Handy: A Haskell-Based Virtual Machine for the Execution of ARM Assembly Code

Miles McGuire
B.A.(Mod.) Computer Science
Final Year Project April 2014
Supervisor: Dr. David Abrahamson

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
ABSTRACT

We introduce Handy, a Haskell-based virtual machine for execution of ARM assembly code implemented in a clean functional style. Handy implements the ARM4T Instruction Set Architecture and features a design informed by the ARM7TDMI processor.
DECLARATION

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

Dublin, April 2014

______________
Miles McGuire, April 23, 2014
PERMISSION TO LEND

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

_Dublin, April 2014_

Miles McGuire, April 23, 2014
We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [3]

ACKNOWLEDGEMENTS

I’d like to thank David Abrahamson for suggesting this project to me
and for his hard work keeping me working towards the finishing line
despite my occasionally unannounced absences from our meetings.

I should also thank Glenn Strong for his willingness to answer my
questions about the finer points of Haskell and for some excellent
suggestions when I was stuck in the wrong train of thought.

I would not have been able to do complete this project without the
support of my mother, who has always done everything in her power
to help me. Thank you.

Finally, Ana, without whom I might never have returned to study
Computer Science at all, and certainly would not be who I am today.
CONTENTS

i  INTRODUCTION  1

1  INTRODUCTION  2
  1.0.1 Motivation  3
  1.0.2 Literature Review  4

ii  DESIGN AND IMPLEMENTATION  5

2  DESIGN  6
  2.1 Overview  6
  2.2 Instructions  7
  2.3 Central Processing Unit  7
  2.4 Arithmetic and Logic Unit  7
  2.5 Memory  8
  2.6 Instruction Decoder  8

3  IMPLEMENTATION  10
  3.1 The Instruction Data Type  10
    3.1.1 The Argument Type  11
    3.1.2 The Condition Type  12
    3.1.3 The ShiftOp Type  13
    3.1.4 Addressing Mode Types  13
  3.2 Central Processing Unit  14
    3.2.1 Register File  17
    3.2.2 Status Register  18
  3.3 Arithmetic Logic Unit  18
  3.4 Memory  22
  3.5 Instruction Decoder  23
  3.6 Instruction Encoder  25

iii  FURTHER WORK AND CONCLUSION  27

4  FURTHER WORK  28
4.1 Remodel Instructions 28
4.2 Processor Modes 28
4.3 Improved Memory 30
4.4 Interrupts 30
4.5 Thumb Mode 31
4.6 General Purpose I/O 31
4.7 Debugger 32
5 CONCLUSIONS 33

APPENDIX 35
A IN-DEPTH REVIEW OF ARM INSTRUCTION STRUCTURE 36
A.1 Condition Codes 36
A.2 Instruction Families 36
A.3 Opcodes and Data-processing Instructions 36
   A.3.1 Data-processing Instruction Operands 39
A.4 Multiplication Instructions 41
A.5 The S Bit 43
A.6 Load and Store Instructions 43
   A.6.1 Load and Store Addressing Modes 43
A.7 Load and Store Multiple Instructions 45
   A.7.1 Load and Store Multiple Addressing Modes 46
B LIST OF ARM INSTRUCTIONS IMPLEMENTED BY HANDY 47

BIBLIOGRAPHY 49
LIST OF FIGURES

Figure 1  Comparison of ARM instructions and their internal representations  11
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>ARM Condition Codes and their meanings[^4, pp. A3-4]</td>
<td>37</td>
</tr>
<tr>
<td>Table 3</td>
<td>Instruction Families</td>
<td>38</td>
</tr>
<tr>
<td>Table 5</td>
<td>Data-processing Instructions and their Opcodes[^4, pp. A3-7]</td>
<td>40</td>
</tr>
<tr>
<td>Table 7</td>
<td>Meanings of P and W bits in Memory Operations[^4, pp. A5-19]</td>
<td>44</td>
</tr>
<tr>
<td>Table 9</td>
<td>Load and Store Multiple Addressing Modes and the P and U bits[^4, pp. A5-42]</td>
<td>46</td>
</tr>
</tbody>
</table>

[^4]: Refer to the cited source for detailed information.
LISTINGS

Listing 1  The Instruction data type  10
Listing 2  The Argument data type  12
Listing 3  The eval function for the Argument type  12
Listing 4  The Condition data type  12
Listing 5  The ShiftOp data type  13
Listing 6  The AddressingModeMain data type  14
Listing 7  The AddressingModeMulti data type  15
Listing 8  The run function  16
Listing 9  The execute function  16
Listing 10  The execute′ function for LDR  17
Listing 11  The compute function  19
Listing 12  The compute′ helper function for ADD, SUB and AND  19
Listing 13  The computeArith function  20
Listing 14  The computeShiftL function  21
Listing 15  The read and write Memory functions  23
Listing 16  The decodeCondition function  23
Listing 17  The decodeType0 function  24
Listing 18  The decodeDataOp makeInstruction functions  25
Listing 19  The serialiseInstruction function for MUL  26
ACRONYMS

CPU    Central Processing Unit
ALU    Arithmetic/Logic Unit
GADT   Generalised Algebraic Data Type
ISA    Instruction Set Architecture
CPSR   Current Program Status Register
ARM7TDMI  A model of ARM processor
ARM4T   A version of the ARM Instruction Set Architecture

A full list of ARM assembly instructions implemented by Handy and their meanings is included in Appendix B
Part I

INTRODUCTION
INTRODUCTION

This report details the design and implementation of Handy, a Haskell-based simulator of the ARM4T Instruction Set Architecture as seen in the ARM7TDMI microprocessor. Handy implements a great majority of the instructions present in the ARM4T ISA in a clean functional style. Handy consists of a small virtual machine, complete with registers and a true Von Neumann memory architecture, including an instruction encoder and decoder capable of assembling or disassembling ARM machine code.

While first and foremost a simulator that implements the operational semantics of the ARM4T Instruction Set Architecture, Handy also implements several features inspired by the physical ARM7TDMI microprocessor. These are:

- Instruction pipelining into three stages:
  - Fetch, where an instruction is retrieved from memory.
  - Decode, where a previously fetched instruction is prepared for execution.
  - Execute, where an instruction is executed and processor state updated based on its result.

- Stalling of execution, where some instructions require the processor to halt for a number of cycles while they complete.

The inclusion of these features allows for a richer and more accurate simulation, allowing Handy to emulate an ARM7TDMI for simple purposes. Handy permits a programmer to execute simple ARM assembly programs that do not require use of interrupts or physical hardware in a light weight environment, removing the need for a
physical development board or full-featured emulator, both typically costly endeavours both in terms of money and effort.

1.0.1 Motivation

The original motivation for development of Handy is to serve as a platform for use in teaching Compiler Design to students in Trinity College Dublin. Students of Trinity College Dublin study the ARM Instruction Set through use of the ARM7TDMI processor in their first and second years of university. In their third year they undertake a mandatory class in Compiler Design in which they implement a simple programming language on a virtual machine which executes its own bytecode and utilises an assembly language contrived specifically for that purpose. While this solution is adequate, the potential to use the ARM assembly language as a target language is very appealing, as it leverages two years worth of study already undertaken by students.

The main constraint to using the ARM assembly language is a lack of a platform to execute it. While Trinity College has a significant number of ARM7TDMI development boards they are in high demand, and requiring their use by students studying Compiler Design might be considered excessive. ARM provide their own development environment μVision which includes a full emulator, but this is a very substantial program that students must apply for an educational license to download and use. Additionally both these options face the significant problem of being very challenging to integrate seamlessly into a work flow where they are not the primary focus.

