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

Initial Document

Sean Handley, October 2006

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, multi-core, superscalar and multi-threaded models.

Project Initial Document submitted to Swansea University
in Partial Fulfilment for the Degree of Bachelor of Science

Department of Computer Science
University of Wales Swansea

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. I shall begin with a programmer's view of the machine, based heavily on the one I implemented as part of the level two system specification course (see *Appendix A*), then proceed to develop pipelined, multi-core, super-scalar and multi-threaded models.

2 Literature Survey

2.1 "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.

2.2 "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.

2.3 "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.

2.4 "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.

2.5 "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.

2.6 "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.

2.7 "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.

2.8 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.

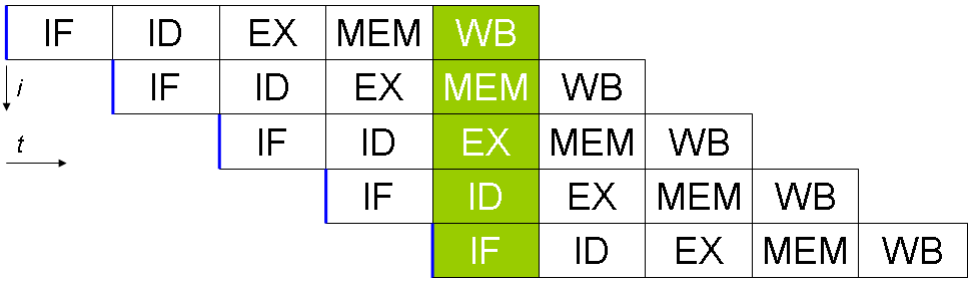
3 Background

3.1 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]

3.1.1 Pipelined Processors

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*[12]

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

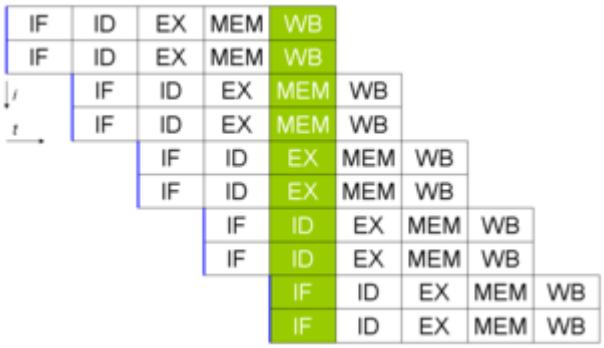
```
Add the value of R1 to the value of R2 and store the result in R3
Subtract the value of R4 from R3 and store the result in R5
```

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.

3.1.2 Superscalar Processors

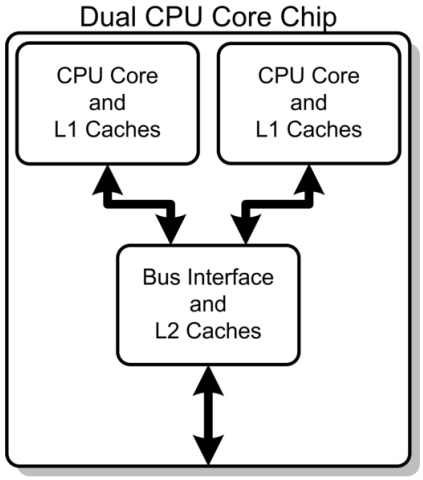
A *superscalar* processor is one which makes use of multiple pipelines in order to achieve superior throughput. In practice, this is achieved by duplication of several hardware components. As with pipelining, the use of superscalar processing introduces a number of complications.

[1]To borrow a term from electronics.



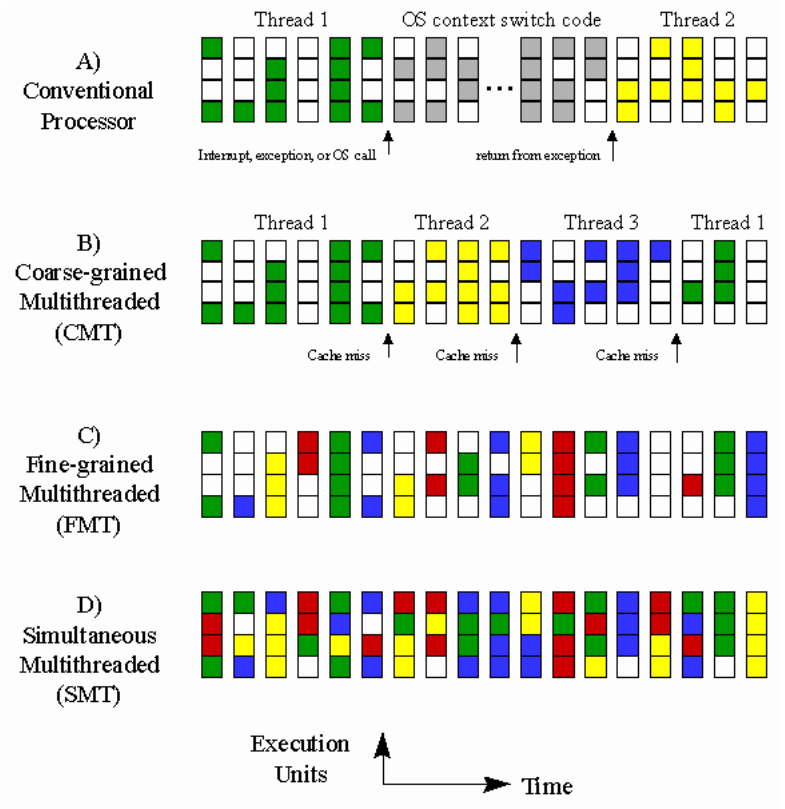
3.1.3 Multi-core Processors

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



3.1.4 Multi-threaded Processors

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.



