = V > W .
    eq A gt B = A > B .
    eq V * W = bits(V ** W,31,0) .
    eq A * B = bits(A ** B,7,0) .
    eq ! V = bits(not V,31,0) .
    eq ! A = bits(not A,7,0) .
    eq V & W = bits(V and W,31,0) .
    eq A & B = bits(A and B,7,0) .
    eq V | W = bits(V or W,31,0) .
    eq A | B = bits(A or B,7,0) .
    eq V << W = bits(V sl W,31,0) .
    eq A << B = bits(A sl B,7,0) .
endfm

***
*** Module for representing memory. Words are 32 bits.
***
fmod MEM is
    protecting MACHINE-WORD .

    sorts Mem .

    op _[_] : Mem Word -> Word .          *** read
    op _[_/_] : Mem Word Word -> Mem .     *** write

    var M : Mem .

    var A B : Word .
    var W : Word .

    eq M[W / A][A] = W .
    eq M[W / A][B] = M[B] [owise] . *** seek if not found
endfm
```

```
***
*** Module for representing registers.
***
fmod REG is
    protecting MACHINE-WORD .

    sorts Reg .

    op _[_] : Reg OpField -> Word .            *** read
    op _[_/_] : Reg Word OpField -> Reg .    *** write

    var R : Reg .
    var A B : OpField .
    var W : Word .

    eq R[W / A][A] = W .
    eq R[W / A][B] = R[B] [owise] . *** seek if not found
endfm

***
*** State of SPM, together with tupling and projection functions
***

fmod SPM-STATE is
    protecting MEM .
    protecting REG .

    sort SPMstate .

    op (_,_,_,_) : Mem Mem Word Reg -> SPMstate .

    *** project out program and data mem
    ops mp_ md_ : SPMstate -> Mem .

    *** project out PC
    op pc_ : SPMstate -> Word .

    *** project out regs
    op reg_ : SPMstate -> Reg .

    var S : SPMstate .
    vars MP MD : Mem .
    var PC : Word .
    var REG : Reg .

    *** tuple member accessor functions
    eq mp(MP,MD,PC,REG) = MP .
    eq md(MP,MD,PC,REG) = MD .
    eq pc(MP,MD,PC,REG) = PC .
    eq reg(MP,MD,PC,REG) = REG .
endfm

***
*** SPM
***
*** This is the "main" function, where we define the state funtion spm and the
*** next-state function next.
***
```

```
fmod SPM is
    protecting SPM-STATE .

    ops ADD32 MULT AND OR NOT : -> OpField .
    ops SLL LD32 ST32 EQ GT JMP : -> OpField .
    op Four : -> Word .

    op spm : Int SPMstate -> SPMstate .

    op next : SPMstate -> SPMstate .

    var SPM : SPMstate .
    var T : Int .
    var MP MD : Mem .
    var PC A : Word .
    var REG : Reg .
    var O P : OpField .

    *** define the opcodes
    eq ADD32 = 0 0 0 0 0 0 0 0 .
    eq MULT = 0 0 0 0 0 0 1 0 .
    eq AND = 0 0 0 0 0 0 1 1 .
    eq OR = 0 0 0 0 0 1 0 0 .
    eq NOT = 0 0 0 0 0 1 0 1 .
    eq SLL = 0 0 0 0 0 1 1 0 .
    eq LD32 = 0 0 0 0 0 1 1 1 .
    eq ST32 = 0 0 0 0 1 0 0 0 .
    eq EQ = 0 0 0 0 1 0 0 1 .
    eq GT = 0 0 0 0 1 0 1 0 .
    eq JMP = 0 0 0 0 1 0 1 1 .

    *** constant four to jump to the next instruction
    eq Four =   0 0 0 0 0 0 0 0
               0 0 0 0 0 0 0 0
               0 0 0 0 0 0 0 0
               0 0 0 0 0 1 0 0 .

    eq spm(0,SPM) = SPM .
    eq spm(T,SPM) = next(spm(T - 1,SPM)) [owise] .

    *** Fix the zero register
    eq REG[0 0 0 0 0 0 0 0] = constzero32 .
    ceq REG[A / O][O] = constzero32 if O == constzero8 .
    eq REG[A / O][P] = REG[P] [owise].

