A HOL Specification of the ARM Instruction Set Architecture

Anthony Fox
Computer Laboratory, University of Cambridge

June 21, 2001

Abstract

This report gives details of a HOL specification of the ARM instruction set architecture. It is shown that the HOL proof tool provides a suitable environment in which to model the architecture. The specification is used to execute fragments of ARM code generated by an assembler. The specification is based primarily around the third version of the ARM architecture, and the intent is to provide a target semantics for future microprocessor verifications.

Contents

1 Introduction 4

2 Methodology 4
- 2.1 State Functions and Iterated Maps 4
- 2.2 Memory and Finite Maps 5

3 The ARM Programmer's Model 5
- 3.1 The State-Space 5
- 3.2 Configuration 5
- 3.3 Registers, Modes and Register Access 6
- 3.4 The Program-Counter 8
- 3.5 Memory Organisation and Access 10
- 3.6 Exceptions and Interrupts 12
- 3.7 Bit-vector Operations 13

4 The Instruction Set 15
- 4.1 Instruction Formats 15
- 4.2 Instruction Summary 15
- 4.3 Conditional Execution 17
- 4.4 Branch and Branch with Link 17
- 4.5 Data Processing 18
- 4.6 PSR Transfer 20
- 4.7 Multiply and Multiply-Accumulate 22
- 4.8 Single Data Transfer 23
- 4.9 Block Data Transfer 25

4.10 Single Data Swap 27
4.11 Software Interrupt 28

5 State and Next-State Functions 28

6 Executing the Specification 30

7 Future Work and Conclusions 32

8 Acknowledgments 32

List of Figures

1 ARM's visible registers. 7
2 Format of the program status registers (PSRs). 7
3 Little- and big-endian memory organisations. 11
4 Exception handling specification. 13
5 ARM instruction set formats. 15
6 Branch and branch with link instruction encoding. 17
7 Branch and branch with link instruction execution. 18
8 Data processing instruction encoding. 19
9 Data processing instruction execution. 20
10 MRS instruction encoding. 21
11 MSR instruction encoding. 21
12 MRS instruction execution. 21
13 MSR instruction execution. 22
14 Multiply instruction encoding. 22
15 Multiply instruction execution. 23
16 Single data transfer instruction encoding. 24
17 Single data transfer instruction execution. 24
18 Block data transfer instruction encoding. 25
19 Block data transfer instruction execution. 26
20 Swap instruction encoding. 27
21 Swap instruction execution. 28
22 Software interrupt instruction encoding. 28
23 Software interrupt instruction execution. 28
24 The ARM next-state function. 29

List of Tables

1 ARM operating modes and register usage. 6
2 Register access operations. 9
3 PSR operations. 9
4 Memory access operations. 11
5 Exception vector addresses. 12
6 Primitive operations over the natural numbers and 32-bit words. 14
7 The ARM instruction set. 16
8 ARM condition codes. 17
9 Operations used to specify branch instructions. 18
10 Operations used to specify data processing instructions. 19
11 Operations used to specify PSR transfer instructions. 21
12 Operations used to specify multiply and multiply-accumulate instructions. 22
13 Operations used to specify single data transfer instructions. 23
14 Operations used to specify block data transfer instructions. 26

1 Introduction

This report details the specification of the ARM instruction set architecture in HOL. The specification is primarily based on version 3G of the architecture, although some of the features of version 4 are adopted, for example, the inclusion of the system mode and the specification of unpredictable behaviour with respect to program-counter usage (see Section 3.4). The official ARM architecture reference is [10]; Furber's book is also a useful introductory text [5]. The specification presented in this report was influenced work at Leeds using SML, see [8].

HOL is founded on Church's theory of simple types [1], and has its origins in Edinburgh LCF [6] and Cambridge LCF [9]. The version of HOL used in the production of this report is HOL98 Taupo-6, which is written in Standard ML (specifically MoscowML). The current HOL distribution may be found at `www.cl.cam.ac.uk/Research/HVG/HOL`.

Section 2 gives a brief overview of the approach taken in modelling the architecture. In Section 3 the ARM programmer's model is discussed and it is shown how this may be translated into HOL. The programmer's model establishes a level of abstraction—the programmer's view of the architecture. Section 4 gives details about the encoding and semantics of the main the ARM instruction classes. This is all brought together in Section 5 with the definition of a next-state function, which provides an operational semantics for the architecture. Finally, in Section 6 it is shown how the specification may be used to simulate the execution of ARM machine code.

2 Methodology

This section outlines the approach used to model the ARM architecture. A more extensive account of this methodology, in the context of microprocessor specification and verification, may be found in [3].

