A Simulation Applet for Microcoding Exercises

Roland N. Ibbett
Institute for Computing Systems Architecture
School of Informatics, University of Edinburgh
Edinburgh EH9 3JZ, UK

Abstract

At the University of Edinburgh we have used a Hierarchical Computer Architecture design and Simulation Environment (HASE) to build a number of architectural models for use in research and teaching. Within HASE, the JavaHASE facility allows models to be translated into applets which can be accessed via the WWW.

The Edinburgh Microcodable Microprocessor Applet (EMMA) was created in response to a need to provide students with a reliable practical experiment on processor design in a Computer Design course. There are currently two versions, a Basic model that can execute single-cycle arithmetic operations and an Enhanced model that can also execute multiply and divide. The Basic model was used successfully by a class of about 80 students in 2003.

I. INTRODUCTION

At the University of Edinburgh we have used a Hierarchical Computer Architecture design and Simulation Environment (HASE) to build a number of architectural models for use in research and teaching. Within HASE, the JavaHASE facility [1] allows models to be translated into applets which can be accessed via the WWW[1]. JavaHASE applets are programmable simulation models with visualisation capabilities that allow activities taking place within the model (data movements, state changes, register/memory content changes, etc) to be displayed on-screen dynamically. Models of Tomasulo's algorithm and the DLX [2] and DASH [3] architectures and, most recently, a microcodable processor (the Edinburgh Microcoded Microprocessor Applet (EMMA)), are currently being used in teaching at Edinburgh. Each model is supported by a web site describing the architecture and the model.

In order for students to be able to carry out exercises using the models, the applets require access to cut and paste facilities, using the clipboard, on the client machine. Although standard security managers for applets do not allow access to the clipboard, the security manager can be configured using a Java policy file to allow clipboard access to applets from specified URLs.

According to Wolffe *et al*'s classification [4], most of the JavaHASE DLX applets [5] are Enhanced Microarchitecture Simulators. These applets are designed to show students what happens inside a processor as programs are executed in a simple pipelined system, in a system with a scoreboard and in a dual-issue (VLIW) system with predication. In a practical exercise involving the DLX with Scoreboard applet shown in Figure 1, for example, students are given an assembly code sequence representing a simple implementation of a scalar (dot) product loop and are asked to run the simulation and note where hazards occur. They are then asked to reorder the code to eliminate or at least reduce the effects of these hazards. As a further optimisation they are asked to unroll the loop to include two iterations of the algorithm in one program loop.

The JavaHASE Tomasulo's algorithm applet (Figure 2) is both an Enhanced Microarchitecture Simulator and a Historical Machine Simulator in Wolffe *et al*'s classification, since it models closely the original system used in

