The Functional Simulation of a Simple Microprocessor

Steve Hill

September 13, 1994

Abstract

This paper documents the microprocessor simulator developed to support the teaching digital systems to undergraduate computer scientists. The framework of the simulation is described, and two variant machines, register-based and stack-based, are given. Finally, a more abstract version of the register machine is detailed.

1 Background

This work arose from the need to provide a platform for the simulation of microprocessor architectures suitable for undergraduate students of computer science. However, we believe that our experience shows that the techniques employed could well have a wider application.

Our problem was this: in the second year of our undergraduate programme, two groups of students study a digital systems course. The first group study Computer Systems Engineering which is oriented more towards electronics than is the Computer Science degree. Originally, the course contained a laboratory experiment which involved a fair amount of practical electronics. We decided that iy was an unreasonable requirement that the main stream computer scientists, especially those from largely mathematical or computing backgrounds, should have to perform this experiment. It was proposed, therefore, that these students be offered a software-based project as an alternative.

The first part of this paper describes the main features of the simulation that was developed for this purpose. From the outset we determined that a functional language would be used for the assessment. There were several reasons for this:

- Functional programming is taught in the first year, and this exercise would provide an opportunity for reinforcement. In particular, it would provide an opportunity to show how functional programming could be used in an unfamiliar role.

- The conciseness of the functional descriptions should help students to understand the concepts of machine architecture without needing to worry about the mechanics of the simulation.
- There would not be much time available (approximately one week as it turned out) for the development of the project. A functional language would support rapid and accurate program development.
- The mathematical elegance of functional language would afford opportunities in the future for formal proofs of properties of the machines. We intend to use them as the basis for a compiling techniques course which is due for introduction in 1995.

The simulation was initially written using Gofer [1], and then manually converted into Miranda [5]. Gofer provided a convenient development language since it can be run on a wide range of platforms. Miranda is the (second) programming language that is taught in our first year. The conversion was a simple task requiring less than an hour to complete. We are not aware of any tools that perform the translation in this direction.

We chose to provide simulations for two architectural styles - a register machine and a stack machine. Both machines share a common core which is extended to provide their peculiar instruction sets. The simulations are constructed in three levels.

- The core machine provides the basic architecture described by means of primitive transitions of machine state.
- The micro-code provides a specialisation of the core machine by implementing an instruction set in terms of the basic transitions.
- The assembly language interface is implemented by an assembler and loader which together construct an initial machine state. This is then run until the machine halts.

2 The Core Machine

The core machine provides a characterisation of the machine architecture. It comprises a type of "machine state" along with a set of permitted state transitions. These transitions are the only ones allowed. The style is similar to that adopted by Peyton Jones and Lester [4] for the description of abstract machines for the implementation of functional languages.

Ideally, the type of machine state should be abstract. This would prevent unwanted modification of the core machine. In our implementation, the type of machine state is not actually abstract, but this is for pragmatic pedagogic reasons – there being insufficient time available in the first year to cover abstract data types in Miranda adequately.

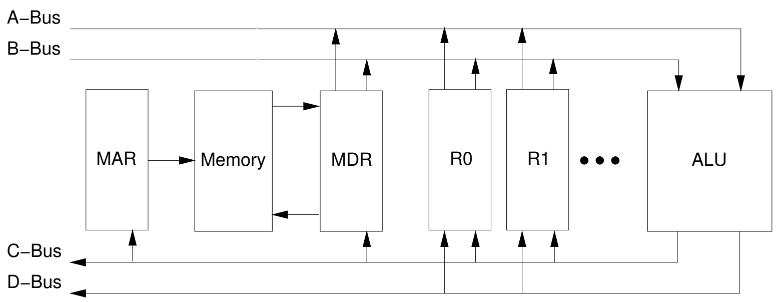

Figure 1: Architecture of the Core Machine

2.1 Components

The machine, depicted in Figure 1 was decomposed into six parts:

- Memory - the memory is modelled as an association list between address and contents.
- Memory Interface - the memory interface comprises two special purpose registers - the memory address register (MAR) and the memory data register (MDR).
- Register File - the registers are modelled as an association list between register number and register contents. The core machine thus makes no commitments as to the number of registers available.
- Buses - the machine has four internal buses or data highways.
- Statistics - the statistics field is used to accumulate measures of the machine's performance.
- Halt Flag - this indicates if the machine has halted.