    *** define instructions

    *** ADD32 (opcode = 0)
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[REG[rega(MP[PC])] + REG[regb(MP[PC])] / regc(MP[PC])])
        if opcode(MP[PC]) == ADD32 .
    *** MULT (opcode = 10)
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[REG[rega(MP[PC])] * REG[regb(MP[PC])] / regc(MP[PC])])
        if opcode(MP[PC]) == MULT .
    *** AND (opcode = 11)
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[REG[rega(MP[PC])] & REG[regb(MP[PC])] / regc(MP[PC])])
        if opcode(MP[PC]) == AND .
    *** OR (opcode = 100)
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[REG[rega(MP[PC])] | REG[regb(MP[PC])] / regc(MP[PC])])
        if opcode(MP[PC]) == OR .
    *** NOT (opcode = 101)
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[!(REG[rega(MP[PC])]) / regc(MP[PC])])
        if opcode(MP[PC]) == NOT .
    *** SLL (opcode = 110)
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[REG[rega(MP[PC])] << REG[regb(MP[PC])] / regc(MP[PC])])
        if opcode(MP[PC]) == SLL .
    *** LD32 (opcode = 111)
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[MD[REG[rega(MP[PC])] + REG[regb(MP[PC])]] / regc(MP[PC])])
        if opcode(MP[PC]) == LD32 .
    *** ST32 (opcode = 1000)
    ceq next(MP,MD,PC,REG) = (MP,
        MD[REG[regc(MP[PC])] / (REG[rega(MP[PC])] + REG[regb(MP[PC])])], PC + Four, REG)
        if opcode(MP[PC]) == ST32 .
    *** EQ (opcode = 1001) [RA == RB]
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 / regc(MP[PC])])
        if opcode(MP[PC]) == EQ and REG[rega(MP[PC])] == REG[regb(MP[PC])] .
    *** EQ (opcode = 1001) [RA =/= RB]
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 / regc(MP[PC])])
        if opcode(MP[PC]) == EQ and REG[rega(MP[PC])] =/= REG[regb(MP[PC])] .
    *** GT (opcode = 1010) [RA > RB]
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 / regc(MP[PC])])
        if opcode(MP[PC]) == GT and REG[rega(MP[PC])] gt REG[regb(MP[PC])] .
    *** GT (opcode = 1010) [RA <= RB]
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
        REG[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 / regc(MP[PC])])
        if opcode(MP[PC]) == GT and not (REG[rega(MP[PC])] gt REG[regb(MP[PC])]) .
    *** JMP (opcode = 1011) [branch not taken]
    ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,REG)
        if opcode(MP[PC]) == JMP and REG[rega(MP[PC])] =/= REG[0 0 0 0 0 0 0 0] .
    *** JMP (opcode = 1011) [branch taken]
    ceq next(MP,MD,PC,REG) = (MP,MD,
        REG[regc(MP[PC])], REG[PC + Four / regb(MP[PC])])
        if opcode(MP[PC]) == JMP and REG[rega(MP[PC])] == REG[0 0 0 0 0 0 0 0] .
endfm

***
*** The final module is to define an actual program and run it.
***
fmod RUNPROGS is
    protecting SPM .        *** import the microprocessor representation

    ops Md Mp : -> Mem .
    op Rg : -> Reg .
    op Pc : -> Word .

    *** Set the PC to zero
```

```
eq Pc = 0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0 .

*** R1 = 1
eq Rg[0 0 0 0 0 0 0 1] =
       0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 1 .

*** R2 = 0
eq Rg[0 0 0 0 0 0 1 0] =
       0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0 .

*** R13 = 252
eq Rg[0 0 0 0 1 1 0 1] =
       0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0
       1 1 1 1 1 1 0 0 .

*** Mem[1] = 6
eq Md[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1] =
       0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0
       0 0 0 0 0 1 1 0 .

*** Mem[2] = 5
eq Md[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0] =
       0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0
       0 0 0 0 0 1 0 1 .

*** INST 1 [ Load (RG1+RG1) into RG3 ] -> Mem[2] = 5
eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] =
       0 0 0 0 0 1 1 1 *** LOAD
       0 0 0 0 0 0 0 1 *** RG1
       0 0 0 0 0 0 0 1 *** RG1
       0 0 0 0 0 0 1 1 . *** RG3

*** INST 2 [ Load (RG1+RG2) into RG4 ] -> Mem[2] = 5
eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0] =
       0 0 0 0 0 1 1 1 *** LOAD
       0 0 0 0 0 0 0 1 *** RG1
       0 0 0 0 0 0 0 1 *** RG1
       0 0 0 0 0 1 0 0 . *** RG4

*** INST 3 [ Shift left R3 by R4 and store in R5 ]
eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0] =
       0 0 0 0 0 1 1 0 *** SHIFTL
       0 0 0 0 0 0 1 1 *** RG3
       0 0 0 0 0 1 0 0 *** RG4
       0 0 0 0 0 1 0 1 . *** RG5

*** INST 4 [ Store RG5 in Mem[RG2+RG3] ] -> Mem[5] = 160
eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0] =
       0 0 0 0 1 0 0 0 *** STORE
       0 0 0 0 0 0 1 0 *** RG2
       0 0 0 0 0 0 1 1 *** RG3
       0 0 0 0 0 1 0 1 . *** RG5

*** INST 5 [ Add RG3 and RG4 and store in RG6 ] -> RG[6] = 10
eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0] =
       0 0 0 0 0 0 0 0 *** ADD
       0 0 0 0 0 0 1 1 *** RG3
       0 0 0 0 0 1 0 0 *** RG4
       0 0 0 0 0 1 1 0 . *** RG6

*** INST 6 [ Mult RG3 by RG4 and store in RG7 ] -> RG[7] = 25
eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0] =
       0 0 0 0 0 0 1 0 *** MULT
       0 0 0 0 0 0 1 1 *** RG3
       0 0 0 0 0 1 0 0 *** RG4
       0 0 0 0 0 1 1 1 . *** RG7

*** INST 7 [ Bitwise and of RG3 and RG4, stored in RG8 ] -> RG[8] = 5
eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0] =
       0 0 0 0 0 0 1 1 *** AND
       0 0 0 0 0 0 1 1 *** RG3
       0 0 0 0 0 1 0 0 *** RG4
       0 0 0 0 1 0 0 0 . *** RG8

