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

Interim Document

Sean Handley, February 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, multi-core, superscalar and multi-threaded models.

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

Department of Computer Science
University of Wales Swansea

Contents

Note: This document assumes the reader has seen the initial document. This can be found here: http://sucs.org/~talyn256/init.pdf

1 Progress Update

When I first began this project in October 2006, I planned to produce 4 microprocessor specifications - a pipelined implementation, a multicore implementation, a superscalar implementation and a multithreaded implementation. This, naturally, lead to four milestones. The first of these, the pipelined microprocessor, I had intended to complete before January.

Currently, the project is behind schedule. The pipelined implementation is almost finished, several weeks later than I intended, therefore the second of my milestones, the multicore implementation, has slipped behind. Writing all four specifications, as ambitiously proposed in October, is probably not feasible in the given time. I am, therefore, going to focus on the pipelined and multicore implementations and attempt work on the superscalar and multithreaded implementations if there is time.

The pipelined implementation has proven complex in subtle ways which I had not previously considered. My initial design decisions have proven to be somewhat naïve and it was only when I engaged in writing the specification that my oversights became clear. I will go into detail of the pipelined specification in the next section and discuss how it works now in contrast to how I imagined it would at the beginning of the project. The subtle differences were largely due to organisation of module substructure and the decision of which components to model and which to leave out. A bitwise specification of every single logic gate would be rather too low a level of abstraction to be practical, yet too high a level makes designing the inner workings and interactions of the functional units rather difficult. Finding the proper level of abstraction has definitely proven to be the greatest challenge so far.

My output parsing program has reached a basic prototype stage in Java but, since it is a simple utility, it can (and probably will) be completely rewritten in Ruby, as considered earlier in the project. This means that the process of executing a test program on the microprocessor specification occurrs in the following fashion:

- Maude is invoked from the command line with the location of the specification file given as the target.
- Maude performs the reductions specified at the end of the model and reports the state of the machine at each time interval.
- This output is piped into the parsing program, along with a flag denoting which of the specifications is being executed[1].
- The parsing program determines the state of components at given times by pattern matching. It converts bitstreams to integer values and produces a simple output in a form similar to the following: *Time 1: Program counter = 9384, Memory(1) = 32,...*

[1] It would be more elegant to produce a parsing program which could deal with any specification output without meta info. However, this is sufficiently complex to make doing so an unnecessary effort. It is much quicker to define the parsing rules statically.

- This data can then be piped into a file, saved, and examined to see if the predicted behaviour matched the actual behaviour.

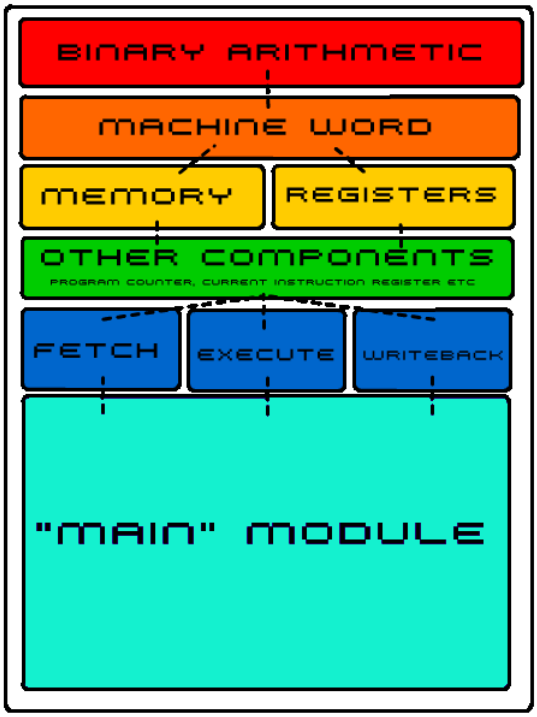
2 The Pipelined Implementation

2.1 An Overview

As mentioned in the previous section, finding the proper level of abstraction has been the largest challenge in the writing of the specification. My initial ideas were rather rooted in a procedural approach to programming, therefore producing a working specification required me to re-think many of my earlier plans. Writing the specification involved, essentially, a refactoring of the original programmer's view model. The binary arithmetic, machine word, memory and register modules were all reusable with the largest changes being made in the "main" method. This involved separating out the execution logic into a module representing the Execution Unit.

Care was required to have each functional unit capable of "seeing" the components it needed, such as the program counter, memory and registers. Also, the instruction set was placed into its own module as a move to further enhance the organisation of the model. Also, several new components have been added to the basic RISC model in order to facilitate pipelining. These include a Current Instruction Register, a Previous Instruction Register and a Stall flag.