For the purposes of this simulation, we have chosen to represent machine words and addresses as integers.[1]

```
address == num
word    == num

memory           == assoclist address word
memory_interface == (word, word)
registers        == assoclist num word
buses            == (word, word, word, word)
stats            == (num, num, num, num, num, [char])
```

[1]Miranda makes no type distinction between floating point and integer types - the distinction is maintained at run time

```
machine          == (memory, memory_interface, registers,
                     buses, stats, bool)
```

2.2 Primitive Transitions

The machine is characterised by its transitions. Most transitions involve the movement of data from one part of the machine to another. The primitive transitions represent the lowest level of the simulation. All machine operations must ultimately be composed of these primitives.

```
transition == machine -> machine
```

The A-bus and B-bus are used to communicate argument values to the ALU. Registers in the register file or the MDR (but not the MAR) may be copied onto either the A-bus or the B-bus. In addition to the data copy, the "register to bus" statistic is incremented.

```
regToAbus, regToBbus :: num -> transition

regToAbus n (m, i, r, (a, b, c, d), s, h)

        = (m, i, r, (a1, b, c, d), incRegBus s, h)
          where
          a1 = aLookup n r

mdrToAbus, mdrToBbus :: transition

mdrToAbus (m, (mar, mdr), r, (a, b, c, d), s, h)

        = (m, (mar, mdr), r, (mdr, b, c, d), incRegBus s, h)
```

The C-bus is used to hold the result of an ALU operation. The D-bus carries the condition codes. Data may be copied from the C-bus to the register file, MAR or MDR. The D-bus is more restricted. Data on the D-bus can only be copied into the register file.

```
cbusToReg, dbusToReg :: num -> transition

cbusToReg n (m, i, r, (a, b, c, d), s, h)

        = (m, i, r1, (a, b, c, d), incBusReg s, h)
          where
          r1 = aBind n c r

cbusToMar, cbusToMdr :: transition

cbusToMar (m, (mar, mdr), r, (a, b, c, d), s, h)

        = (m, (c, mdr), r, (a, b, c, d), incBusReg s, h)
```

Access to the memory is via the MAR and MDR. To read memory, the address is placed in the MAR and a memory read cycle executed. The word that is read is placed in the MDR.

```
memRead :: transition

memRead (m, (mar, mdr), r, b, s, h)

        = (m, (mar, mdr1), r, b, incReads s, h)
          where
          mdr1 = aLookup mar m
```

To write data into the memory, the data is placed in the MDR and the destination address placed in the MAR. A memory read cycle is then executed.

```
memWrite :: transition

memWrite (m, (mar, mdr), r, b, s, h)

        = (m1, (mar, mdr), r, b, incWrites s, h)
          where
          m1 = aBind mar mdr m
```

The ALU-cycle transition performs calculations. Data is placed onto the A-bus and possibly the B-bus, then an ALU-cycle executed The result appears on the C-bus. The D-bus holds the condition codes which result from the calculation. The operation of the ALU is specified via an enumerated type, although perhaps a word would be more appropriate.

```
aluOp ::= AluA | AluB | AluIncA | AluDecA | AluNegA | AluAbsA |
          AluAdd | AluSub | AluMul | AluDiv | AluMod

aluCycle :: aluOp

aluCycle op (m, i, r, (a, b, c, d), s, h)

        = (m, i, r, (a, b, c1, d1), incAluCycle s, h)
          where
          c1 = a, if op = AluA
             = b, if op = AluB
             = a+1, if op = AluIncA
             || And other unary operations
             = a+b, if op = AluAdd
             = a-b, if op = AluSub
             || And other binary operations

          d1 = condbit (c1 = 0) aluZero +
               condbit (c1 < 0) aluNeg
```

The calculation of the condition code bits is rather cumbersome. It would be much simpler in a language which provided *words* as a primitive data type.