*** INST 8 [ Bitwise or of RG3 and RG4, stored in RG9 ] -> RG[9] = 5
eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0] =
       0 0 0 0 0 1 0 0 *** OR
       0 0 0 0 0 0 1 1 *** RG3
       0 0 0 0 0 1 0 0 *** RG4
       0 0 0 0 1 0 0 1 . *** RG9

*** INST 9 [ Inverse of RG3, stored in R10 ]
eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0] =
       0 0 0 0 0 1 0 1 *** NOT
       0 0 0 0 0 0 1 1 *** RG3
       0 0 0 0 0 0 0 0 *** RG0
       0 0 0 0 1 0 1 0 . *** RG10

*** INST 10 [ Test if RG7 = RG8 ] -> RG[7] = 25, RG[8] = 5, Answer = -1 (false)
eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0] =
       0 0 0 0 1 0 0 1 *** EQ
       0 0 0 0 0 1 1 1 *** RG7
       0 0 0 0 1 0 0 0 *** RG8
       0 0 0 0 1 0 1 1 . *** R11

*** INST 11 [ Test if R11 == 0 ] -> false, no jump
eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0] =
       0 0 0 0 1 0 1 1 *** JMP
       0 0 0 0 1 0 1 1 *** R11
       0 0 0 0 1 1 0 0 *** R12
       0 0 0 0 1 1 0 1 . *** R13
```

```
    *** INST 12 [ Test if RG7 > RG8 ] -> RG[7] = 25, RG[8] = 5, Answer = 0 (true)
    eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0] =
            0 0 0 0 1 0 1 0 *** GT
            0 0 0 0 0 1 1 1 *** RG7
            0 0 0 0 1 0 0 0 *** RG8
            0 0 0 0 1 0 1 1 . *** RG11

    *** INST 13 [ Test if RG11 == 0 ] -> true, jump to R13, store PC+4 in RG12
    eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0] =
            0 0 0 0 1 0 1 1 *** JMP
            0 0 0 0 1 0 1 1 *** RG11
            0 0 0 0 1 1 0 0 *** RG12
            0 0 0 0 1 1 0 1 . *** RG13

    *** INST 14 [ Subroutine ][ Add RG3 to RG4 and store in RG14 ] -> R[14] = 10
    eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0] =
            0 0 0 0 0 0 0 0 *** ADD
            0 0 0 0 0 0 1 1 *** RG3
            0 0 0 0 0 1 0 0 *** RG4
            0 0 0 0 1 1 1 0 . *** RG14

    *** INST 15 [ Test if R11 == 0] -> True, jump back to R[12]
    eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0] =
            0 0 0 0 1 0 1 1 *** JMP
            0 0 0 0 1 0 1 1 *** RG11
            0 0 0 0 1 1 1 1 *** RG15
            0 0 0 0 1 1 0 0 . *** RG12

    *** INST 16 [ Return from subroutine ] [ Add R7 to R8 and store in R16 ] -> R[16] = 25 + 5 =3
    eq Mp[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0] =
            0 0 0 0 0 0 0 0 *** ADD
            0 0 0 0 0 1 1 1 *** RG7
            0 0 0 0 1 0 0 0 *** RG8
            0 0 0 1 0 0 0 0 . *** R16
endfm

***
*** Now run the program
***
reduce spm(1, (Mp,Md,Pc,Rg)) .
reduce spm(2, (Mp,Md,Pc,Rg)) .
reduce spm(3, (Mp,Md,Pc,Rg)) .
reduce spm(4, (Mp,Md,Pc,Rg)) .
reduce spm(5, (Mp,Md,Pc,Rg)) .
reduce spm(6, (Mp,Md,Pc,Rg)) .
reduce spm(7, (Mp,Md,Pc,Rg)) .
reduce spm(8, (Mp,Md,Pc,Rg)) .
reduce spm(9, (Mp,Md,Pc,Rg)) .
reduce spm(10, (Mp,Md,Pc,Rg)) .
reduce spm(11, (Mp,Md,Pc,Rg)) .
reduce spm(12, (Mp,Md,Pc,Rg)) .
reduce spm(13, (Mp,Md,Pc,Rg)) .
reduce spm(14, (Mp,Md,Pc,Rg)) .
reduce spm(15, (Mp,Md,Pc,Rg)) .
reduce spm(16, (Mp,Md,Pc,Rg)) .

q
```

```
***
*** "A 32-bit Pipelined RISC Microprocessor Specification"
*** Sean Handley, sean.handley@gmail.com
*** 2006-11-10
***

***
*** Definitions for binary arithmetic
***
fmod BINARY is
   protecting INT .

   sorts Bit Bits .

   subsort Bit < Bits .