The current structural design of the pipelined processor is shown below:



- **Binary Module** - Rules for basic binary arithmetic operations, such as addition and multiplication.
- **Machine Word** - Defines the 32-bit word structure and provides a method for instructions to seperate the operator from its operands.
- **Memory/Registers** - Facilitate depiction of a memory hierarchy by using arrays of words.
- **Other Components** - This module contains components which the "main" method and functional units need, such as the program counter, special registers etc.
- **Functional Units** - Each functional unit here represents a pipeline stage.

  Fetch gets the next instruction indicated by the program counter and places it in the current instruction register.

  Execute carries out the instruction found in the current instruction register.

  Writeback writes the results of the instruction to the target registers.
- **"Main Method"** - This is where the functional units and components are coordinated by a retiming function which correlates the states of each functional unit at a given time.

2.2 Dealing With 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*[2] *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.

[2]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.

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 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*.

2.3 Code Excerpts

The following Maude code demonstrates key sections of the preliminary pipelined implementation.

```
***
*** 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 .

    *** define the opcodes
    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 .
    eq NOP     = 0 0 0 0 0 0 0 1 .

    eq NOPWORD = 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 .

endfm

***
*** State of PMP, together with tupling and projection functions
***
fmod PIPELINE-STATE is
    protecting MEM .
    protecting REG .

    sort Pstate .

    *** Tuple representing the microprocessor state:
    ***
    *** - Program memory
    *** - Data memory
    *** - PC
    *** - Current Instruction Register
    *** - Previous Instruction Register
    *** - Register values
    *** - Stall Flag
    op (_,_,_,_,_,_,_) : Mem Mem Word Word Word Reg Bool -> Pstate .

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

    *** project out PC, CIR and PIR
    op pc_ cir_ pir_ : Pstate -> Word .
```

```
    *** project out regs
    op reg_ : Pstate -> Reg .

    *** project out stall
    op stall_ : Pstate -> Bool .

    var S : Pstate .
    vars MP MD : Mem .
    vars PC CIR PIR: Word .
    var REG : Reg .
    var STALL : Bool .

    *** tuple member accessor functions
    eq mp(MP,MD,PC,CIR,PIR,REG,STALL) = MP .
    eq md(MP,MD,PC,CIR,PIR,REG,STALL) = MD .
    eq pc(MP,MD,PC,CIR,PIR,REG,STALL) = PC .
    eq cir(MP,MD,PC,CIR,PIR,REG,STALL) = CIR .
    eq pir(MP,MD,PC,CIR,PIR,REG,STALL) = PIR .
    eq reg(MP,MD,PC,CIR,PIR,REG,STALL) = REG .
    eq stall(MP,MD,PC,CIR,PIR,REG,STALL) = STALL .

    op Four : -> Word .

    op pmp : Int Pstate -> Pstate .

    op next : Pstate -> Pstate .

    var S : Pstate .
    var T : Int .
    var MP MD : Mem .
    vars PC CIR PIR A : Word .
    var REG : Reg .
    var O P : OpField .
    var STALL : Bool .

    *** Fix the zero register to always = 0
    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].

    eq STALL = false .

endfm

endfm

*** ====== BEGIN PIPELINE FUNCTIONAL UNITS ====== ***

***
*** Fetch Unit State with projection
```

```
***
fmod FETCH-UNIT-STATE is
    protecting MACHINE-WORD .

    sort FeState .

    op (_) : Word -> FeState .

    var INSTRUCTION : Word .

    *** project out instructions
    op instruction_ : FeState -> Word .
endfm

***
*** Fetch Unit
***
*** Fetches the next instruction from program memory.
***
fmod FETCH-UNIT is
    protecting FETCH-UNIT-STATE .
    protecting PIPELINE-STATE .
    protecting INSTRUCTION-SET .

    op feu : Int FeState -> FeState .

    op fenext : FeState -> FeState .

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

    *** iterated map
    eq feu(0,feS) = feS .
    eq feu(feT,feS) = fenext(feu(feT - 1,feS)) [owise] .

    ceq fenext(INSTRUCTION) = MP[PC] if STALL(MP[PC]) == false .
    eq fenext(INSTRUCTION) = NOPWORD .

endfm

***
*** Execution Unit State with projection
***
fmod EXECUTION-UNIT-STATE is
    protecting MACHINE-WORD .

    sort ExState .

    op (_) : Word -> ExState .
```

```
    var RESULT : Word .

    *** project out results
    op result_ : ExState -> Word .

endfm

***
*** Execution Unit
***
*** Executes the current instruction.
***
fmod EXECUTION-UNIT is
    protecting EXECUTION-UNIT-STATE .
    protecting PIPELINE-STATE .
    protecting INSTRUCTION-SET .

    op exu : Int ExState -> ExState .

    op exnext : ExState -> ExState .

    var exS : ExState . *** state
    var exT : Int .     *** time

    *** iterated map
    eq exu(0,exS) = exS .
    eq exu(exT,exS) = exnext(exu(exT - 1,exS)) [owise] .

    *** define instructions

    *** ADD (opcode = 0)
    ceq exnext(RESULT) = (REG[rega(currentInstruction)]
            + REG[regb(currentInstruction)])
        if opcode(currentInstruction) == ADD .
    *** MULT (opcode = 10)
    ceq exnext(RESULT) = (REG[rega(currentInstruction)]
            * REG[regb(currentInstruction)])
        if opcode(currentInstruction) == MULT .
    *** AND (opcode = 11)
    ceq exnext(RESULT) = (REG[rega(currentInstruction)]
            & REG[regb(currentInstruction)])
        if opcode(currentInstruction) == AND .
    *** OR (opcode = 100)
    ceq exnext(RESULT) = (REG[rega(currentInstruction)] |
            REG[regb(currentInstruction)])
        if opcode(currentInstruction) == OR .
    *** NOT (opcode = 101)
    ceq exnext(RESULT) = (!(REG[rega(currentInstruction)]))
        if opcode(currentInstruction) == NOT .
    *** SLL (opcode = 110)
```

```
    ceq exnext(RESULT) = (REG[rega(currentInstruction)]
            << REG[regb(currentInstruction)])
        if opcode(currentInstruction) == SLL .
    *** LD (opcode = 111)
    ceq exnext(RESULT) = (REG[rega(currentInstruction)]
            + REG[regb(currentInstruction)])
        if opcode(currentInstruction) == LD .
    *** ST (opcode = 1000)
    ceq exnext(RESULT) = (REG[rega(currentInstruction)]
            + REG[regb(currentInstruction)])
        if opcode(currentInstruction) == ST .
    *** EQ (opcode = 1001) [RA == RB]
    ceq exnext(RESULT) = (REG[constzero32])
        if opcode(currentInstruction) == EQ
                and REG[rega(currentInstruction)] == REG[regb(currentInstruction)] .
    *** EQ (opcode = 1001) [RA =/= RB]
    ceq exnext(RESULT) = (REG[constminus1])
        if opcode(currentInstruction) == EQ
                and REG[rega(currentInstruction)] =/= REG[regb(currentInstruction)] .
    *** GT (opcode = 1010) [RA > RB]
    ceq exnext(RESULT) = (REG[constzero32])
        if opcode(currentInstruction) == GT
                and REG[rega(currentInstruction)] gt REG[regb(currentInstruction)] .
    *** GT (opcode = 1010) [RA <= RB]
    ceq exnext(RESULT) = (REG[constminus1])
        if opcode(currentInstruction) == GT
                and not (REG[rega(currentInstruction)] gt REG[regb(currentInstruction)]) .
    *** JMP (opcode = 1011) [branch not taken]
    ceq exnext(RESULT) = (REG[constminus1])
        if opcode(currentInstruction) == JMP
                and REG[rega(currentInstruction)] =/= REG[constzero8] .
    *** JMP (opcode = 1011) [branch taken]
    ceq exnext(RESULT) = (REG[constzero32])
        if opcode(currentInstruction) == JMP
                and REG[rega(currentInstruction)] == REG[constzero8] .

endfm
```