[1] www.icsa.inf.ed.ac.uk/research/groups/hase/

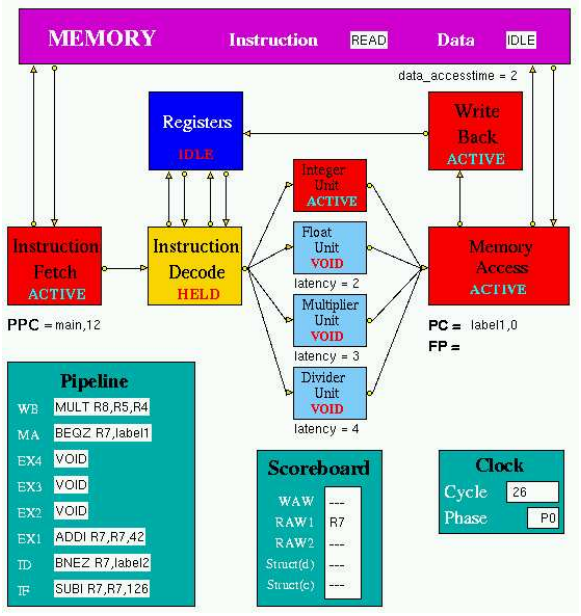

Figure 1: DLX with Scoreboard

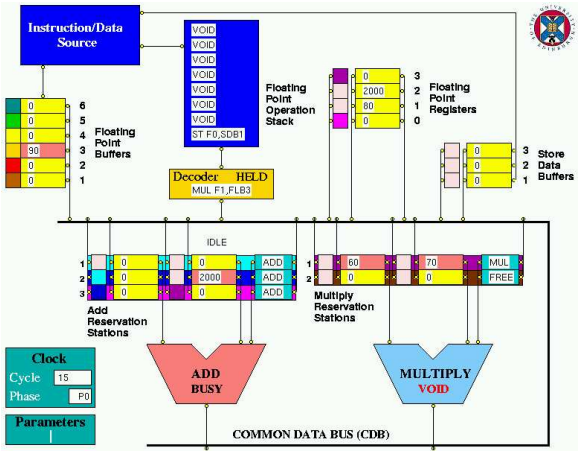

Figure 2: Tomasulo's Algorithm Applet

the IBM System/360 Model 91 Floating-point Unit [6]. The algorithm is difficult to explain to students without a dynamic demonstration, so the applet is designed to show, in particular, how the tags move around the system during program execution. The 360 processor and memory are represented in the model by an Instruction/Data Source Unit which stores a sequence of instructions and a set of data values. These are sent to the Floating-point Operation Stack in sequence. The website explains the operation of the algorithm in terms of the sequence of instructions contained in the applet when it is downloaded but instructors and students can load their own code and data into the applet's memories.

The JavaHASE DASH Cluster Model is also, in a sense, a Historical Machine Simulator, though it was designed to show the operation of the DASH snoopy bus cache coherence protocol rather than to be a complete historical model and is therefore better thought of as a Multi-Processor Simulator in Wolffe *et al*'s classification scheme. The cluster consists of four nodes attached via a cluster bus to a memory. Each node contains a primary cache, a secondary cache and processor that is simply modelled as a source of addresses.

In a practical exercise, students are asked to display the contents of the processors (*i.e.* the lists of addresses) and the caches and, by running the animation in single shot mode, to observe what happens as the simulation proceeds. They are then asked to submit listings of the addresses in each of the four processors, annotated to show the responses of the caches to each access.

EMMA was created in response to a need to provide students with a reliable practical experiment on processor design in a Computer Design course and falls into Wolffe *et al*'s Simple Hypothetical Machine Simulator category. It exists in two versions, the Basic version (EMMA-1) can execute single-cycle arithmetic operations whilst the Enhanced version (EMMA-2) can also execute multiply and divide. Of the simulators identified in [4], only the MicroArchitecture Simulator[2] appears to offer similar facilities to EMMA but it is not web-based and requires the use of MacOS. Most currently available processor simulation applets are targeted at demonstrating processor operation at the register transfer level by allowing students to enter their own assembly code, *e.g.* the Little Man Computer[3] and cpu-sim[4]. cpu-sim shares some characteristics with EMMA in that it shows the movement of information inside a processor by means a 'worm' moving along the data paths

[2] www.dslextreme.com/users/fabrizioo/msim.html
[3] www.acs.ilstu.edu/faculty/javila/lmc/
[4] www.cs.gordon.edu/courses/cs111/

as assembly code instructions are executed, but the micro-execution sequence for each instruction is predefined within the model.

II. EMMA

A. *Processor Architecture*

EMMA is a load/store, register-register arithmetic processor implemented as shown in Figure 3. It was designed with simplicity and elegance in mind but was nevertheless intended to give students a feel for issues which can arise in the design of real systems. It uses a Harvard architecture, with separate instruction and data memories. This is not only convenient from a simulation perspective (instructions are represented using a C++ struct which allows them to be displayed in readable assembly code format whilst data is represented in 32-bit integer format) but also reflects the use of separate instruction and data caches in most real microprocessors. The microcode word is also 32 bits, divided up into eight hex characters, with one or more hex characters being assigned to each unit.