   ops 0 1 : -> Bit .
   op __ : Bits Bits -> Bits [assoc prec 1 gather (e E)] .
   op |_| : Bits -> Int .
   op normalize : Bits -> Bits .
   op bits : Bits Int Int -> Bits .
   op _++_  : Bits Bits -> Bits [assoc comm prec 5 gather (E e)] .
   op _**_  : Bits Bits -> Bits [assoc comm prec 4 gather (E e)] .
   op _>_  : Bits Bits -> Bool [prec 6 gather (E E)] .
   op not_ : Bits -> Bits [prec 2 gather (E)] .
   op _and_ : Bits Bits -> Bits [assoc comm prec 2 gather (E e)] .
   op _or_ : Bits Bits -> Bits [assoc comm prec 2 gather (E e)] .
   op _sl_ : Bits Bits -> Bits [prec 2 gather (E e)] .
   op _-- : Bits -> Bits [prec 2 gather (E)] .
   op bin2int : Bits -> Int .

   vars S T : Bits .
   vars B C : Bit .
   var L : Bool .
   vars I J : Int .

   op constzero32 : -> Bits .
   op constzero8 : -> Bits .
   op constminus1 : -> Bits .

   *** define constants for zero^32 and zero^8
   eq constzero32 =   0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0
                     0 0 0 0 0 0 0 0 .

   eq constzero8 =    0 0 0 0 0 0 0 0 .

   eq constminus1 =   1 1 1 1 1 1 1 1
                     1 1 1 1 1 1 1 1
                     1 1 1 1 1 1 1 1
                     1 1 1 1 1 1 1 1 .

   *** Binary to Integer
   ceq bin2int(B) = 0 if normalize(B) == 0 .
   ceq bin2int(B) = 1 if normalize(B) == 1 .
   eq bin2int(S) = 1 + bin2int((S)--) .

   *** Length
   eq | B | = 1 .
   eq | S B | = | S | + 1 .

   *** Extract Bits...
   eq bits(S B,0,0) = B .
   eq bits(B,J,0) = B .
   ceq bits(S B,J,0) = bits(S, J - 1,0) B if J > 0 .
   ceq bits(S B,J,I) = bits(S,J - 1,I - 1) if I > 0 and J > 0 .

   *** Not
   eq not (S T) = (not S) (not T) .
   eq not 0 = 1 .
   eq not 1 = 0 .

   *** And
   eq B and 0 = 0 .
   eq B and 1 = B .
   eq (S B) and (T C) = (S and T) (B and C) .

   *** Or
   eq B or 0 = B .
   eq B or 1 = 1 .
   eq (S B) or (T C) = (S or T) (B or C) .

   *** Normalize supresses zeros at the
   *** left of a binary number
   eq normalize(0) = 0 .
   eq normalize(1) = 1 .
   eq normalize(0 S) = normalize(S) .
   eq normalize(1 S) = 1 S .

   *** Greater than
   eq 0 > S = false .
   eq 1 > (0).Bit = true .
   eq 1 > (1).Bit = false .
   eq B > (0 S) = B > S .
   eq B > (1 S) = false .
   eq (1 S) > B = true .
   eq (B S) > (C T)
   = if | normalize(B S) | > | normalize(C T) |
       then true
       else if | normalize(B S) | < | normalize(C T) |
           then false
       else (S > T)
       fi
   fi .

   *** Binary addition
   eq 0 ++ S = S .
   eq 1 ++ 1 = 1 0 .
   eq 1 ++ (T 0) = T 1 .
   eq 1 ++ (T 1) = (T ++ 1) 0 .
   eq (S B) ++ (T 0) = (S ++ T) B .
   eq (S 1) ++ (T 1) = (S ++ T ++ 1) 0 .

   *** Binary multiplication
   eq 0 ** T = 0 .
   eq 1 ** T = T .
   eq (S B) ** T = ((S ** T) 0) ++ (B ** T) .

   *** Decrement
   eq 0 -- = 0 .
```

```
    eq 1 -- = 0 .
    eq (S 1) -- = normalize(S 0) .
    ceq (S 0) -- = normalize(S --) 1 if normalize(S) =/= 0 .
    ceq (S 0) -- = 0 if normalize(S) == 0 .

    *** Shift left
    ceq S sl T = ((S 0) sl (T --)) if bin2int(T) > 0 .
    eq S sl T = S .
endfm

***
*** Module for dealing with machine words and instruction formats.
***
fmod MACHINE-WORD is
    protecting BINARY .

    *** 32-bit machine word, 1 byte per opcode/reg address
    *** Opfields and register addresses are both 1 byte so they share a name

    sorts OpField Word .

    subsort OpField < Bits .
    subsort Word < Bits .

    op opcode : Word -> OpField .
    ops rega regb regc : Word -> OpField .

    op _+_ : Word Word -> Word .
    op _+_ : OpField OpField -> OpField .

    op _*_ : Word Word -> Word .
    op _*_ : OpField OpField -> OpField .

    op _&_ : Word Word -> Word .
    op _&_ : OpField OpField -> OpField .

    op _|_ : Word Word -> Word .
    op _|_ : OpField OpField -> OpField .

    op !_ : Word -> Word .
    op !_ : OpField -> OpField .

    op _<<_ : Word Word -> Word .
    op _<<_ : OpField OpField -> OpField .

    op _gt_ : Word Word -> Bool .
    op _gt_ : OpField OpField -> Bool .

    op Four : -> Bits .

    vars B1 B2 B3 B4 B5 B6 B7 B8 : Bit .
    vars B9 B10 B11 B12 B13 B14 B15 B16 : Bit .
    vars B17 B18 B19 B20 B21 B22 B23 B24 : Bit .
    vars B25 B26 B27 B28 B29 B30 B31 B32 : Bit .

    vars V W : Word .
    vars A B : OpField .