2.1 State Functions and Iterated Maps

The ARM architecture is modelled as a finite state machine and is given an operational semantics. The set of all possible machine states, as perceived by the programmer, is called the *state-space* and this is defined in Section 3.1. The system is modelled by a *state function* $F : T \to A \to A$ that specifies the state of the machine at clock cycle $t \in T = \{0, 1, \dots\}$ for a given preliminary (pre-initialised) machine state $a \in A$. The state function for the ARM architecture is an *iterated map* and is defined in Section 5.

Definition. Let A be a non-empty set (the state-space), and let T be a set of cycles (the clock). A function $F : T \to A \to A$ is called an *iterated map* when it is defined by the equations:

$$\begin{aligned} F\,0\,a &= h\,a, \\ F\,(t+1)\,a &= f(F\,t\,a) \end{aligned}$$

where $h : A \to A$ is an *initialisation function* and $f : A \to A$ is a *next-state function*.

Iterated maps generate deterministic state sequences of the form:

$$h\ a,\ f(h\ a),\ f(f(h\ a)), \dots, f^t(h\ a), \dots$$

With the ARM architecture, an initialisation function is not used because all machine states are suitable initial states[1], therefore the state at time t is $F\ t\ a = f^t(a)$. One instruction is executed

---

[1] This is not strictly true because some processor states are best avoided. For example, to avoid unpredictable behaviour, it is important to ensure that the PSR control bits (see Section 3.3) are set appropriately.

on each successive clock cycle, therefore the state at cycle t corresponds with the execution of t instructions.

The specification presented in this report is intended to be a purely *abstract* model of the ARM instruction set architecture. The state-space only contains components that are visible to the programmer, and every effort is made to ensure that the specification is not too strong or over-specified (i.e. implementation specific). This is done to ensure that the specification represents a suitable target semantics for microprocessor implementations.

2.2 Memory and Finite Maps

Addressable memory is a core feature of most computer architectures and relatively large numbers of general purpose registers are available with most RISC machines, including the ARM. The functionality (if not the size and speed) of these two storage components is very similar and both are modelled as maps from a finite *address/index-space* to a *memory/register-cell-space*. HOL provides a theory of finite maps, but for the purposes of this work it is sufficient to model memory using a function `m:'a→'b`, where `'a` and `'b` are suitable (but not necessarily finite) indices and data types.

Function application corresponds with a memory read, with `m a` representing the contents of memory `m` at address `a`. A substitution function `SUBST:('a→'b)→'a#'b→'a→'b` is used to overwrite data at a given memory address, thus modelling a memory write:

$$\vdash_{def} \texttt{SUBST m (a,w) b = (if a = b then w else m b)}$$

For example, `SUBST m (a,w)` is the memory `m` with the value `w` stored at address `a`.

3 The ARM Programmer's Model

This section describes the ARM's programmer's model and its formalisation in HOL.

3.1 The State-Space

The ARM state-space consists of a main memory, general-purpose registers and processor status registers. This is represented in HOL using the following data type declaration:

```
Hol_datatype `state_ARM = ARM of (w30→w32)⇒reg⇒psr`;
```

The types `w32`, `reg` and `psr` are introduced in Section 3.3, and the main memory, of type `w30→w32`, is discussed in Section 3.5. The type declaration introduces a type constructor function:

```
ARM:(w30→w32)→reg→psr→state_ARM,
```

and this acts much like an ML type constructor.

3.2 Configuration

Like most microprocessors, ARM designs support different operational configurations, to account for various system setups, absence/presence of auxiliary hardware and versions of instruction code. Such configuration is inherently implementation dependent and is principally achieved by using a small number of special purpose registers that are hidden with respect to the programmer's model. Flags contained within these registers are read (and in some cases set) using co-processor instructions.

CPSR[4:0]	Mode	Use	Accessible register set	
10000	User	Normal user code	r0..r15	CPSR
10001	FIQ	Processing fast interrupts	r0..r7, r8_fiq..r14_fiq, r15	CPSR, SPSR_fiq
10010	IRQ	Processing standard interrupts	r0..r12, r13_irq, r14_irq, r15	CPSR, SPSR_irq
10011	Supervisor	Processing software interrupts	r0..r12, r13_svc, r14_svc, r15	CPSR, SPSR_svc
10111	Abort	Processing memory faults	r0..r12, r13_abt, r14_abt, r15	CPSR, SPSR_abt
11011	Undefined	Handling undefined exception traps	r0..r12, r13_und, r14_und, r15	CPSR, SPSR_und
11111	System	Running privileged operating system tasks	r0..r15	CPSR

Table 1: ARM operating modes and register usage.

The HOL specification in this report just models the abstract architecture and, as such, only a limited amount of configuration is visible.