Such a type could have been defined in Miranda, but at the loss of the set of familiar built-in operators associated with the datatype *num*. Miranda, unlike Gofer and Haskell does not permit the definition of overloaded functions.

The final transition which deals with the simulation proper is halt:

```
halt :: transition

halt (m, i, r, b, s, h)

        = (m, i, r, b, s, True)
```

A set of functions which log messages in the statistics field of the machine state are also provided.

```
printMar, printMdr, printAbus,
printBbus, printCbus, printDbus :: transition

printString :: [char] -> transition
printReg    :: num -> transition
printMem    :: address -> transition
```

2.3 Compound Transitions

The operation of the machine is specified as a sequence of primitive transitions. Two transitions could be combined using functional composition but a variant, which has its arguments reversed, is provided instead. It was thought that the ordering of the arguments for this function would be more intuitive for students unpractised in functional programming.

```
comma :: transition -> transition -> transition

(t1 $comma t2) m = t2 (t1 m)
```

Most machine operations require a sequence of transitions. A list of transitions is performed sequentially by the following function:[2]

```
do :: [transition] -> transition

do []     = id
do (t:ts) = t $comma do ts
```

The *switch* transition is more specialised. It allows a transition to be selected from a table according to the contents of a register. Its role mimics the operation of the mapping PROM in a micro-code engine. This definition would be more concise and readable if Miranda supported the *as* and *don't care* patterns.

[2]A rather more elegant definition in terms of *foldr* could be given, but the simple recursive definition is preferred.

```
switch :: num -> assoclist num transition -> transition

switch reg tab (m, i, r, b, s, h)

        = (aLookup (aLookup reg r) tab) (m, i, r, b, s, h)
```

Similarly, it is often the case that a section of micro-code is parameterised on a register value. The following function allows for this.

```
passReg :: num -> (num -> transition) -> transition

passReg reg tr (m, i, r, b, s, h)

        = tr (aLookup reg r) (m, i, r, b, s, h)
```

2.4 Register Transfer

We are now in a position to be able to define transitions which correspond more closely to the register transfer style. The first allows the contents of one register to be copied to another and might be written as:

$$R_s \rightarrow R_d$$

```
regToReg :: num -> num -> transition

regToReg rs rd

 = do [
        regToAbus rs,
        aluCycle AluA,
        cbusToReg rd
   ]
```

In a similar way transitions for copying data to and from the MAR and to the MAR from the register bank are provided.

```
mdrToReg, regToMdr, regToMar :: num -> transition
```

Finally, some compound transitions for combining registers via the ALU are provided. These might be written in the register transfer style thus:

$$\ominus R_n \rightarrow R_d$$
$$R_n \oplus R_m \rightarrow R_d$$

The second of these transitions is presented:

```
op2 :: num -> aluOp -> num -> num -> transition

op2 rn op rm rd
```

```
 = do [
        regToAbus rn,
        regToBbus rm,
        aluCycle,
        cbusToReg rd
    ]
```

3 A Register Machine

We have implemented two instruction sets for the machine. The first is a register machine loosely based on the Motorola 68000. The second is a paper stack machine which is described in the digital systems lecture course. The register machine is described in detail. The stack machine is rather simpler, so only brief details of its implementation are given.

3.1 Registers

The machine has the following registers.

- *pc* – the program counter
- *ir* – the instruction register
- *tmp1, tmp2* – two temporary registers, not intended for general use
- *sp* – the stack pointer
- *ccr* – the condition code register
- *r0, r1, r2, r3* – four general purpose registers

3.2 Instruction Encoding

The instruction encoding for this machine is very simple. Each instruction is identified by a word. Any arguments are represented by two words following the instruction. The first identifies the addressing mode, and the second the actual argument value *eg.* a register number or address.

```
|| Instruction codes
moveW = 1
addW  = 2

|| Addressing modes
litW  = 1
regW  = 2
```

For example a typical move instruction might be represented by the sequence *moveW, litW, 10, regW, r0* (move literal value 10 into register zero).