3 Thoughts on the Multicore Implementation

Having written the pipelined specification, I feel to have gleaned several valuable insights with respect to modelling processors algebraically. I feel this experience will help me write the multi-core specification with far fewer problems. I envisage the model will build upon the pipelined model by encapsulating two of the old "main" methods within a new "main" method. This new module will deal with retimings, as the old modules did, as well as a basic notion of process scheduling. This is, technically, a task for the operating system. However, it's necessary to model this type of input in order to achieve a more realistic model. As such, this will require a re-write of the test program to encompass multiple

processes or, more simply, a second test program which the operating system process scheduler can dispatch to the second of the cores.

In addition to these changes, it will be necessary to reconsider the memory hierarchy by altering the architecture to encorporate the idea of caches. This, potentially, makes things rather more complex. However, multicore processors do share caches at some level, therefore it is necessary to accommodate this by definition.

4 Revised Schedule

- 14th February
  - Begin the dissertation write-up
  - Start writing the Ruby output parser
- 21st February
  - Finish the multicore implementation
  - Finish the Ruby parser
  - Begin the superscalar implementation
- 21st March
  - Finish the superscalar implementation
  - Begin the multithreaded implementation
- 21st April
  - Finish the multithreaded implementation
  - Finish the dissertation write-up
- Early May
  - Demonstration and Viva