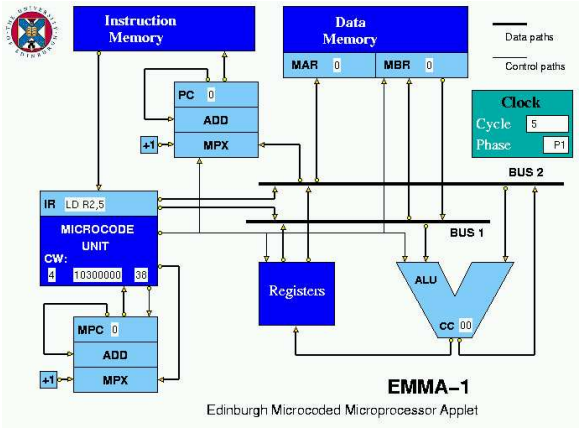

Figure 3: Basic EMMA

The units making up the processor itself are two data buses (BUS 1 and BUS 2), an ALU, a Register Unit (containing 16 32-bit registers, with R0 always set to 0), a Program Counter, a Microcode Program Counter and a Microcode Unit (which also contains the Microcode Memory). The Program Counter (PC) is, in effect, the memory address register for the Instruction Memory with the Instruction Memory buffer register being the Instruction Register (IR) in the Microcode Unit. The Data Memory has built-in Memory Address and Memory Buffer Registers (MAR and MBR) connected to BUS 1 and BUS 2 respectively.

EMMA operates on a two phase clock (shown as P0 and P1 in the Clock entity display). In clock cycles in which they are active, each unit executes its internal actions in the first phase of the clock (P0) and sends out a result packet in the second phase (P1). The Microcode Unit, for example, reads its microcode memory in P0 and sends the appropriate microcode fields to other units, if they are to be activated, in P1.

B. *Instruction Set*

The instruction set is prefined in the applets. In the Basic version it includes absolute jumps (JUMP and JREG) and relative branches (BEQZ and BNEG), loads (LD, LDL, LDX) and stores (ST and STX), register-register operations (ADD, SUB, AND, OR, XOR, SLL, SRL, SRA) and register-literal arithmetic operations (ADDL, etc), as shown in Table 1, and a STOP instruction, which stops the simulation. RD is the destination register, RS, RS1 and RS2 are source registers and L is a Literal (immediate) operand.

JUMP	PC = L
JREG	PC = (RS)
BEQZ	PC = PC + L if ALU = 0
BNEG	PC = PC + L if ALU -ve
LD	RD = Memory(RS)
LDL	RD = L
LDX	RD = Memory(RS + L)
ST	Memory(RD) = RS
ADD	RD = RS1 + RS2
ADDL	RD = RS1 + L

Table 1: Basic Instruction Set

The Enhanced version also includes multiply (MUL, MULL) and divide (DIV, DIVL) operations and two undefined register-register operations (OP1 and OP2) which can be used for functions of the student's own choice, *e.g.* RD = RD + (RS1 * RS2).

C. *Processor Design*

1) *Microcode Unit:* The Microcode Unit has one input from the Instruction Memory and one from the Microprogram Counter (MPC). (There is also an input, not shown in the display, carrying the Condition Code from the ALU.) The microcode memory contained in the Microcode Unit is addressed by the function field of the Instruction Register in the clock cycle in which a new instruction is received from the Instruction Memory and by the Microcode Program Counter (MPC) in subsequent clock cycles. (Erroneous microcode can cause both to occur simultaneously; this is automatically detected and displayed as an error.)

The Microcode Unit has an output control path to each of the units it controls and three output data paths, one to the MPC and one to each of the buses. These data paths are activated by bits in the most significant hex character in the microcode word (Table 2). This position was chosen because it leaves the most significant bit unused and thus avoids problems with negative numbers.

The second hex character controls conditional execution. The Microcode Unit always sends a control code to the MPC but only sends to the other units if

- neither conditional bit is set
- the *execute if condition is true* bit is set and the condition is true
- the *execute if condition is false* bit is set and the condition is false

Conditional branches can thus be implemented by executing either a microcode instruction that adds the Literal operand to PC or an instruction that adds +1 to PC.