The solution, given the unsuitability of the two major options already available, was to forge a new path. By implementing a virtual machine that executes only the ARM assembly language without simulating the full range of physical components as a single purpose binary it could easily be included in a tool chain provided to students.
1.0.2 Literature Review

The use of Haskell to implement an ARM4T simulator is not without precedent. A package in the Haskell “Hackage” repository entitled HARM written for use in a University of Connecticut Computer Science class, “CSE240 Intermediate Computer Systems” in 2001 seeks to accomplish a similar goal. The possible use or extension of this project was investigated but it was found not to be desirable. The style of its implementation is unidiomatic to the point that many of the benefits of using Haskell for this purpose are lost. It was quickly dismissed.

Excluding HARM we were unable to find any other extant projects in this vein, and so turned to implementation from scratch. The primary source of information throughout this endeavour was the ARM Architecture Reference Manual[4]. When this proved too dense or required further explanation, we made reference to ARM Assembly Language: Fundamental and Techniques[2] and ARM System-on-Chip Architecture[1]. Between these three texts more than enough information for a full implementation could be found, and in particular the ARM Architecture Reference Manual[4] will be referenced many times throughout this report.
Part II

DESIGN AND IMPLEMENTATION
DESIGN

2.1 OVERVIEW

The architecture and overall design of Handy is based on the layout of a physical ARM7TDMI processor with separate CPU, ALU, memory, and instruction decoder as well as subcomponents that are necessary to construct each. In this chapter the design of each major component will be discussed in turn. Conceptually the components attempt to maintain strict separation with communication between each being brokered by a CPU. In addition to the components that perform the execution of instructions, perhaps the most important components of all are the instructions themselves. These are modeled very closely on their appearance in ARM assembly programs, primarily because this allows a hypothetical ARM assembly programmer to more easily approach working with Handy. Additionally, this format clearly and succinctly enumerates all necessary fields of an instruction as an abstraction of machine code.

The primary goal throughout the design was a complete and semantically accurate reproduction of the ARM instruction set architecture. This goal evolved over time once implementation started and grew to encompass additional features, taking inspiration from the physical ARM7TDMI processor to add simulation of hardware components such as the instruction pipeline.
2.2 INSTRUCTIONS

Instructions were designed as a generalised algebraic data type that would enclose all possible constructions, using data constructors to represent the instruction name and parameters for each subsequent field. The original intention was that this would yield easily readable instructions even in Handy’s internal representation, with the use of GADTs strictly enforcing types of all parameters.

2.3 CENTRAL PROCESSING UNIT

Early in the design process I decided that the CPU should be the only “stateful” component in the simulator. The entire design process from then on was informed by this decision. This necessitated more careful planning of various components but also made reasoning about the system as a whole considerably simpler. By constraining stateful actions to one small part of the code base, the remainder can be written in simple, explicit Haskell functions that can be tested in isolation.

The result of these steps is a small monadic core that can be easily reasoned about in a formal fashion and implemented or extended very rapidly. Even complex modeling issues such as instruction pipelining and stalling of execution could be easily captured by this representation, and implementations of new instructions introduced to the system concisely.

2.4 ARITHMETIC AND LOGIC UNIT

The major design goal for the ALU was to conform exactly with the operational semantics given as pseudo-code in the ARM Architecture Reference Manual for each instruction. This was accomplished through an iterative design and implementation process — an instruction would be studied in detail in the Reference Manual then
implemented precisely in the ALU. When patterns emerged in the implementation of instructions they would be abstracted into a higher order function and the process would repeat.

The ALU itself is designed in a purely functional style, declaring no data types and having no internal state. Its existence as a “component” of the simulator is entirely notional, as it is designed as a library of functions that implement arithmetic and logic operations when given an instruction rather than being modeled as part of the system.

2.5 MEMORY

Memory in Handy is designed to be as simple as possible, satisfying only the most vital requirements. With that in mind, memory only needs to be a dumb store which maps 32-bit keys to 8-bit values and supports fetching individual bytes or Big Endian four byte words. Implementation details beyond this simple interface are irrelevant for our purposes.

2.6 INSTRUCTION DECODER

The necessity of the Instruction Decoder only became apparent after implementation of other components had commenced. To implement a correct Von Neumann architecture instructions need to be able to be stored in the data memory of the simulator, and thus must be represented in byte form. This necessitates a binary parser to convert the 32 bit instruction words to the Instruction type used by the simulator. Fortunately all necessary information for decoding the binary form of instructions could be found in the ARM Architecture Reference Manual[4], albeit listed in alphabetical order with only the internal structure of an individual instruction’s bit pattern enumerated. Rather than attempt to arrive at a formalisation ahead of time
design and implementation took place in parallel, developing a decoder for limited subsets of the ARM instruction set and confirming correctness or redesigning as necessary.

This naturally converged on an effective and correct decoder that can disassemble machine code representations of all instructions implemented by Handy. Haskell’s Data.Binary package shone as being the perfect tool for implementation, providing access to powerful tools such as parser combinators that allowed components of the decoder to be implemented and tested in isolation before being combined into larger wholes.

Due to this evolutionary development process no in depth design took place for the Instruction Decoder ahead of time. An in-depth review of structure of ARM machine code may be found in Appendix A, which is useful in understanding the issues faced in designing the instruction decoder and summarises all necessary information required for parsing ARM machine code. A more detailed discussion of how the problem was tackled will be found in Chapter 3 - Implementation.
IMPLEMENTATION

3.1 THE INSTRUCTION DATA TYPE

The Instruction data type is integral to the operation of Handy. All components of the system, other than the memory, require touching upon it to a greater or lesser extent. It is represented as a Generalised Algebraic Data Type (GADT), allowing for fine grained control of the types that an instruction can be constructed by. An abbreviated example of the source code listing for the Instruction data type is seen in Listing 1.

Each parameter of the GADT shown in Listing 1 maps directly to a component of an assembly instruction defined in the ARM Architecture Reference Manual. The order that parameters appear in the Instruction data constructors corresponds to the order these fields appear in the actual ARM assembly language, though some fields are optional in the ARM assembly language but all fields are compulsory in this representation. The result of this modeling is such that

Listing 1: The Instruction data type

data Instruction where

ADD  :: Condition -> S -> Destination -> Argument Register
     -> Argument a -> ShiftOp b -> Instruction
MUL  :: Condition -> S -> Destination -> Argument Register
     -> Argument Register -> Instruction
B    :: Condition -> Argument Constant -> Instruction
STM  :: Condition -> AddressingModeMulti -> Argument
     Register -> UpdateReg -> [Register] -> Instruction
Handy’s internal representation of an ARM instruction appears very similar to that of the instruction it represents, as shown in Figure 1.

The implementation of individual components of the Instruction data type will be discussed in subsections. The only exceptions are the “Destination” and “Constant” types, which are type synonyms of the “Register” and “Int32” types respectively and are used only to clarify the language in type signatures throughout the code base.

It is important to understand that the Instruction data type itself has no operations defined upon it and is simply a data model used as a means of driving execution of the other components.

3.1.1 The Argument Type

The Argument type is a wrapper around the “Constant” and “Register” types allowing for interchangeable use of the two in positions which demand it - namely the third operand of a number of arithmetic and logic instructions. It is defined as shown in Listing 2.