    *** 8 bits = opfield
    mb (B1 B2 B3 B4 B5 B6 B7 B8) : OpField .

    *** 32 bits = word and/or memory address
    mb (B1 B2 B3 B4 B5 B6 B7 B8
        B9 B10 B11 B12 B13 B14 B15 B16
        B17 B18 B19 B20 B21 B22 B23 B24
        B25 B26 B27 B28 B29 B30 B31 B32) : Word .

    *** 1 byte per opcode/reg address
    eq opcode(W) = bits(W,31,24) .
    *** eq opcode(W) = bits(W,7,0) .
    eq rega(W) = bits(W,23,16) .
    *** eq rega(W) = bits(W,15,8) .
    eq regb(W) = bits(W,15,8) .
    *** eq regb(W) = bits(W,23,16) .
    eq regc(W) = bits(W,7,0) .
    *** eq regc(W) = bits(W,31,24) .

    *** truncate the last 32 bits/8 bits resp
    eq V + W = bits(V ++ W,31,0) .
    eq A + B = bits(A ++ B,7,0) .
    eq V gt W = V > W .
    eq A gt B = A > B .
    eq V * W = bits(V ** W,31,0) .
    eq A * B = bits(A ** B,7,0) .
    eq ! V = bits(not V,31,0) .
    eq ! A = bits(not A,7,0) .
    eq V & W = bits(V and W,31,0) .
    eq A & B = bits(A and B,7,0) .
    eq V | W = bits(V or W,31,0) .
    eq A | B = bits(A or B,7,0) .
    eq V << W = bits(V sl W,31,0) .
    eq A << B = bits(A sl B,7,0) .

    *** constant four to jump to the next instruction
    eq Four   = 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0
                0 0 0 0 0 1 0 0 .

endfm

***
*** Module for representing memory. Words are 32 bits.
***
fmod MEM is
    protecting MACHINE-WORD .

    sorts Mem .

    op _[_] : Mem Word -> Word .          *** read
    op _[_/_] : Mem Word Word -> Mem .     *** write

    var M : Mem .

    var A B : Word .
    var W : Word .

    eq M[W / A][A] = W .
    eq M[W / A][B] = M[B] [owise] .
endfm

***
*** Module for representing registers.
***
```

```
fmod REG is
   protecting MACHINE-WORD .

   sorts Reg .

   op _[_] : Reg OpField -> Word .          *** read
   op _[_/_] : Reg Word OpField -> Reg .     *** write

   var R : Reg .
   var A B : OpField .
   var W : Word .

   eq R[W / A][A] = W .
   eq R[W / A][B] = R[B] [owise] .
endfm

***
*** Instruction definitions
***
fmod INSTRUCTION-SET is
   protecting MACHINE-WORD .

   ops ADD MULT AND OR NOT : -> OpField .
   ops SLL LD ST EQ GT JMP NOP : -> OpField .
   op NOPWORD : -> Word .

   *** define the opcodes
   eq ADD    = 0 0 0 0 0 0 0 1 .
   eq MULT   = 0 0 0 0 0 0 1 0 .
   eq AND    = 0 0 0 0 0 0 1 1 .
   eq OR     = 0 0 0 0 0 1 0 0 .
   eq NOT    = 0 0 0 0 0 1 0 1 .
   eq SLL    = 0 0 0 0 0 1 1 0 .
   eq LD     = 0 0 0 0 0 1 1 1 .
   eq ST     = 0 0 0 0 1 0 0 0 .
   eq EQ     = 0 0 0 0 1 0 0 1 .
   eq GT     = 0 0 0 0 1 0 1 0 .
   eq JMP    = 0 0 0 0 1 0 1 1 .
   eq NOP    = 0 0 0 0 0 0 0 0 .

   eq NOPWORD = 0 0 0 0 0 0 0 0
               0 0 0 0 0 0 0 0
               0 0 0 0 0 0 0 0
               0 0 0 0 0 0 0 0 .

endfm

***
*** Functional Units
***
fmod FUNCTIONAL-UNITS is
   protecting MEM .
   protecting REG .
   protecting INSTRUCTION-SET .

   sort FeState .
   sort ExState .
   sort WbState .

*** FETCH OPS
```

```
    op (_,_,_,_) : Mem Word Word Word -> FeState .

    op pm_ : FeState -> Mem .
    ops pc_ cir_ pir_ : FeState -> Word .

    op feu : Int FeState ExState WbState -> FeState .

    op fenext : FeState ExState WbState -> FeState .

*** EXECUTE OPS

    *** Result, Taken Flag, WBFlag, MemWBLoc, RegWBLoc
    op (_,_,_,_,_) : Word Bool Int Word OpField -> ExState .

    op exu : Int FeState ExState WbState -> ExState .

    op exnext : FeState ExState WbState -> ExState .

    ops result_ memwbloc_ : ExState -> Word .
    op taken_ : ExState -> Bool .
    op wbflag_ : ExState -> Int .
    op regwbloc : ExState -> OpField .