2) *PC and MPC:* The Program Counter and Microprogram Counter units behave identically. They contain the relevant register together with an adder which receives one input from the register itself and the other from a multiplexer (MPX) which has inputs of +1 or a value taken from BUS 2 in the case of PC or the Microcode Unit in the case of MPC. Each has two outputs: Output1 is connected back to the adder (and is permanently enabled), Output2 is enabled under microcode control.

Whenever PC or MPC is activated by receipt of a microcode packet (it should also have received appropriate data packets), it enables the appropriate inputs to the multiplexer and sends the result of the addition to Output1, and thence back to its own adder input and to Output2 if the corresponding microcode bit is set.

3) *The Buses:* The buses (BUS 1 and BUS 2) have a number of input connections but should of course receive data from only one of them in any one clock period (in P1). Inputs which do not receive data are set to zero. The inputs are internally ORed together (simulating a wired-OR bus) and the result sent to all the outputs.

4) *Data Memory:* The microcode for the Data Memory contains two bits which control its input registers (MAR and MBR), one bit which activates its output register (MBR) and a Read/Write bit. For a read operation (Read/Write = 0), the address received from BUS 2 in P1 of one clock period is copied into MAR in P0 of the next clock period, the memory is read and the result copied into MBR. In P1 the value in MBR is sent to BUS 1. For a write operation (Read/Write = 1), the address sent from BUS 2 is copied into MAR in P0 of next clock period, the data value sent from BUS 1 is copied into MBR and the value is written into the memory.

5) *Registers:* The Registers unit contains 16 general purpose registers, with R0 being permanently set to 0. It receives input values from the ALU and has two outputs connected to BUS 1 and BUS 2. Whenever the Microcode Unit sends a microcode command to the Registers, it appends the appropriate source and destination register numbers extracted from the instruction in IR. In an instruction such as ADD RD RS1 RS2, the value in register RS1 is sent to BUS 1 and that in RS2 to BUS 2.

6) *ALU:* In the Basic version of EMMA, the ALU has two microcode control fields, one to control its inputs (one from each of BUS1 and BUS2) and outputs (one to the Registers and one to BUS2) and one for the function. It executes Add, Subtract, AND, OR, XOR, Shift Left Logical, Shift Right Logical and Shift Right Arithmetic. At the end of each operation it sets the Condition Code bits: CC0 = 0 if the result is 0, CC1 = 1 if the result is negative.

D. *Microcode Format*

The Microcode Unit contains a 128-word microcode memory, with each word having the following format:

Label	Microcode Word	Address

The Label is a string of characters used only for readability purposes. The Microcode Word is represented in hexadecimal format and is structured as shown in Table 2. The bits are clustered into hex characters, two for the Microcode Unit itself, one each for the MPC and PC units, one for the Registers, one for the Memory and two for the ALU.

The (integer) Address field is used for jumps within the microcode memory. The first 32 locations form a jump table indexed by the function number, except for location 0 which contains the (one) microcode word needed to implement the JUMP instruction.

If MPC is loaded with a new address (*i.e.* not MPC+1), there is a 1-clock delay before the new value is returned to the Microcde Unit. In this branch slot, the Microcode unit sets the microcode word to 0x00D0000, thus incrementing MPC automatically in the next clock.

III. ENHANCED EMMA

The Enhanced version of EMMA differs from the Basic version in a number of ways. The ALU is shown in more detail (Figure 4) and contains an additional counter (D) which is required for the implementation of divide. The microcode memory is doubled in size to 256 words and is logically divided into two sections, the Standard Microcode memory (addresses 0-127) and the Alternative Microcode memory (addresses 128-255). The Alternative Microcode allows for the implementation of multi-cycle operations such as multiply and divide. Address 255 is special in that when accessed, it halts the simulation and can thus be used to prevent an attempt to divide by 0, for example.