“Argument” is also the only type composing the Instruction type with an operation. The eval function takes an argument and a register file and evaluates it in the context of that register file. Its implementation is shown in Listing 3.

1 The register file type is explored in more detail in 3.2.1
3.1 The Instruction Data Type

Listing 2: The Argument data type

```haskell
data Argument a where
    ArgC :: Constant -> Argument Constant
    ArgR :: Register -> Argument Register
```

Listing 3: The eval function for the Argument type

```haskell
eval :: Argument a -> RegisterFile -> Int32
eval (ArgC v) _ = v
eval (ArgR r) rf = rf 'get' r
```

3.1.2 The Condition Type

The Condition type is a nullary data type of eighteen constructors corresponding to the eighteen conditional execution modes of ARM assembly instructions[2]. These constructors serve as symbols which are used to specify which flags to test against when executing an instruction, and have no operations defined directly upon them. A brief example of their implementation is shown in Listing 4.

Listing 4: The Condition data type

```haskell
data Condition = EQ
                | NE
                | CS
                | CC
                | AL
                | NV
```
3.1.3 The ShiftOp Type

The ShiftOp type represents one of the six data processing operations available to certain instructions in the ARM assembly language, for example a logical shift left applied to the third operand of an instruction before the instruction itself is executed.²

The internal representation of these operations is a data type with four unary and two nullary type constructors and can be seen in Listing 5. These data constructors are used as tokens by the ALU (Section 3.3) to decide which operation to apply to the third operand of any instruction in which such “ShiftOp” term is valid as defined by the “Instruction” data type. Such an operation is applied in all cases, resulting in the need for a “NoShift” data constructor which simply selects the identity function and performs no transformation.

3.1.4 Addressing Mode Types

To support the full range of addressing modes used in memory load and store operations Handy utilises two types AddressingModeMain and AddressingModeMulti.

AddressingModeMain consists of four data constructors that implement the nine possible addressing modes for loading or storing

² An example of such can be seen in Figure 1, the “LSL” term in the second example.
a single byte or word value. The definition for AddressingModeMain may be seen in Listing 6.

The UpdateReg and OffsetDir types seen in Listing 6 are binary flags used to specify whether the result of evaluating an instruction in a given addressing mode should update the register holding the base address, and in whether the offset specified by the addressing mode should be added to or subtracted from the base register. Use of these flags allow these four data constructors to represent all nine addressing modes specified by the ISA.

AddressingModeMulti represents the alternative set of addressing modes utilised by the LDM and STM (load/store multiple) instructions. Unlike AddressingModeMain these modes do not require any arguments and are represented by eight nullary data constructors that serve as flags to inform the CPU how the base address register is to be incremented when between stores.

3.2 CENTRAL PROCESSING UNIT

The CPU is implemented in what can be considered a number of parts. First and most significantly is a record type “Machine” which
Listing 7: The AddressingModeMulti data type

```haskell
data Machine = Machine { registers :: RegisterFile
  , memory :: Memory
  , cpsr :: StatusRegister
  , fetchR :: FetchRegister
  , decodeR :: DecodeRegister
  , executeR :: ExecuteRegister
  , stall :: Word8
  , executing :: Bool
}
```

encapsulates all of the stateful components of the CPU: the registers, memory, status register, instruction pipeline and some flags used in the process of execution. The implementation of the Machine type is shown in Listing 7.

Memory shall be discussed in greater detail in a later section while the Register File and Status Registers are discussed in subsections 3.2.1 and 3.2.2 of this section.

The Machine data type represents the state of a machine at one discrete interval of time and is immutable. Clearly this is insufficient to model a running processor, and must only form part of the picture. An additional type “CPU” is defined as the State monad containing the Machine type. This monadic wrapping of Machine allows for stateful computation to be performed within the limits of Haskell’s pure, referentially transparent programming environment and is sufficient to model the running processor.

In addition to these types are a number of functions for manipulating the state of the machine. A driving function run recursively executes the program in memory until a “HALT” pseudo-instruction

---

3 The “HALT” instruction is not in the ARM specification and will be discussed in greater detail later
is encountered. A source code listing for the run function is found in Listing 8.

Listing 8: The run function

```
run :: CPU ()
run = do
  running <- gets executing
  when running $ do
    stalled <- isStalled
    if stalled then
      modify reduceStall
    else do
      modify pipeline
      execute =<< gets executeR
      modify incPC
    run
```

This function drives the pipelining of instructions, stalling of execution, and incrementing of the program counter as well as invoking the execute function which takes an instruction and modifies the processor state with the result of executing it.

Execution of most instructions is handled by the ALU, however the CPU executes branch and memory load and store operations directly as I felt these did not fall under the purview of arithmetic or logic and related instead directly to the machine state. Execution of instructions is driven by a function execute (Listing 9) which wraps a helper function — execute’ — with additional logic to stall execution based on the instruction and trigger a pipeline flush if execution of an instruction modifies the program counter, thereby allowing the semantics of individual instructions enumerated by execute’ to be described in the absence of these concerns.

Listing 9: The execute function

```
execute :: Maybe Instruction -> CPU ()
execute Nothing = return ()
```
execute (Just i) =
  do rf_pre <- gets registers
  execute' i
  modify $ stallMachine i
  rf_post <- gets registers
  when (rf_pre 'Reg.get' Reg.PC /= rf_post 'Reg.get' Reg.PC)
    $ modify flushPipeline

An example of the implementation of the execute' function for the LDR (load machine word) instruction is shown in Listing 10. This example demonstrates that the CPU does utilise the ALU in executing load and store operations (the function computeAddress is defined by the ALU). This example illustrates the relative ease with which instructions are defined, requiring

Listing 10: The execute' function for LDR

execute' :: Instruction -> CPU ()
execute' (LDR cond (ArgR dest) addrm) =
  do machine@(Machine rf mem sr _ _ _ _ _ _) <- get
     when (checkCondition cond sr) $ do
       let (addr,rf') = computeAddress addrm rf sr
       rf'' = Reg.set rf' dest (fromIntegral (mem 'getWord' addr))
       put $ machine { registers = rf'' }

3.2.1 Register File

The Register File is implemented as a record type containing sixteen fields corresponding to the sixteen registers of the ARM7TDMI, R0 through R15. Each of these fields is a single signed 32 bit integer. However, in Haskell record types expose their accessors for getting and setting values as functions. The overhead introduced by using
these functions throughout the simulator necessitated introducing an additional layer of abstraction.

Rather than use the Haskell generated record accessors I introduced additional datatype “Register” with nullary data constructors corresponding to the names of the different registers in the ARM7TDI. These data constructors provide a symbolic representation of the registers. Two functions, get and set allow these symbols to be used on a register file to fetch or update values.

This additional abstraction yields a number of benefits. Since at this level registers are represented as data rather than functions, they can be easily manipulated and used in pattern matching and other control flow constructs in Handy’s source code. Another benefit of this style is to allow for simple aliasing of registers: two names can be made to refer to the same register in the RegisterFile record by mapping them appropriately. This is useful as in ARM assembly language such synonyms appear frequently for example the register R15 serves as the program counter and can be referenced as PC.

### 3.2.2 Status Register

The Status Register or CPSR is implemented as a record type containing four fields representing each of the Carry, Zero, Negative and Overflow flags specified in the ARM Architecture Reference Manual[4, pp. A3-29]. Each of these fields is a boolean value.