*** WRITEBACK OPS

    op (_,_) : Mem Reg -> WbState .

    op dm_ : WbState -> Mem .
    op reg_ : WbState -> Reg .

    op wbu : Int FeState ExState WbState -> WbState .

    op wbnext : FeState ExState WbState -> WbState .

*** FETCH VARIABLES

    var PM : Mem .
    var PC PIR CIR : Word .

    var feS : FeState . *** state
    var T : Int .      *** time

*** EXECUTE VARIABLES

    var TAKEN : Bool .
    var WBFLAG : Int . *** 0 = no writeback, 1 = mem, 2 = reg
    var RESULT MEMWBLOC : Word .
    var REGWBLOC : OpField .

    var exS : ExState . *** state

*** WRITEBACK VARIABLES

    var wbS : WbState . *** state
    var DM : Mem .
    var REG : Reg .

*** FETCH EQUATIONS

    eq pm(PM,PC,CIR,PIR) = PM .
    eq pc(PM,PC,CIR,PIR) = PC .
    eq cir(PM,PC,CIR,PIR) = CIR .
    eq pir(PM,PC,CIR,PIR) = PIR .
    *** iterated map
    eq feu(0,feS,exS,wbS) = feS .
    eq feu(T,feS,exS,wbS) = fenext(feu(T - 1,feS,exS,wbS),exu(T - 1,feS,exS,wbS),wbu(T - 1,feS,exS,wbS)) [owise] .

    ceq fenext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) = (PM,PC + Four,PM[PC],CIR)
       if opcode(cir(PM,PC,CIR,PIR)) =/= JMP .
    ceq fenext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) = (PM,PC + Four,PM[PC],CIR)
       if opcode(cir(PM,PC,CIR,PIR)) == JMP and taken(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) == false .
    ceq fenext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) = (PM,REG[regc(CIR)],PM[REG[regc(CIR)]],CIR
       if opcode(cir(PM,PC,CIR,PIR)) == JMP and taken(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) == true .

*** EXECUTE EQUATIONS

    eq result(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) = RESULT .
    eq taken(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) = TAKEN .
    eq wbflag(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) = WBFLAG .
    eq memwbloc(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) = MEMWBLOC .
    eq regwbloc(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) = REGWBLOC .

    *** iterated map
    eq exu(0,feS,exS,wbS) = exS .
    eq exu(T,feS,exS,wbS) = exnext(feu(T - 1,feS,exS,wbS),exu(T - 1,feS,exS,wbS),wbu(T - 1,feS,exS,wbS)) [owise] .

    *** define instructions

    *** NOP (opcode = 0)
    ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) = (NOPWORD,false,0,NOPWORD,NOP)
       if opcode(cir(PM,PC,CIR,PIR)) == NOP .
    *** ADD (opcode = 1)
    ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
       (REG[rega(CIR)] + REG[regb(CIR)],false,2,NOPWORD,REG[regc(CIR)])
       if opcode(cir(PM,PC,CIR,PIR)) == ADD .
    *** MULT (opcode = 10)
    ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
       (REG[rega(CIR)] * REG[regb(CIR)],false,2,NOPWORD,REG[regc(CIR)])
       if opcode(cir(PM,PC,CIR,PIR)) == MULT .
    *** AND (opcode = 11)
    ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
       (REG[rega(CIR)] & REG[regb(CIR)],false,2,NOPWORD,REG[regc(CIR)])
       if opcode(cir(PM,PC,CIR,PIR)) == AND .
    *** OR (opcode = 100)
    ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
       (REG[rega(CIR)] | REG[regb(CIR)],false,2,NOPWORD,REG[regc(CIR)])
       if opcode(cir(PM,PC,CIR,PIR)) == OR .
    *** NOT (opcode = 101)
    ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
       (!(REG[rega(CIR)]),false,2,NOPWORD,REG[regc(CIR)])
       if opcode(cir(PM,PC,CIR,PIR)) == NOT .
    *** SLL (opcode = 110)
    ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
       (REG[rega(CIR)] << REG[regb(CIR)],false,2,NOPWORD,REG[regc(CIR)])
       if opcode(cir(PM,PC,CIR,PIR)) == SLL .
    *** LD (opcode = 111)
    ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
       (DM[REG[rega(CIR)] + REG[regb(CIR)]],false,2,NOPWORD,REG[regc(CIR)])
       if opcode(cir(PM,PC,CIR,PIR)) == LD .
    *** ST (opcode = 1000)
    ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
       (REG[rega(CIR)] + REG[regb(CIR)],false,1,REG[regc(CIR)],NOP)
```