3.2 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.

3.2.1 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].

3.2.2 Streams

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

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

3.2.3 Iterated Maps

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

$$F : T \times A \rightarrow A$$

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:

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

3.2.4 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*.

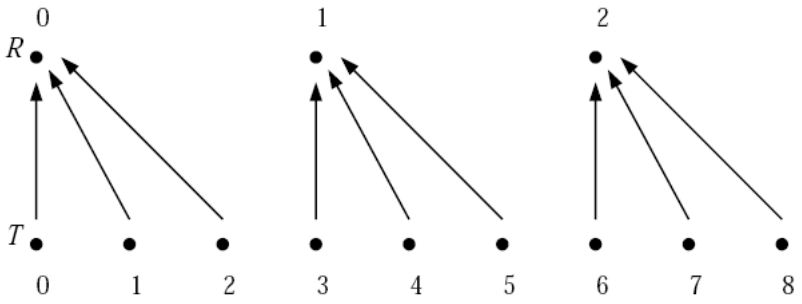
3.2.5 Comparing Iterated Maps

Iterated maps may be compared using the following commutative diagram:

$$\begin{array}{ccc} T \times A & \xrightarrow{F} & A \\ \uparrow{\scriptstyle (\lambda,\psi)} & & \uparrow{\scriptstyle \psi} \\ S \times B & \xrightarrow{G} & B \end{array}$$

3.2.6 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* $\bar{\lambda} : R \to T$,

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

4 Aims of this Project

- Develop methods to model each aspect of a microprocessor in Maude
- Use these methods to produce models at varying levels of abstraction
- Develop a way to make Maude's output more legible and managable

5 Main Methods and Tools

Initially, my microprocessor models will be developed mathematically on paper. Once a working model has been finished, it can be expressed using the algebraic specification language, Maude.

Here is a list of concrete design decisions:

- 32-bit word
- Load-store Architecture (i.e. all values are loaded into registers prior to processing)
- 255 general purpose registers (register zero is bound to constant zero)
- Positive integer types only (no floating point or negatives)
- A simple RISC-style instruction set

  `ADD MULTIPLY GREATERTHAN AND OR NOT`

  `SHIFTLEFT EQUALS JUMP LOAD STORE`
- Simple addressing mode
- Simple instruction format i.e. `INST OP [OP] [OP]`
- Simple 3-stage pipeline
- Simple branch-prediction (always taken)
- Two cores for the multicore

5.1 Maude

Maude is a *specification language* which allows a system to be modelled algebraically[9]. 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]
```

5.2 Java/Ruby

Since Maude's output can be somewhat cryptic (particularly when using a binary number system), I believe it will be beneficial to the project if I produce a program to which Maude's output can be piped. This program will convert binary numbers to decimal numbers and help reduce the time needed to test the models. For no particular reason, other than a knowledge of the language, I have decided to use Java[10] to achieve this. However, there is a possibility I will use Ruby[11] instead, as it has excellent pattern matching methods. The ultimate decision on this is dependent upon the progress of the microprocessor models themselves.

6 Project Plan (for first 8 weeks)

- MICHAELMAS TERM
- Week 1 (October 2nd - October 8th)
  - Initial meeting with Neal to discuss the project.
  - Continue to accumulate sources of information.
  - Begin writing up the background for Algebras/Microprocessors.
- Week 2 (October 9th - October 15th)
  - Finish writing up the background.
  - Finish writing literature review.
  - Elaborate upon scientific questions and technical problems.
  - Discuss main methods and tools and reasons for choice.
- Week 3 (October 16th - October 22nd)
  - Finalise the initial document and tie up loose ends.
  - Give to Neal to check over.

  DEADLINE: Initial document, Friday 20th.
- Week 4 (October 23rd - October 29th)
  - Go back over the CS-213 microprocessor coursework
  - Improve it and re-write the binary arithmetic to do negative numbers and FP
  - Get a working programmer's model.

- Week 5 (October 30th - November 5th)
  - Start writing the pipelined version.
- Week 6 (November 6th - November 12th)
  - Begin putting presentation together.
- Week 7 (November 13th - November 19th)
  - Finalise presentation.
- Week 8 (November 20th - November 26th)
  - Gregynog (20th-22nd)

  DEADLINE: Give presentation on 20th or 21st.

  - Continue work on the pipelined version.

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] Intel's White Papers, The Intel Technology Journal (1998 - 2006).

[9] "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.

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

[11] "Programming Ruby: The Pragmatic Programmer's Guide", by Dave Thomas, Chad Fowler, Andy Hunt, 2004.

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

A The original programmer's model

```
***
*** "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 _++8_ : 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 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 .

*** Add 8 [not yet implemented]
eq S ++8 T = S ++ T .

*** 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 +8 W = bits(V ++8 W,31,0) .
eq A +8 B = bits(A ++8 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 ADD8 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 ADD8 = 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 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 / 0][O] = constzero32 if O == constzero8 .
eq REG[A / 0][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 .
*** ADD8 (opcode = 1)
ceq next(MP,MD,PC,REG) = (MP, MD, PC + Four,
REG[REG[rega(MP[PC])] +8 REG[regb(MP[PC])] / regc(MP[PC])])
if opcode(MP[PC]) == ADD8 .
```

```
*** 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 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 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 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 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 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 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 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 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 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 ][ Add8 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 0 1 1 1 1 1 1 0 0] =
0 0 0 0 0 0 0 1 *** ADD8
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
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 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
```