### 3.3 Arithmetic Logic Unit

Implementation of the ALU takes the form of a number of pure functions with no declarations of data types. A single function compute serves as an entry point to be invoked by the CPU when executing an instruction. This function takes as its arguments the instruction to be executed and a register file and status register to serve as context
for the execution, and yields as the result of its evaluation a new register file and status register. The source code listing for the compute function may be seen in Listing 11.

Listing 11: The compute function

```
compute :: Instruction
  -> RegisterFile
  -> StatusRegister
  -> (RegisterFile, StatusRegister)
compute i rf sr = (rf',sr'') where (rf',sr') = compute' i rf sr
  srr'' = case getS i of
    S    -> sr'
    NoS  -> sr
```

The main work of the compute function is shifted to a helper, compute’, which utilises pattern matching on instructions to select appropriate behavior. compute’ in turn delegates its work to a higher order function computeArith except in the case of the SMULL and SMLAL instructions⁴ which require unique implementation to accommodate their implementation. Some example listings of the compute’ function can be found in Listing 12, which highlights the brevity with which the semantics of instructions can be defined and the similarity of implementation between various instructions.

Listing 12: The compute’ helper function for ADD, SUB and AND

```
compute' :: Instruction
  -> RegisterFile
  -> StatusRegister
  -> (RegisterFile, StatusRegister)
compute' (ADD cond _ dest src1 src2 shft) rf sr =
  computeArith (+) dest src1 arg2 cond sr rf setSRarith2
```

⁴ SMULL is Signed Multiply Long, a 64-bit version of multiplication. SMLAL is Signed Multiply Long with Accumulate, which takes a third argument and adds the result of multiplication to it
where arg2 = ArgC shiftresult  
(shiftresult, _) = computeShift src2 shft rf sr

compute’ (SUB cond _ dest src1 src2 shft) rf sr =  
computeArith (-) dest src1 arg2 cond sr rf setSRarith3  
where arg2 = ArgC shiftresult  
(shiftresult, _) = computeShift src2 shft rf sr

compute’ (AND cond _ dest src1 src2 shft) rf sr =  
computeArith (.&) dest src1 arg2 cond sr’ rf setSRarith1  
where arg2 = ArgC shiftresult  
(shiftresult, sr’) = computeShift src2 shft rf sr

The source code listing of the computeArith function is seen in Listing 13. It takes as its arguments a binary operation to apply to the two source arguments of the instruction to be executed and the destination register in which to store the result, the condition under which the instruction is to be executed, the status register and register file, and a function with which to update the status register. This additional abstraction of the status register update was found necessary in the course of implementation as not all instructions defined in the ARM4T ISA update all status flags, or can have exceptional effects on certain status flags, to the point of being not so easily generalised. The computeArith function, the functions used to update the status register and the binary operations applied to arguments by the computeArith function are based directly on the pseudocode listings describing the semantics of each instruction found in the ARM Architecture Reference Manual.

Listing 13: The computeArith function

computeArith :: (Int32 -> Int32 -> Int32)  
-> Destination  
-> Argument a  
-> Argument b  
-> Condition
The ALU also computes the result of applying the data processing operations described in Section 3.1.3. This is implemented in similar fashion to the compute function described above, with a function computeShift serving as an entry point and using pattern matching to delegate to the appropriate helper function. The source code listing for the computeShiftL helper used in the calculation of leftward shifts is shown in Listing 14, and is illustrative of the implementation of the various other data processing operations.

Listing 14: The computeShiftL function

```haskell
computeShiftL :: (Num a, Bits a) => a -> Argument b -> RegisterFile -> StatusRegister -> (a, StatusRegister)
computeShiftL val shft rf sr =
  (result, sr')
  where result = val 'shiftL' degree
```
The final responsibilities of the ALU include computing branch and address offsets for the CPU. These are included in the ALU for the sake of consistency of responsibility despite their only being evaluated by functions in the CPU.

### 3.4 MEMORY

As discussed in Section 2.5 the main design requirement for the memory component is not an accurate simulation of the full memory infrastructure of the ARM7TDMI. To satisfy the simulator’s requirements memory must be a data structure that contains bytes and is indexed by 32-bit integers. The original implementation was an array of $2^{32}$ bytes, however this proved not to be viable as allocating such a large block of memory caused a runtime crash when the Haskell runtime’s heap was filled over capacity.

The original array-based implementation was replaced with a library data structure, Data.IntMap, which is implemented internally as a Radix Tree. Data.IntMap utilises Haskell’s lazy evaluation such that only memory locations that contain a value are instantiated on the heap. This implementation does sacrifice the $O(1)$ performance characteristics of an array in favour of $O(\min(n, W))$, where $n$ is the number of elements and $W$ is the width of a machine word — sub-optimal but still sufficiently performant for our purposes considering performance is not a primary concern.

The standard library functions specified by Data.IntMap are wrapped to provide functions for reading and writing 8-, 16-, or 32-bit Big Endian values. The 8-bit version of these is shown in Listing 15. The

```haskell
degree = fromIntegral $ shift 'eval' rf
       sr' | degree == 0 = sr
        | degree <= 32 = sr { carry = testBit (32 - degree) }
        | otherwise = sr { carry = False }
```

```bash
The final responsibilities of the ALU include computing branch and address offsets for the CPU. These are included in the ALU for the sake of consistency of responsibility despite their only being evaluated by functions in the CPU.

### 3.4 MEMORY

As discussed in Section 2.5 the main design requirement for the memory component is not an accurate simulation of the full memory infrastructure of the ARM7TDMI. To satisfy the simulator’s requirements memory must be a data structure that contains bytes and is indexed by 32-bit integers. The original implementation was an array of $2^{32}$ bytes, however this proved not to be viable as allocating such a large block of memory caused a runtime crash when the Haskell runtime’s heap was filled over capacity.

The original array-based implementation was replaced with a library data structure, Data.IntMap, which is implemented internally as a Radix Tree. Data.IntMap utilises Haskell’s lazy evaluation such that only memory locations that contain a value are instantiated on the heap. This implementation does sacrifice the $O(1)$ performance characteristics of an array in favour of $O(\min(n, W))$, where $n$ is the number of elements and $W$ is the width of a machine word — sub-optimal but still sufficiently performant for our purposes considering performance is not a primary concern.

The standard library functions specified by Data.IntMap are wrapped to provide functions for reading and writing 8-, 16-, or 32-bit Big Endian values. The 8-bit version of these is shown in Listing 15. The

```haskell
degree = fromIntegral $ shift 'eval' rf
       sr' | degree == 0 = sr
        | degree <= 32 = sr { carry = testBit (32 - degree) }
        | otherwise = sr { carry = False }
```
16-bit functions are implemented in terms of the 8-bit functions, and the 32-bit functions in terms of the 16-bit functions.

Listing 15: The read and write Memory functions