```
        if opcode(cir(PM,PC,CIR,PIR)) == ST .
    *** EQ (opcode = 1001) [RA == RB]
    ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
        (constzero32,false,2,NOPWORD,REG[regc(CIR)])
        if opcode(cir(PM,PC,CIR,PIR)) == EQ and REG[rega(CIR)] == REG[regb(CIR)] .
    *** EQ (opcode = 1001) [RA =/= RB]
    ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
        (constminus1,false,2,NOPWORD,REG[regc(CIR)])
        if opcode(cir(PM,PC,CIR,PIR)) == EQ and REG[rega(CIR)] =/= REG[regb(CIR)] .
    *** GT (opcode = 1010) [RA > RB]
    ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
        (constzero32,false,2,NOPWORD,REG[regc(CIR)])
        if opcode(cir(PM,PC,CIR,PIR)) == GT and REG[rega(CIR)] gt REG[regb(CIR)] .
    *** GT (opcode = 1010) [RA <= RB]
    ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
        (constminus1,false,2,NOPWORD,REG[regc(CIR)])
        if opcode(cir(PM,PC,CIR,PIR)) == GT and not REG[rega(CIR)] gt REG[regb(CIR)] .
    *** JMP (opcode = 1011) [branch not taken]
    ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
        (constzero32,false,0,NOPWORD,NOP)
        if opcode(cir(PM,PC,CIR,PIR)) == JMP and REG[rega(CIR)] =/= REG[constzero8] .
    *** JMP (opcode = 1011) [branch taken]
    ceq exnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
        (PC + Four,true,2,NOPWORD,REG[regb(CIR)])
        if opcode(cir(PM,PC,CIR,PIR)) == JMP and REG[rega(CIR)] == REG[constzero8] .

*** WRITEBACK EQUATIONS

    eq dm(DM,REG) = DM .
    eq reg(DM,REG) = REG .

    eq wbu(0,feS,exS,wbS) = wbS .
    eq wbu(T,feS,exS,wbS) = wbnext(feu(T - 1,feS,exS,wbS),exu(T - 1,feS,exS,wbS),wbu(T - 1,feS,exS,wbS)) [owise] .

    ceq wbnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
        (DM[result(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) / memwbloc(RESULT,TAKEN,WBFLAG,MEMWBLOC,REG
        if wbflag(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) == 1 .
    ceq wbnext((PM,PC,CIR,PIR),(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC),(DM,REG)) =
        (DM,REG[result(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) / regwbloc(RESULT,TAKEN,WBFLAG,MEMWBLOC,
        if wbflag(RESULT,TAKEN,WBFLAG,MEMWBLOC,REGWBLOC) == 2 .

endfm

***
*** Pipeline
***
*** This is the "main" function, where we define the state funtion pmp and the
*** next-state function pnext.
***
fmod PMP is
     protecting FUNCTIONAL-UNITS .

    sort PState .

    op (_,_,_) : FeState ExState WbState -> PState .
    op pmp : Int PState -> PState .
    op pnext : PState -> PState .

    op fetchunit_ : PState -> FeState .
    op executeunit_ : PState -> ExState .
    op writebackunit_ : PState -> WbState .

    var FETCHUNIT : FeState .
    var EXECUTEUNIT : ExState .
    var WRITEBACKUNIT : WbState .

    var PMP : PState .
    var Time : Int .

    eq fetchunit(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT) = FETCHUNIT .
    eq executeunit(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT) = EXECUTEUNIT .
    eq writebackunit(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT) = WRITEBACKUNIT .

    eq pmp(0,PMP) = PMP .
    eq pmp(Time,PMP) = pnext(pmp(Time - 1,PMP)) [owise] .

    eq pnext(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT) =
       (fenext(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT),
       exnext(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT),
       wbnext(FETCHUNIT,EXECUTEUNIT,WRITEBACKUNIT)) .

endfm

***
*** The final module is to define an actual program and run it.
***
fmod RUNPROGS is
    protecting PMP .       *** import the microprocessor representation

    ops Dm Pm : -> Mem .
    op Rg : -> Reg .
    ops Pc Pir Cir Res : -> Word .
    op Tk : -> Bool .
    op Wflag : -> Int .
    op Wmemloc : -> Word .
    op Wregloc : -> OpField .

    *** Set the values to zero
    eq Pc = constzero32 .
    eq Cir = constzero32 .
    eq Pir = constzero32 .
    eq Res = constzero32 .
    eq Wmemloc = constzero32 .
    eq Tk = false .
    eq Wflag = 0 .
    eq Wregloc = constzero8 .

    *** R1 = 1
    eq Rg[0 0 0 0 0 0 0 1] =
        0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 1 .

    *** R2 = 0
    eq Rg[0 0 0 0 0 0 1 0] =
        0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 .
```

```
*** R13 = 252
eq Rg[0 0 0 0 1 1 0 1] =
      0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0
      1 1 1 1 1 1 0 0 .

*** Mem[1] = 6
eq Dm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1] =
      0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0
      0 0 0 0 0 1 1 0 .

*** Mem[2] = 5
eq Dm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0] =
      0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0
      0 0 0 0 0 1 0 1 .