Three boolean constants are defined:

- `ALIGN_CHECK` determines whether or not misaligned memory addresses generate exceptions, see Section 3.5;
- `BIG_ENDIAN` determines whether a big-endian or little-endian memory organisation is employed, see Section 3.5; and
- `LATE_ABORT` controls the write-back behaviour of load/store instructions after data aborts, see Sections 3.6 and 4.8.

The configuration is static (i.e. not reflected in the state-space), and hence term-rewriting may be used to simplify the specification, thus providing an instance of a given configuration.

3.3 Registers, Modes and Register Access

In the ARM programmer's model, access to registers is determined by the current *operating mode* of the processor. The current mode is encoded within five bits of a Current Processor Status Register (CPSR), see Table 1. Seven modes are defined and these provide a means to develop multi-user operating systems, and a mechanism to handle interrupts and exceptions. Figure 1 shows the full set of 37 visible 32-bit registers, as organised by operating mode. The program-counter is accessible in all modes and is treated as the general purpose register `r15`, see Section 3.4. The Processor Status Registers (PSRs) are used to keep track of the processor's operating mode, together with the interrupt status and the state of the condition code flags, see Figure 2. The CPSR directly controls the execution of instructions, whereas the Saved Processor Status Registers (SPSRs) are used to effect mode changes—saving the CPSR value for subsequent restoration. Conceptually the PSRs are thirty-two bits long but, in terms of the architecture presented here, bits five and 8–27 are all redundant.

Registers are represented in HOL using the type `w32`, which is introduced with the data type declaration:

```
Hol_datatype ‘w32 = W32 of num‘;
```

One advantage of this choice of representation is that words can be constructed from unsigned integer values, for example, the term "`W32 123`" represents the word 0x0000007B. By defining the

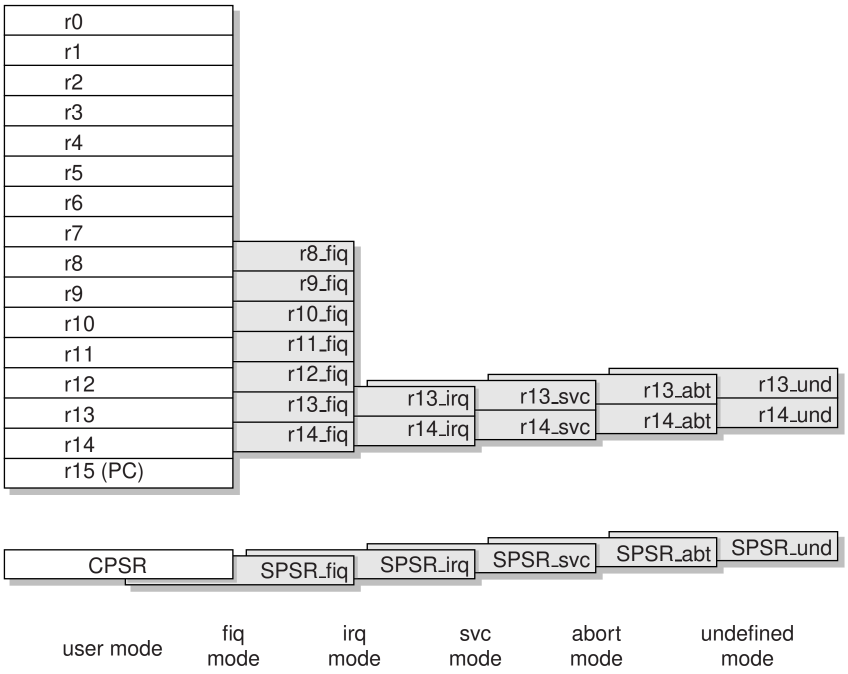

Figure 1: ARM's visible registers.

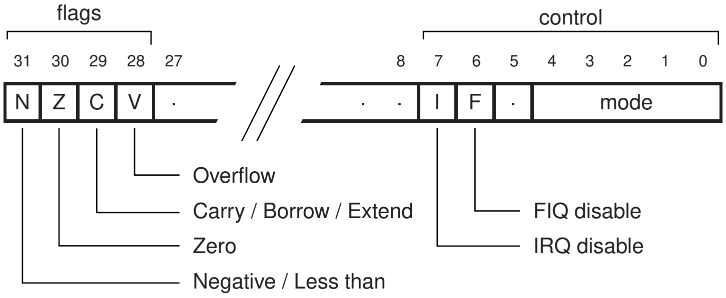

Figure 2: Format of the program status registers (PSRs).

type `w32` in this way, one can also make prudent use of HOL's built-in support for natural number arithmetic—this simplifies the task of defining operations over words, see Section 3.7. One should be aware that with this approach term equivalence does not correspond with word equivalence because when one writes "`W32` n " this is intended to mean the word with unsigned value $n \bmod 2^{32}$. In practice this is not a problem because all operations over words are defined so as to produce results in the desired range: $0 \leq n < 2^{32}$.