```haskell
getByte :: Memory -> Word32 -> Word8
getByte mem a = M.findWithDefault 0 (fromIntegral a) mem

writeByte :: Memory -> Word32 -> Word8 -> Memory
writeByte mem a v = M.insert (fromIntegral a) v mem
```

3.5 INSTRUCTION DECODER

The Instruction Decoder is a substantial component of the simulator. It is in effect a simple disassembler for ARM machine code built to decode the bit patterns specified in the ARM Architecture Reference Manual[4, ch. A3]. The decoder is implemented using the Get monad from Haskell’s Data.Binary package. Use of the Get monad was not explicitly required for implementation of the decoder but yielded compelling benefits for speed and correctness of development by providing access to Haskell’s “Alternative” type class for use in parser combinators.

Instruction words are passed to the decoder as a ByteString, the format utilised by Data.Binary’s Get monad. Parsing of binary representations of ARM instructions takes place in stages. As all ARM instructions contain a condition field as described in Section A.1 this field is extracted immediately. The Condition data type described in Section 3.1.2 is structured as an enumeration of the same structure as the condition codes detailed in Section A.1 and derives the Enum type class. This allows a direct transformation of the uppermost four bits to the Condition data type, as shown in Listing 16.

Listing 16: The decodeCondition function
decodeCondition :: G.Get Condition
decodeCondition = do byte <- G.getWord8
                      let cond = toEnum $ fromIntegral $ byte 'shiftR' 4
                      return cond

The only other field contained in all instruction words is the “instruction family” sequence in bits 25 through 27 as described in Section A.2. This field is extracted and used to select the parser to be used in parsing the remaining bits. An example of one of these further parsing functions appears in Listing 17, showing the parser for instructions in the “000” family. Listing 17 highlights the use of Haskell’s Control.Applicative.Alternative operator <|> in implementing the Instruction Decoder. In this example, the decoder first attempts to parse using the decodeDataOp\(^5\) function. If this function fails to parse an instruction from the input its effects are discarded and the decodeMultiply instruction is applied instead. If at any point a function successfully parses an instruction from the input data that instruction is returned and no further parsing occurs.

Listing 17: The decodeType0 function

```
decodeType0 :: Condition -> G.Get Instruction
decodeType0 cond = decodeDataOp cond decodeDataShiftImm
                     <|> decodeMultiply cond
                     <|> decodeMisc cond
                     <|> decodeDataOp cond decodeDataShiftReg
                     <|> junk
```

Each of the parsing functions applied at this level follows a similar pattern. The separate fields of the instruction are extracted from the correct positions in the input stream — in the case of a DataOp this means extracting the destination register and source registers, the “S”

---

\(^5\) A DataOp is any arithmetic or logic instruction other than multiplication. This distinction and the details of it are discussed in depth in Appendix A
bit\textsuperscript{6}, and the opcode. These fields can then be combined to form the instruction, as shown in Listing 18. While these examples apply only to one type of instruction (albeit the most numerous type), the process is essentially the same for all other types.

Listing 18: The decodeDataOp makeInstruction functions

\begin{verbatim}
decodeDataOp :: Condition
    -> G.Get (Argument a, ShiftOp b)
    -> G.Get Instruction

decodeDataOp cond parser =
do (src2, shft) <- G.lookAhead parser
    s <- G.lookAhead decodeS
    opcode <- G.lookAhead getOpcode
    src1 <- G.lookAhead decodeRegisterSrc
    dest <- G.lookAhead decodeRegisterDest
    makeInstruction opcode cond s dest src1 src2 shft

makeInstruction opcode cond s dest src1 src2 shft =
case opcode of
    0 -> return $ AND cond s dest src1 src2 shft
    1 -> return $ EOR cond s dest src1 src2 shft
    2 -> return $ SUB cond s dest src1 src2 shft
    ...    
    14 -> return $ BIC cond s dest src1 src2 shft
    15 -> return $ MVN cond s dest src2 shft
    _ -> empty
\end{verbatim}

3.6 INSTRUCTION ENCODER

The final piece is a rudimentary assembler that converts Handy’s Instruction data type to binary form. This allows programs to be executed to be written in Haskell notation rather than passed as binary

\footnote{See Section A.5}
to the running simulator, as well as rapid testing of the validity of Instruction Decoder outputs. This component is not part of the original design, and was instead written as a tool for development and is included in the source for completeness’ sake.

The encoder provides a suite of serialising functions for converting instruction fields to 32 bit words using Haskell’s Data.Bits (a package containing bitwise operations) to assemble them manually. This is simple to conceptualise and implement using the ARM Architecture Reference Manual[4] as a blueprint, though leads to somewhat verbose function definitions. Steps were taken to minimise this as much as possible but as this component is treated as being outside of the true scope of the project it saw only enough effort as was required to make it viable and useful for development purposes.

Appendix A goes into great detail as regards the binary structure of ARM instructions, and the Instruction Encoder implements these rules dogmatically. Listing 19 shows an illustrative example of how these functions are implemented. The .|. operator seen in Listing 19 represents bitwise OR.

Listing 19: The `serialiseInstruction` function for MUL