*** INST 1 [ Load (RG1+RG1) into RG3 ] -> Mem[2] = 5
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] =
      0 0 0 0 0 1 1 1 *** LOAD
      0 0 0 0 0 0 0 1 *** RG1
      0 0 0 0 0 0 0 1 *** RG1
      0 0 0 0 0 0 1 1 . *** RG3

*** INST 2 [ Load (RG1+RG2) into RG4 ] -> Mem[2] = 5
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0] =
      0 0 0 0 0 1 1 1 *** LOAD
      0 0 0 0 0 0 0 1 *** RG1
      0 0 0 0 0 0 0 1 *** RG1
      0 0 0 0 0 1 0 0 . *** RG4

*** INST 3 [ Shift left R3 by R4 and store in R5 ]
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0] =
      0 0 0 0 0 1 1 0 *** SHIFTL
      0 0 0 0 0 0 1 1 *** RG3
      0 0 0 0 0 1 0 0 *** RG4
      0 0 0 0 0 1 0 1 . *** RG5

*** INST 4 [ Store RG5 in Mem[RG2+RG3] ] -> Mem[5] = 160
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0] =
      0 0 0 0 1 0 0 0 *** STORE
      0 0 0 0 0 0 1 0 *** RG2
      0 0 0 0 0 0 1 1 *** RG3
      0 0 0 0 0 1 0 1 . *** RG5

*** INST 5 [ Add RG3 and RG4 and store in RG6 ] -> RG[6] = 10
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0] =
      0 0 0 0 0 0 0 1 *** ADD
      0 0 0 0 0 0 1 1 *** RG3
      0 0 0 0 0 1 0 0 *** RG4
      0 0 0 0 0 1 1 0 . *** RG6

*** INST 6 [ Mult RG3 by RG4 and store in RG7 ] -> RG[7] = 25
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0] =
      0 0 0 0 0 0 1 0 *** MULT
      0 0 0 0 0 0 1 1 *** RG3
      0 0 0 0 0 1 0 0 *** RG4
      0 0 0 0 0 1 1 1 . *** RG7

*** INST 7 [ Bitwise and of RG3 and RG4, stored in RG8 ] -> RG[8] = 5
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0] =
      0 0 0 0 0 0 1 1 *** AND
      0 0 0 0 0 0 1 1 *** RG3
      0 0 0 0 0 1 0 0 *** RG4
      0 0 0 0 1 0 0 0 . *** RG8

*** INST 8 [ Bitwise or of RG3 and RG4, stored in RG9 ] -> RG[9] = 5
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0] =
      0 0 0 0 0 1 0 0 *** OR
      0 0 0 0 0 0 1 1 *** RG3
      0 0 0 0 0 1 0 0 *** RG4
      0 0 0 0 1 0 0 1 . *** RG9

*** INST 9 [ Inverse of RG3, stored in R10 ]
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0] =
      0 0 0 0 0 1 0 1 *** NOT
      0 0 0 0 0 0 1 1 *** RG3
      0 0 0 0 0 0 0 0 *** RG0
      0 0 0 0 1 0 1 0 . *** RG10

*** INST 10 [ Test if RG7 = RG8 ] -> RG[7] = 25, RG[8] = 5, Answer = -1 (false)
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0] =
      0 0 0 0 1 0 0 1 *** EQ
      0 0 0 0 0 1 1 1 *** RG7
      0 0 0 0 1 0 0 0 *** RG8
      0 0 0 0 1 0 1 1 . *** R11

*** INST 11 [ Test if R11 == 0 ] -> false, no jump
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0] =
      0 0 0 0 1 0 1 1 *** JMP
      0 0 0 0 1 0 1 1 *** R11
      0 0 0 0 1 1 0 0 *** R12
      0 0 0 0 1 1 0 1 . *** R13

*** INST 12 [ Test if RG7 > RG8 ] -> RG[7] = 25, RG[8] = 5, Answer = 0 (true)
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0] =
      0 0 0 0 1 0 1 0 *** GT
      0 0 0 0 0 1 1 1 *** RG7
      0 0 0 0 1 0 0 0 *** RG8
      0 0 0 0 1 0 1 1 . *** RG11

*** INST 13 [ Test if RG11 == 0 ] -> true, jump to R13, store PC+4 in RG12
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0] =
      0 0 0 0 1 0 1 1 *** JMP
      0 0 0 0 1 0 1 1 *** RG11
      0 0 0 0 1 1 0 0 *** RG12
      0 0 0 0 1 1 0 1 . *** RG13

*** INST 14 [ Subroutine ][ Add RG3 to RG4 and store in RG14 ] -> R[14] = 10
eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0] =
      0 0 0 0 0 0 0 1 *** ADD
      0 0 0 0 0 0 1 1 *** RG3
      0 0 0 0 0 1 0 0 *** RG4
      0 0 0 0 1 1 1 0 . *** RG14
```

```
    *** INST 15 [ Test if R11 == 0] -> True, jump back to R[12]
    eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0] =
         0 0 0 0 1 0 1 1 *** JMP
         0 0 0 0 1 0 1 1 *** RG11
         0 0 0 0 1 1 1 1 *** RG15
         0 0 0 0 1 1 0 0 . *** RG12

    *** INST 16 [ Return from subroutine ] [ Add R7 to R8 and store in R16 ] -> R[16] = 25 + 5 =30
    eq Pm[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0] =
         0 0 0 0 0 0 0 0 *** ADD
         0 0 0 0 0 1 1 1 *** RG7
         0 0 0 0 1 0 0 0 *** RG8
         0 0 0 1 0 0 0 0 . *** R16
endfm

***
*** Now run the program
***
reduce pmp(1, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(2, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(3, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(4, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(5, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(6, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(7, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(8, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(9, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(10, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(11, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(12, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(13, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(14, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(15, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
reduce pmp(16, ((Pm,Pc,Cir,Pir),(Res,Tk,Wflag,Wmemloc,Wregloc),(Dm,Rg))) .
q
```