Unit	Bit	Signal
00	00	Not used
	01	Addr/Lit to BUS1
Mcode	02	Addr/Lit to BUS2
	03	Mcode Addr to MPC
01	04	Select ~CC0
	05	Select CC1
Mcode	06	Execute if True
	07	Execute if False
02	08	Input1 from MPC
	09	Input2 (+ 1)
MPC	10	Input3 from Mcode
	11	O/p To Mcode
03	12	Input1 from PC
	13	Input2 + 1
PC	14	Input3 from BUS2
	15	O/p to I_Memory
04	16	Not used
	17	Write
Regs	18	Source1 to BUS1
	19	Source2 to BUS2
05	20	MAR Input
	21	MBR Input
Data	22	Read/Write
Memory	23	MBR Output
06	24	Input1
	25	Input2
ALU	26	O/p to Regs
	27	O/p BUS2
07	28	Not used
	29	Function
ALU	30	Function
	31	Function

Table 2: Microcode Format

Words in the Alternative Microcode memory have the same basic format as words in the Standard Microcode memory but the Data Memory field in the standard format is replaced by an additional ALU field used to

control the extra facilities needed in the ALU to implement multiply and divide operations (Table 3). In addition, the Address field in the microcode word is sent to the ALU for use as a Literal (immediate) operand.

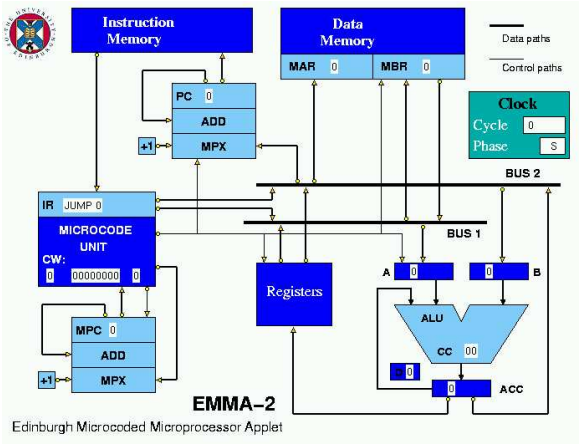

Figure 4: Enhanced EMMA

The Enhanced version of EMMA is upwardly compatible with the Basic version, so microcode written for the Basic version can be used unchanged in the Standard Microcode section of the Enhanced version and will achieve the same effects.

Unit	Bit	Signal
04	16	Dest. to BUS2
	17	Write
Regs	18	Source1 to BUS1
	19	Source2 to BUS2
05	20	Inhibit A to alu
	21	Inhibit B to alu
ALU	22	Enable ACC to alu
	23	Inhibit B to alu
		if A<31> = 0
06	24	Input1
	25	Input2
ALU	26	O/p to Regs
	27	O/p BUS2
07	28	Function
	29	Function
ALU	30	Function
	31	Function

Table 3: Alternative Microcode Format

Other differences when using the Alternative format are that the Destination register in the Registers unit can be routed to BUS 2 and the ALU has extra functions as shown in Table 4. Four of these functions operate directly on the two input registers (A and B) and two manipulate the counter (D) which can be used to implement Division.

NOP
Reverse Subtract
Negate A
Negate B
Shift A $\rightarrow$ by literal & set CC
Shift B $\leftarrow$ by literal
Set D = 17
Decrement D & set CC

Table 4: Extra ALU Functions

The shift functions take as their argument the literal value in the microcode. This argument can be positive (for use in multiplication) or negative (for use in division). When the argument is negative the shift is in the opposite direction to that shown in Table 4. In the case of a left shift on A, the value shifted into the least significant bit is the inverse of CC1, (*i.e.* = 1 if the value in ACC is non-negative). Normally the argument is +1 or -1 but -16 is also required for division.

A. *Muliply and Divide*

1) *Multiply:* The ALU is designed to be able to multiply together the numbers in A and B by repeatedly adding the value in input B to the accumulator (ACC). In each cycle, either the value in B or zero is added to ACC according to the value of the least significant bit of A; A is then shifted right one place and B is shifted left. The operation stops when A becomes zero (whenever A is shifted, the Condition Code bits are set according to the new value in A). This algorithm only works for positive values of A. If A is initially negative, it must first be negated and the final result negated before being returned to the destination register.