```haskell
serialiseInstruction (MUL cond s dest (ArgR src1) (ArgR src2))
    = serialiseCondition cond
    .|. serialiseS s
    .|. serialiseReg 16 dest
    .|. serialiseReg 0 src1
    .|. serialiseReg 8 src2
    .|. bit 4
    .|. bit 7
```
Part III

FURTHER WORK AND CONCLUSION
FURTHER WORK

As we have seen, a significant amount of work has already been done to make Handy a viable simulator for use in a classroom context. There do however remain a number of areas in which Handy could be improved.

4.1 REMODEL INSTRUCTIONS

The Instruction data type seemed outwardly simple to model, requiring almost no transformation whatsoever from the implementation described in the ARM Architecture Reference Manual. Unfortunately this lead to over eager implementation before certain subtleties came to light. The design currently implemented as described in Chapter 3 is serviceable but sub-optimal. A more rigorous solution would move away from GADTs as a basis for the design and use Haskell’s type classes more extensively instead. This would yield a more easily testable code base and simplify implementation throughout, possibly eliding the need for an “Argument” type entirely in favour of an Argument type class.

4.2 PROCESSOR MODES

All programs running in Handy run in effectively System mode, and the processor cannot be shifted to another mode. While it would be a somewhat substantial undertaking it would be very desirable to extend the simulator to accommodate switching between the various ARM architecture defined processor modes. With this in place, sys-
tem functions such as instruction set extension and interrupts could be realised correctly. This omission is the single biggest deviance from the ARM4T Instruction Set Architecture in Handy. To address this the Register, CPU and Status Register components would need to be expanded as ARM processors provide separate register banks and status registers for each mode of execution. This is further complicated by some registers being shared between register banks.

One possible solution to this problem would be to expand the RegisterFile type to contain enough fields for every register in every mode, and reimplement the get and set functions to also take the current processor mode as a parameter in addition to the register symbol currently provided. This would allow the get and set functions to select the appropriate register based on the processor mode whilst having a relatively small footprint in terms of code needing to be rewritten.

The StatusRegister type would need to be expanded to include the processor mode. Certain modes have an entirely separate status register but restore the old status register when returning to the previous mode. This feature would also need to be accounted for, as this is used to implement exception handlers and similar. A possible solution is to expand the definition of the StatusRegister type to encompass the status registers of all possible modes and provide functions that retrieve information from the appropriate fields based on the mode in the same fashion as suggested for the register file above.

With these changes in place Handy’s simulation of the ARM4T architecture would be complete. Processor modes alone would be of little practical application but are a necessity if other extensions were to be made, for example interrupts would require both IRQ (interrupt) and FIQ (fast interrupt) modes to be implemented correctly.
4.3 Improved Memory

Currently the memory system used by Handy is incredibly simple. With some additional work this could be greatly improved. Rather than providing a map directly from 32-bit addresses to 8-bit values the memory implementation could be modified to utilise a page based system. The upper 20 bits of an address could be used as an address to select a 4096 byte array from a map of memory pages. This array-based page would offer \( O(1) \) performance for reads and writes. This could be further expanded on using Haskell’s Data.LRU to implement a cache or caches in front of memory to provide a simulation of a more complete memory hierarchy with better performance characteristics. The current model also provides no concept of write protection for memory regions, with all locations being writable. This could be accommodated in a revised system, allowing for Read Only pages and privileged vs unprivileged access to memory regions.

In addition to improved performance, this would allow Handy to model much more complex systems as well as potentially serving as a tool for teaching students about caching and the memory hierarchy, if sufficient instrumentation could be provided to illustrate activity.

4.4 Interrupts

Handy has no concept of interrupts whatsoever. This is not a great failing for most classroom purposes and a great deal can be done with Handy in its current state, but for a full-featured simulation it is an important oversight. The system would need rigorous design before any implementation could take place — incorrect implementation would be worse than no implementation as any knowledge the user has of a real ARM processor would now misinform their decisions with Handy — but inclusion of interrupts would be a very significant step towards full hardware simulation and would allow
Handy to simulate very complex programs, for example pre-emptive multitasking. We have no recommendations at this time as to how a fully implemented Interrupts system for Handy might look. Presumably it would require multi-threading to handle the asynchronous nature of interrupts, though at this time that is purely conjecture.

4.5 THUMB MODE

Thumb Mode is a special processor mode provided by some ARM processors\(^1\) which utilises a re-encoded subset of the ARM instruction set in 16-bits. It is perfectly reasonable never to implement Thumb mode in Handy, but it would be required if one had designs to some day make Handy a full fledged ARM7TDMI emulator. Thumb mode does not in fact alter the underlying programmers’ model of the ARM architecture\(^4\), pp. A6-2] so implementation should not be particularly arduous, the most complicated component being an implementation of a Thumb instruction decoder which could be based on the present 32-bit instruction decoder.

4.6 GENERAL PURPOSE I/O

Another “nice to have” feature that could potentially be implemented is simulation of physical outputs such as GPIO pins either using operating system signals or an in-Haskell implementation. This seems outwardly to be a relatively simple task to implement but was entirely out of scope for the purposes of this project. With such an addition Handy could be used to simulate an even greater spectrum of ARM programs.

\(^1\) the “T” in ARM7TDMI denotes support of Thumb Mode
A potentially very useful feature would be a full debugging system either within or around Handy that allowed for pausing of execution, introspection of registers and memory in a running simulation, breakpoints and so on. This would greatly improve the quality of life for students and other programmers using Handy as a platform for testing or developing programs. Unfortunately it is potentially a very significant undertaking, requiring instrumentation or modification of practically every component of the system.
I am very satisfied with how Handy has taken shape over the course of recent months. Handy met and exceeded the original specification of a minimum viable simulator, and has been a joy to work on. Handy implements the vast majority of the ARM4T instruction set in a semantically correct fashion, and generally exceeded even my own expectations. The potential for Handy to continue growing into an even more capable tool as detailed in Chapter 4 is an exciting prospect, and one which I have a mind to pursue in my own time in the future.

The majority of work in proving the correctness of implementation in Handy was done manually, using the ARM Architecture Reference Manual’s pseudo-code operational semantics for each instruction as a guide. I have, however, had the opportunity to compare Handy to a physical ARM7TDMI in Trinity College Dublin in recent weeks. The results of this test were outstandingly positive, producing identical results on all test programs.

Even so, Handy is not perfect. As mentioned in Chapter 4 the modeling of the Instruction data type in particular is sub-optimal but I did not feel confident refactoring such an integral part of the system by the time these problems came to light. A better implementation using the knowledge gained from this first attempt could potentially mean significant improvements in the readability and maintainability of the code base, and allow for the introduction of automatic testing using QuickCheck to provide an additional level of correctness guarantees going forwards.

The most challenging part of developing a system like Handy is complete — reaching the first useful iteration. Whether it proves to
be sufficient in its current state or sees further development in coming years remains to be seen.
Part IV

APPENDIX
A.1 CONDITION CODES

Bits 28 through 31 of any ARM instruction word denote the condition code for the instruction. The “NV” condition code, which is architecturally undefined on the ARM4T architecture, is co-opted to serve as the “HALT” pseudo-instruction used to tell Handy to end simulation. This is required as no equivalent instruction exists in the actual processor. Table 1 lists all condition bit fields and the condition they refer to.