3.3 Instructions and Addressing

The basic operation of this machine consists of two operations. The transition *fetch* retrieves a word from the address held in the program counter. This instruction is then *executed* by selecting the appropriate micro-code via a *switch* and invoking it.

```
fetch :: transition

fetch

 = do [
        regToMar pc,
        memRead,
        mdrTo ir,
        op1 pc AluIncA pc
   ]

execute :: transition

execute

 = switch ir [
       (moveW, moveI),
       (addW,  addI),
       || Other instructions
       (haltW, haltI)
   ]
```

Each instruction is now represented as a machine transition. For example:

```
moveI

 = do [
        srcOpTo tmp1,
        dbusToReg ccr,
        destOpFrom tmp1
   ]

addI

 = do [
        srcOpTo tmp1,
        srcOpTo tmp2,
        op2 tmp1 AluAdd tmp2 tmp1,
        destOpFrom tmp1
   ]
```

All these instructions make use of the functions *srcOpTo* and *destOpFrom* which handle addressing modes for the source and destination arguments respectively.

```
srcOpTo :: num -> transition

srcOpTo r

 = do [
        fetch,
        switch ir [
            (litW, do [fetch, regToReg ir r]),
            (absW, do [fetch, regToMar ir, memRead, mdrToReg r]),
            (regW, do [fetch, passReg ir ((flip regToReg) r)]),
            (indW, do [fetch, passReg ir ((flip memToReg1) r)])
        ]
   ]

destOpFrom r

 = do [
        fetch,
        switch ir [
            (litW, halt),
            (absW, do [fetch, regToMem r ir]),
            (regW, do [fetch, passReg ir (regToReg r)]),
            (indW, do [fetch, passReg ir (regToMem r)])
        ]
   ]
```

The description of the machine is now complete. The literal destination mode, although allowed by the instruction set is clearly a nonsense and has been implemented as a *halt* transition.

3.4 Assembler and Loader

The final stage of the simulation was to provide an assembly language, loader and functions to run programs to completion. Using a functional progamming environment was of great benefit. Programs were represented as lists of instructions which were themselves simply elements of an algebraic datatype. There was no need to have a concrete syntax for assembly language programs. Instead the syntax of lists and constructors is used directly.

For simplicity, labels are not implemented. In retrospect this was probably a mistake. Many of the errors that students encountered in their test data were due to incorrect jumps.

```
program == [instruction]

instruction

 ::= Move operand operand |
     Add  operand operand operand |
     || Other instructions
     Halt
```

It is also possible to provide directives or pseudo-ops. In students' version of the simulator a *define constant data* directive was provided, but it was hardly used. The *operand* type describes the set of addressing modes.

```
operand

 ::= Lit word |
     Reg word |
     Abs word |
     Ind word
```

It would have been possible to specialise the operands according to their use. For example, two sorts of operand (one for source operands and another for destinations) could be provided. The possibility of nonsenses such as a literal destination are then excluded.

The task of the assembler is to produce a memory binding. For simplicity it is assumed that programs always start at address zero.

```
assemble :: program -> memory

assemble = assemble1 0 []
```

The main part of the assembler creates a memory binding starting at the specified address from the given program. The memory binding is accumulated in the second argument.

```
assemble1 :: word -> memory -> program -> memory

assemble1 w m []     = m
assemble1 w m (i:is) = assemble1 w1 m1 is
                       where
                       (w1, m1) = assemI w m i
```

Each instruction is converted into its internal representation and placed in memory. The work of assembling an instruction is performed by *assemI*. It assembles the instruction i starting at address w by augmenting the memory bindings in m. It returns the augmented memory binding and the memory location at which subsequent code should be placed.

```
assemI :: word -> memory -> instruction -> (word, memory)

assemI w m (Move src dst)      = assemI2 w m moveW src dst
assemI w m (Add src1 src2 dst) = assemI3 w m addW src1 src2 dst
|| Other instructions
assemI w m Halt                = (w+1, aBind w haltW m)
```