2) *Divide:* The ALU is designed to be able to divide 16-bit numbers by first shifting the divisor in input B left 16 places and then repeatedly subtracting it from the value

in ACC, testing for a negative result and shifting B right one place. If the result is negative, B is added back to ACC before repeating the cycle. At the start of the operation, the Quotient, in A, is set to zero (this can be done by loading it from the bus but without sending a value to the bus beforehand). After each subtraction, A is shifted left one place with the value shifted into the least significant position being 1 if the value in ACC is non-negative (as described above). At the end of the operation, A contains the quotient of the result, whilst the remainder is in ACC. However, there is no way in the current version of EMMA to return each value to a different register.

This algorithm only works for 16-bit positive numbers. If either value is negative, it must be negated at the start and the result negated, if appropriate, at the end. The ALU simulation code itself checks its inputs and stops the simulation, with a warning, if a number is out of range. It is also essential not to proceed with a divide if the divisor is zero.

IV. USING THE APPLETS

When downloaded, each applet contains the microcode for the JUMP and LD instructions. The applet automatically executes a JUMP 0 instruction at the start of each simulation. The Data Memory contains values in each of its first 32 locations equal to their address whilst the assembly code contained in the Instruction Memory consists of just two instructions:

```
LD R2 5
STOP
```

Suggested exercises for the students are:

- Using the Basic version, implement the remaining microcode for all single cycle instructions and write an appropriate assembly code program to demonstrate that they work.

- Using the Enhanced version, implement the multiply and divide instructions and demonstrate that they work on suitable test data, including all possible combinations of positive and negative numbers.

- Implement OP1 and OP2 with new functions.

The Basic version requires around 100 lines of microcode in total, including the jump table at the start. The arithmetic instructions (ADD/ADDL, etc) require 3 lines each, BEQZ, BNEG and JREG each require 2, while LD, LDL and LDX require 2, 3 and 5 respectively. The Basic version was used by around 80 students in Autumn 2003. Most of them got most of the instructions working correctly, though some had difficulties, particularly with LDX and STX. Student reaction to using the applet was positive; they felt that it gave them a good insight into how a processor works.

In the Enhanced version, MUL/MULL and DIV/DIVL can be implemented using 18 and 43 microcode instructions respectively. MULL and DIVL use most of the same code as MUL and DIV but each requires different initial microcode instructions to load the operands at the start.

V. CONCLUSION

JavaHASE applets have been successfully used in teaching in a number of courses, either as demonstrations or for practical exercises. Because they are accessible via the WWW, they can be used by students who wish to work off campus or at times outside normal hours. The most recent applet (EMMA) has been used as a replacement for aging laboratory equipment.

VI. Acknowlegements

The development of HASE has been supported by the UK EPSRC through grants GR/J43295 and GR/K19716. Particular thanks are due to Frederic Mallet who created the JavaHASE facility, to the Computer Design class of 2003/4 for their forbearance during development of the Basic model and to Eric McKenzie who taught them.

References

[1] F. Mallet and R.N. Ibbett, "JavaHASE: A Web-based simulation environment", *SCSC'03*, SCS, 2003.

[2] J.L. Hennessy and D.A. Patterson, "Computer Architecture: A Quantitative Approach", Morgan Kaufmann, 1996.

[3] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens et al "The DASH prototype: Implementation and Performance", *Proc ICSA*, 1992, pp 82-103.

[4] G.S. Wolffe, W. Yurcik, H. Osborne and M.A. Holliday, "Teaching Computer Organization/Architecture With Limited Resources Using Simulators", *SIGCSE'02*, 2002,

[5] R.N. Ibbett and F. Mallet, "Computer Architecture Simulation Applets For Use In Teaching", *Proc FIE 2003* IEEE, Nov 2003.

[6] R.M Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", *IBM Journal of Research & Development, Vol 11*, 1967, pp 25-33.