A.2 INSTRUCTION FAMILIES

Bits 25 through 27 of the instruction word divide the remaining possible values into one of eight families. This is not explicitly stated in any literature read in the course of implementing this simulator but was instead observed from Figure A3-1 of the ARM Architecture Reference Manual[4, pp. A3-2]. This information simplified implementation of the parser, especially so since two of the eight families contained no instructions required by the simulator1.

A.3 OP CODES AND DATA-PROCESSING INSTRUCTIONS

For instructions other than memory loads and stores and multiplications, bits 21 through 24 hold the opcode. These instructions are

---

1 Coprocessor manipulating instructions and software interrupts
### Table 1: ARM Condition Codes and their meanings [4, pp. A3-4]

<table>
<thead>
<tr>
<th>Bits 31:28</th>
<th>Name</th>
<th>Meaning</th>
<th>Flag States</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td>Equal</td>
<td>Z set</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>Not Equal</td>
<td>Z clear</td>
</tr>
<tr>
<td>0010</td>
<td>CS/HS</td>
<td>Carry set/unsigned higher or same</td>
<td>C set</td>
</tr>
<tr>
<td>0011</td>
<td>CC/LO</td>
<td>Carry clear/unsigned lower</td>
<td>C clear</td>
</tr>
<tr>
<td>0100</td>
<td>MI</td>
<td>Minus/negative</td>
<td>N set</td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
<td>Plus/positive or zero</td>
<td>N clear</td>
</tr>
<tr>
<td>0110</td>
<td>VS</td>
<td>Overflow</td>
<td>V set</td>
</tr>
<tr>
<td>0111</td>
<td>VC</td>
<td>No overflow</td>
<td>V clear</td>
</tr>
<tr>
<td>1000</td>
<td>HI</td>
<td>Unsigned higher</td>
<td>C set and Z clear</td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
<td>Unsigned lower or same</td>
<td>C clear or Z set</td>
</tr>
<tr>
<td>1010</td>
<td>GE</td>
<td>Signed greater than or equal</td>
<td>N equal to V</td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
<td>Signed less than</td>
<td>N not equal to V</td>
</tr>
<tr>
<td>1100</td>
<td>GT</td>
<td>Signed greater than</td>
<td>Z clear and N equal to V</td>
</tr>
<tr>
<td>1101</td>
<td>LE</td>
<td>Signed less than or equal</td>
<td>Z set or N not equal to V</td>
</tr>
<tr>
<td>1110</td>
<td>AL</td>
<td>Always</td>
<td>True</td>
</tr>
<tr>
<td>1111</td>
<td>X</td>
<td>Undefined for ARM4T architecture</td>
<td>X</td>
</tr>
</tbody>
</table>
### A.3 Opcodes and Data-Processing Instructions

#### Table 3: Instruction Families

<table>
<thead>
<tr>
<th>Bits 27:25</th>
<th>Family</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Data processing immediate shift</td>
</tr>
<tr>
<td></td>
<td>Data processing register shift</td>
</tr>
<tr>
<td></td>
<td>Miscellaneous Instruction</td>
</tr>
<tr>
<td></td>
<td>Multiplies</td>
</tr>
<tr>
<td></td>
<td>Extra Loads/Stores (not implemented)</td>
</tr>
<tr>
<td>001</td>
<td>Data processing immediate value</td>
</tr>
<tr>
<td></td>
<td>Move immediate to status register (not implemented)</td>
</tr>
<tr>
<td></td>
<td>Undefined instruction</td>
</tr>
<tr>
<td>010</td>
<td>Load/store immediate offset</td>
</tr>
<tr>
<td>011</td>
<td>Load/Store register offset</td>
</tr>
<tr>
<td></td>
<td>Media instruction (not implemented)</td>
</tr>
<tr>
<td></td>
<td>Architecturally undefined</td>
</tr>
<tr>
<td>100</td>
<td>Load/Store Multiple</td>
</tr>
<tr>
<td>101</td>
<td>Branch / Branch and Link</td>
</tr>
<tr>
<td>110</td>
<td>Coprocessor Load/Store (not implemented)</td>
</tr>
<tr>
<td></td>
<td>Double Register Transfers (not implemented)</td>
</tr>
<tr>
<td>111</td>
<td>Coprocessor Data Processing (not implemented)</td>
</tr>
<tr>
<td></td>
<td>Coprocessor Register Transfer (not implemented)</td>
</tr>
<tr>
<td></td>
<td>Software Interrupt (not implemented)</td>
</tr>
</tbody>
</table>
split across two of the “families” described in Section A.2 depending on whether the third operand of the instruction is an immediate or register value. These opcodes represent what the ARM Architecture Reference Manual describes as “Data-processing instructions”[4, pp. A3-7], and all of these sixteen data processing instructions share similar structures in their binary representations. Multiplications and memory operations are exceptional and have neither opcodes in this sense nor similar structures to data-processing instructions. The sixteen data-processing instructions and their opcodes are shown in Table 5.

A.3.1 Data-processing Instruction Operands

For data processing instructions the destination and first source value must be registers. Bits 12 through 15 denote the destination register while bits 16 through 19 represent the source register. The second source operand is a “data-processing operand” as defined in Section A5.1 of the ARM Architecture Reference Manual[4, pp. A5-2], which can be any of an immediate value, a register, or a register with some logical or arithmetic shift or rotation applied to it.

If the instruction family bits of the instruction word are “000” then the least significant twelve bits of the instruction word are interpreted as a register value with some operation (possibly the identity operation) applied to it. In this case, bits 0 through 3 specify the register number. Bits 5 and 6 specify the manner of transformation to apply. Bit 4 specifies whether the register is to be shifted by an immediate value or by a value taken from a register. If bit 4 is clear bits 7 through 11 represent a 5 bit immediate value by which to shift the value in the source register. If bit 4 is set bit 7 must be clear and bits 8 through 11 represent the register from which to take the value by which to shift.

2 00 = shift left, 01 = logical shift right, 10 = arithmetic shift right, 11 = rotate right
### Table 5: Data-processing Instructions and their Opcodes[4, pp. A3-7]

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND - Logical AND</td>
</tr>
<tr>
<td>0001</td>
<td>EOR - Logical Exclusive OR</td>
</tr>
<tr>
<td>0010</td>
<td>SUB - Subtract</td>
</tr>
<tr>
<td>0011</td>
<td>RSB - Reverse Subtract</td>
</tr>
<tr>
<td>0100</td>
<td>ADD - Add</td>
</tr>
<tr>
<td>0101</td>
<td>ADC - Add with Carry</td>
</tr>
<tr>
<td>0110</td>
<td>SBC - Subtract with Carry</td>
</tr>
<tr>
<td>0111</td>
<td>RSC - Reverse Subtract with Carry</td>
</tr>
<tr>
<td>1000</td>
<td>TST - Test</td>
</tr>
<tr>
<td>1001</td>
<td>TEQ - Test Equivalence</td>
</tr>
<tr>
<td>1010</td>
<td>CMP - Compare</td>
</tr>
<tr>
<td>1011</td>
<td>CMN - Compare Negated</td>
</tr>
<tr>
<td>1100</td>
<td>ORR - Logical (inclusive) OR</td>
</tr>
<tr>
<td>1101</td>
<td>MOV - Move</td>
</tr>
<tr>
<td>1110</td>
<td>BIC - Bit Clear</td>
</tr>
<tr>
<td>1111</td>
<td>MVN - Move Not</td>
</tr>
</tbody>
</table>
the value in the source register. There are two special cases of the above:

1. A register with no operation applied to it is represented as a shift left by an immediate value of 0.

2. A rotate right by an immediate value of zero is a “rotate right with extend” operation, which shifts the instruction operand in the specified register by one position to the right and inserts a 1 at the vacated most significant bit if the carry flag was set and a 0 otherwise. The carry out from the shifter is the least significant bit of the original value, though this does not necessarily replace the carry flag in the CPU’s status register.

Conversely, if the instruction family bits of the instruction word are “001” then the least significant twelve bits of the instruction word are interpreted as an immediate value, with bits 0 through 7 being an eight bit constant and bits 8 through 11 a four bit rotation. The actual value of the immediate is calculated by left shifting the four bit rotation value by one place and then right-rotating the eight bit constant by that many positions. This allows the ARM instruction set to represent immediate values much larger than 12 bits would allow, but for constants larger than 255 only some values are valid and representable with this encoding.

A.4 multiplication instructions

There are six possible multiplication instructions in the ARM4T instruction set. They are respectively:

1. MUL - Multiplies the value of two registers together, truncates the result to 32 bits, and stores the result in a third register. [4, Section A3.5.1, pp. A3-10]
2. MLA - Multiplies the value of two registers together, adds the value of a third register, truncates the result to 32 bits, and stores the result in a fourth register.[4, Section A3.5.1, pp. A3-10]

3. SMULL and UMULL - Multiply the values of two registers together and store the 64-bit result in two parts in a third and a fourth register. SMULL treats operands as signed and UMULL treats operands as unsigned.[4, Section A3.5.2, pp. A3-10]

4. SMLAL and UMLAL - Multiply the values of two registers together and add the resultant 64-bit value to a 64-bit value taken from a third and a fourth register, then store the final value back in the third and fourth registers in two parts.[4, Section A3.5.2, pp. A3-10]

Multipies are part of instruction family “000” and are distinguished from data-processing operations by having both bits 4 and 7 set and bits 5 and 6 are clear. Bit 23 specifies whether the instruction produces a 32- or 64-bit result. If bit 23 is set the instruction is a “long” multiplication and bit 22 specifies whether it is the signed or unsigned variant. Bit 22 must be clear if bit 23 is clear. Bit 21 specifies whether the instruction is an accumulate variant.