Instructions are assembled according to the number of operands. For example:

```
assemI2 w m instr src dst

        = (w3, m3)
          where
          w1 = w + 1
          m1 = aBind w instr m
          (w2, m2) = assemO w1 m1 src
          (w3, m3) = assemO w2 m2 dst
```

Finally operands are themselves assembled by the function *assemO*.

```
assemO w m (Lit x) = assemO1 w m litW x
assemO w m (Reg x) = assemO1 w m regW x
assemO w m (Abs x) = assemO1 w m absW x
assemO w m (Ind x) = assemO1 w m indW x

assemO1 w m mode val

        = (w2, m2)
          where
          w1 = w + 1
          m1 = aBind w mode m
          w2 = w1 + 1
          m2 = aBind w1 val m1
```

The simulation is completed by the definition of a loader and a function to execute a program to completion. The loader creates a machine in its initial configuration, where the memory is bound to the result of assembling a program.

```
load :: memory -> machine

load mem

        = (mem, (0, 0), aBind pc 0 initial_regs,
            (0, 0, 0, 0), initial_stats, False)
```

The function *run* assembles a program, loads it, and executes it to completion *ie.* until the halt flag becomes true. The result of the run function is a string containing any diagnostic messages generated during the program run, followed by a dump of the machine's final state.

```
run = run' . load . assemble

run' mc

        = d ++ showMachine m2, if h
        = d ++ run' m2       , otherwise
          where
          m1 = execute (fetch mc)
          d = getDiagnostics m1
          m2 = resetDiagnostics m1
```

4 A Stack Machine

The core machine has also been used to implement a stack-based architecture. In this section a brief overview of its unique features is given. This is achieved by reprogramming it with a new set of micro-code. The new machine has only the minimum of internal registers: *pc*, *ir*, *tmp1*, *tmp2*, *sp* and *ccr*.

4.1 Instructions

The stack machine has far more instructions than the register machine, but fewer addressing modes. Most instructions work on data held in the stack. Therefore, the following simple transitions, used in the implementation of many other instructions, are defined.

```
push r

 = do [
        op1 sp AluIncA sp,
        regToMar sp,
        regToMdr r,
        memWrite
   ]

pop r

 = do [
        regToMar sp,
        memRead,
        mdrToReg r,
        op1 sp AluDecA sp
   ]
```

The transitions place the contents of a register onto the stack and pop the top of the stack into a register respectively. The following machine instructions are typical of the bulk of the implementation:

```
addI

 = do [
        pop ir,
        pop tmp1,
        op2 ir AluAdd tmp1 ir,
        dbusToReg ccr,
        push ir
   ]


dupI

 = do [
```

```
        pop ir,
        push ir,
        push ir
   ]
```

There is scope for optimisation of these instructions by careful tracking of values. In many cases it is possible to avoid expensive memory accesses by caching the top elements of the stack. This was set as one of the tasks in the digital systems assignment.

In order to effect addressing modes, there are a number of special instructions which push and pop data to and from the stack. In summary there are:

- *pushLit* – pushes a literal value onto the stack
- *pushAbs*, *popAbs* – push a value held in a specified memory location and pop the top of stack into a specified memory location respectively.
- *pushRel*, *popRel* – push a value held in the stack at a specified offset from *sp*, pop the top of the stack into a location at a specified offset from the *sp* respectively.

This set facilitates the manipulation of constants and local and global variables. There is no provision for indirection, although this would be simple enough to add. An example of these instructions is:

```
popAbsI

 = do [
        fetch,
        pop tmp1,
        regToMar ir,
        regToMdr tmp1,
        memWrite
   ]
```

4.2 Assembler and Loader

The instruction set for the stack machine is much simpler than the register machine, and consequently the assembler is much simpler. The implementation has only to deal with instructions of zero or no operands, and operands when present consist of a single item.

There are only minor differences between the loader for the stack machine and that of the register machine.