Unlike data-processing instructions, multiplication instructions can only have register operands and cannot have immediate values specified. Bits 0 through 3 represent the left source operand and bits 8 through 11 represent the right source operand. For MUL, bits 12 through 15 are specified as “Should Be Zero” though other values do not cause an error. For MLA, bits 12 through 15 specify the source of the accumulator value that should be added to the result of the multiplication before storing in the destination register. For all 64-bit variants bits 12 through 15 specify the destination for the least significant 32 bits of the result. Bits 16-19 specify the destination register for MUL and MLA or the destination for the most significant 32 bits of the result otherwise.
A.5 THE S BIT

Data-processing and multiplication instructions can optionally\(^3\) update the status flags of the CPU. Whether this occurs is decided by bit 20 of the instruction word, referred to as the “S bit”[4, pp. A3-7], and the method by which an instruction updates the status flags with its result varies by instruction.

A.6 LOAD AND STORE INSTRUCTIONS

Load and Store instructions have yet another distinct structure, and are divided across two instruction families depending on whether the addressing mode uses a register or immediate value as an offset. Bit 20 specifies whether the instruction is a load (bit 20 is set) or a store (bit 20 is clear) instruction. Bit 22 specifies whether the instruction loads a byte (bit 22 is set) or a word (bit 22 is clear) value. Bits 21, 23, 24, and 25 specify the addressing mode and bits 0 through 11 are addressing mode specific. Bits 16 through 19 specify the register containing the base address and bits 12 through 15 specify the register whose contents are either to be stored or replaced by the loaded value.

A.6.1 Load and Store Addressing Modes

Broadly, there are three[4, pp. A5-18] types of addressing mode for Load and Store instructions:

1. Immediate offset/index

2. Register offset/index\(^4\)

\(^3\) Or necessarily, in the case of CMP, CMN, TST and TEQ instructions, though this is also accomplished using the S bit where its being unset is an undefined instruction[4, pp. A3-7]

\(^4\) Though the Register offset/index class can be considered a special case of Scaled register offset/index the ARM Architecture Reference manual treats them separately
Table 7: Meanings of P and W bits in Memory Operations[4, pp. A5-19]

<table>
<thead>
<tr>
<th>P</th>
<th>W</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Post-indexed addressing: Value taken from base address register and used as address, then the offset is applied and the result written back to the base address register.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Unprivileged operation: The memory operation is performed in User mode. As this simulator does not implement modes other than Supervisor mode, this is undefined for our purposes.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Offset addressing: Offset applied to value in base address register to create the address, but the modified value is discarded after the operation.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Pre-indexed addressing: Offset applied to the value taken from the base address register and used as the address for the operation. After the operation the base address register is updated with the result of applying the offset.</td>
</tr>
</tbody>
</table>

3. Scaled register offset/index

Bit 23 is referred to by the ARM Architecture Reference Manual as the “U” bit, and specifies whether the offset described by bits 0 through 11 is to be added or subtracted from the base address. If the U bit is set, the offset is added. If the U bit is clear, the offset is subtracted.

Bits 21 and 24 are referred to as the “W” and “P” bits respectively, and their meanings dependent on each other and are shown in Table 7.
Whether an addressing mode uses an immediate or register offset is decided by bit 25\(^5\), with bit 25 being set indicating a register indexed addressing mode.

The offset value is represented by bits 0 through 11. In immediate addressing mode bits 0 through 11 are treated as a 12-bit constant and directly. In register and scaled register addressing modes bits 0 through 3 specify the register containing the offset and bit 4 must be zero. Bits 5 and 6 represent the scaling type, using the same scheme and semantics as described in Section A.3.1. Bits 7 through 11 specify the amount by which the offset is to be shifted.

### A.7 Load and Store Multiple Instructions

Distinct from ordinary load and store instructions are what the ARM Architecture Reference Manual refers to as “Load and Store Multiple” instructions[4, pp. A3-26]. These instructions reside in an entirely separate instruction family and have a unique bit pattern. Load and Store Multiple instructions are composed of five distinct parts:

1. A register list represented as a bit array in bits 0 through 15, with bit 0 representing \(R_0\) and bit 15 representing \(R_{15}\) and set bits denoting membership in the set. The registers listed are those to be populated with loaded values or to have their values stored in memory.

2. The \(P\), \(U\) and \(W\) bits (bits 24, 23 and 20 respectively). The \(P\) and \(U\) bits specify the addressing mode for the instruction and the \(W\) bit specifies whether the base address register is to be updated by execution of the instruction.

---

\(^5\) Which is one of the bits I describe as deciding the “instruction family”, ergo immediate indexed loads and stores reside in a separate instruction family to register indexed loads and stores
A.7 load and store multiple instructions

Table 9: Load and Store Multiple Addressing Modes and the P and U bits[4, pp. A5-42]

<table>
<thead>
<tr>
<th>P</th>
<th>U</th>
<th>Name and Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>DA/FA (Decrement After / Full Ascending) - Base address register decremented by 4 after each register</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>IA/FD (Increment After / Full Descending) - Base address register incremented by 4 after each register</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>DB/EA (Decrement Before / Empty Ascending) - Base address register decremented by 4 before each register</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>IB/ED (Increment Before / Empty Descending) - Base address register incremented by 4 before each register</td>
</tr>
</tbody>
</table>

3. The S bit (bit 22) the purpose of which pertains to privileged or unprivileged execution modes and therefore falls outside of the bounds of the simulator, meaning it is ignored.

4. The L bit (bit 20) which distinguishes between a Load and a Store operation (set and unset respectively).

5. The base address register specified by bits 16 through 19.

A.7.1 Load and Store Multiple Addressing Modes

Load and Store Multiple instructions must be in one of four addressing modes, which describe the manner in which the base address register is modified when storing the registers specified by the register list. Each addressing mode is known by two names, with one name being most useful when describing block data transfer operations and another name for the same mode in terms of stack operations. These names and how they relate to the P and U bits appear in Table 9.
# List of ARM Instructions Implemented by Handy

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Addition</td>
</tr>
<tr>
<td>ADC</td>
<td>Addition with carry</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtraction</td>
</tr>
<tr>
<td>SBC</td>
<td>Subtraction with carry</td>
</tr>
<tr>
<td>RSB</td>
<td>Reverse subtraction</td>
</tr>
<tr>
<td>RSC</td>
<td>Reverse subtraction with carry</td>
</tr>
<tr>
<td>AND</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>ORR</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>EOR</td>
<td>Bitwise Exclusive-OR</td>
</tr>
<tr>
<td>BIC</td>
<td>Bit Clear</td>
</tr>
<tr>
<td>MUL</td>
<td>32-bit multiplication</td>
</tr>
<tr>
<td>MLA</td>
<td>32-bit multiplication and accumulate</td>
</tr>
<tr>
<td>SMULL</td>
<td>Signed 64-bit multiplication</td>
</tr>
<tr>
<td>SMLAL</td>
<td>Signed 64-bit multiplication with accumulate</td>
</tr>
<tr>
<td>UMULL</td>
<td>Unsigned 64-bit multiplication</td>
</tr>
<tr>
<td>UMLAL</td>
<td>Unsigned 64-bit multiplication with accumulate</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
</tr>
<tr>
<td>CMN</td>
<td>Compare Negative</td>
</tr>
<tr>
<td>TST</td>
<td>Test</td>
</tr>
<tr>
<td>TEQ</td>
<td>Test Equivalence</td>
</tr>
<tr>
<td>MOV</td>
<td>Move</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>MVN</td>
<td>Move Negated</td>
</tr>
<tr>
<td>B</td>
<td>Branch</td>
</tr>
<tr>
<td>BL</td>
<td>Branch and Link</td>
</tr>
<tr>
<td>BX</td>
<td>Branch and Exchange</td>
</tr>
<tr>
<td>LDR</td>
<td>Load 32-bit Value</td>
</tr>
<tr>
<td>LDRB</td>
<td>Load 8-bit Value</td>
</tr>
<tr>
<td>STR</td>
<td>Store 32-bit Value</td>
</tr>
<tr>
<td>STRB</td>
<td>Store 8-bit Value</td>
</tr>
<tr>
<td>LDM</td>
<td>Load Multiple Values</td>
</tr>
<tr>
<td>STM</td>
<td>Store Multiple Values</td>
</tr>
</tbody>
</table>
BIBLIOGRAPHY