5 Refinements

In the simulation described, the level of abstraction (micro-code) was mandated by circumstance. The simulation was specifically designed at a level that coincided with the teaching in the digital systems course. However, if one were to simulate a microprocessor in earnest one would probably wish to start with a high-level description and to refine it (not necessarily in its entirety).

5.1 A Simplified Machine

Initially the internal operations of the device are not a concern. What is of interest is its observable behaviour. The machine state is redefined to reflect this in the following manner:

```
machine == (memory, registers, stats, bool)
```

Notice that the memory interface and the internal buses have been removed. The registers are retained since they are directly observable. A new set of transitions which describe the basic operations of the machine can now be defined. Interfacing to the memory is via the following two transitions:

```
regToMem :: num -> address -> transition
memToReg :: address -> num -> transition
```

Operations to transfer data and manipulate data within the machine are also required. They apply an ALU operation to the contents of two registers and place the result in a destination register. The variant transition *opc* also sets a condition code register.

```
op  :: num -> num -> num -> aluOp -> transition
opc :: num -> num -> num -> num -> aluOp -> transition
```

There is a *halt* transition and a number of transitions responsible for diagnostics as in the micro-code machine.

As with the low-level machine, primitives to sequence transitions, select transitions according to a register value, and to pass register contents to a transition are provided.

```
comma   :: transition -> transition -> transition
do      :: [transition] -> transition
switch  :: num -> assocList num transition -> transition
passReg :: num -> (word -> transition) -> transition
```

The machine is described in a similar fashion to the earlier simulation:

```
fetch :: transition

fetch

 = passReg pc (flip memToReg ir)

execute :: transition

execute
```

```
 = switch ir [
       (moveW, moveI),
       || other instructions
   ]
```

And the individual instructions are similarly simplified, for example:

```
moveI :: transition

moveI

 = do [
        srcOpTo tmp1,
        opc tmp1 tmp1 tmp1 ccr,
        dstOpFrom tmp1
   ]
```

The functions *srcOpTo* and *dstOpFrom* are defined as before.

Work is currently in progress to prove that the two machines are equivalent. A logical framework for reasoning about Miranda programs has been constructed under the Isabelle system [3]. A paper reporting this work is in preparation.

6 Conclusions and Future Work

Experience of marking the assignments based on this simulator would suggest that the students have had little difficulty with it. Few of the questions about the simulator related to the workings of functional languages. Some students have not grasped the difference between the simulation and the device that is being simulated, but this is not a problem confined to a functional implementation. However, we must be somewhat cautious. The group that attempt this assessment are self-selecting. Any student who struggled with functional programming in the first year is unlikely to want to attempt this assessment. Between a half and a third of the CS cohort opted to avoid the simulation exercise (or opted for the interfacing lab) each year.

From the point of view of the implementer, the simulator has been a great success. During the two years of its use, we have identified only a few minor bugs which were fixed in a matter of minutes. One was due to a typographical error and a couple of others were introduced when the simulation was modified to emulate a transputer-style architecture. Performance was not a problem for us since the students' test programs were quite small. On the basis of our experience, we would recommend using functional languages for the rapid and accurate development of software.

Although this work was primarily motivated by the requirements of an undergraduate course, we believe that it is possible to use functional description techniques to design hardware. Design would start with a high-level abstract description of a device, and then proceed via a number of formally verified refinement steps to a concrete description suitable for fabrication, for example [2].

References

[1] Mark P. Jones. *Introduction to Gofer 2.20*, 1991. Available via ftp from *nebula.cs.yale.edu.*

[2] John T. O'Donnell. Generating netlists from executable circuit specifications in a pure functional language. In John Launchbury and Patrick Sansom, editors, *Workshop on Functional Programming.* Springer-Verlag, 1992.

[3] Lawrence C. Paulson. The foundation of a generic theorem prover. Technical Report 130, University of Cambridge, Computer Laboratory, 1988.

[4] Simon L. Peyton Jones. *Implementing Functional Languages.* Prentice-Hall, 1992.

[5] D. A. Turner. An overview of Miranda. *SIGPLAN Notices*, December 1986.