An alternative approach would have been to use lists or vectors/tuples of boolean values to represent words. The advantage of using vectors or tuples is that the data type may be of the correct cardinality (with exactly 2^{32} possible values), hence term equivalence is always word equivalence. Using lists helps when defining bit manipulation operations (such as, concatenation and bitwise and/or/eor) but arithmetic operations (such as, addition and multiplication) are more involved and would require a proof of correctness. There is also the problem that term equivalence is not word equivalence even for values less than 2^{32}. This is because leading zeroes may be present, for example, with the least-significant bits first, the list `[0;1]` has the same word value as the list `[0;1;0]`.

ARM's banks of visible registers are represented in HOL using the data types `reg` and `psr`:

```
Hol_datatype `reg = REG of (reg_usr→w32)⇒(reg_fiq→w32)⇒(reg_irq→w32)⇒
                           (reg_svc→w32)⇒(reg_abt→w32)⇒(reg_und→w32)`;
Hol_datatype `psr = PSR of w32⇒(spsr→w32)`;
```

The general-purpose registers are modelled as a group of six maps from register indices to words, and the PSRs are modelled as a single word (the CPSR) together with a map from SPSR names to words. User registers are indexed by 4-bit words and all other register are named; the following data type declarations achieve this:

```
Hol_datatype `reg_usr = W4 of num`;
Hol_datatype `reg_fiq = r8_fiq | r9_fiq | r10_fiq | r11_fiq | r12_fiq | r13_fiq | r14_fiq`;
Hol_datatype `reg_irq = r13_irq | r14_irq`;
Hol_datatype `reg_svc = r13_svc | r14_svc`;
Hol_datatype `reg_abt = r13_abt | r14_abt`;
Hol_datatype `reg_und = r13_und | r14_und`;
Hol_datatype `spsr = spsr_fiq | spsr_irq | spsr_svc | spsr_abt | spsr_und`;
```

Note that "`W4 16`" is a valid term but it is not a valid register index: this is not a problem because the ARM specification ensures that all references to user registers are made using numbers in the range 0–15. Table 2 describes the main functions used to access registers. The operational modes are defined using the following data type:

```
Hol_datatype `mode = usr | fiq | irq | svc | abt | und | sys`;
```

The system mode `sys` was introduced in version 4 of the ARM architecture.

Operations for manipulating PSRs are listed in Table 3. The function `DECODE_PSR` is used to decode a PSR (see Figure 2) and produce a tuple of the from:

```
(Z,N,C,V,irq,fiq,mode,valid) = DECODE_PSR w
```

The last element of this tuple is only true when the 5-bit encoding of the mode in `w` is correct i.e. when it is one the encodings in Table 1.

3.4 The Program-Counter

The program-counter is visible to the programmer and may be accessed as `r15`. When accessing the program-counter pipelined functionality becomes apparent. The behaviour of the program-counter is based around that of a 3-stage pipelined microprocessor, such as the ARM6. The state

Operation	Type	Meaning / Use
`CPSR_READ`	`psr→w32`	Gives the state of the CPSR
`CPSR_WRITE`	`psr→w32→psr`	Updates the state of the CPSR. The mode is forced to be valid by mapping erroneous codes into the user mode
`SPSR_READ`	`psr→mode→w32`	Gives the state of the SPSR corresponding with a given mode
`SPSR_WRITE`	`psr→mode→w32→psr`	Updates the state of an SPSR with a new value
`REG_READ`	`reg→mode→num→w32`	Gives the state of general purpose register, as referenced by mode and register number. The program-counter is incremented by eight, this corresponds with a data read for a 3-stage pipeline.
`REG_WRITE`	`reg→mode→num→w32→reg`	Updates the state of a register with a new value, the register is referenced by mode and number
`INC_PC`	`reg→reg`	Increments the program-counter so as to point to the next instruction i.e. adds four to its value
`FETCH_PC`	`reg→w32`	Gives the actual state of the program counter

Table 2: Register access operations.

Operation	Type	Meaning / Use
`SET_NZC`	`bool→bool→bool→w32→w32`	Set the N, Z and C flags of a PSR
`SET_NZCV`	`bool→bool→bool→bool→w32→w32`	Set all of the condition code flags of a PSR
`SET_MODE`	`mode→w32→w32`	Set the mode bits of a PSR
`SET_IFMODE`	`bool→bool→mode→w32→w32`	Set the interrupt status and mode bits of a PSR
`DECODE_PSR`	`w32→bool#bool#bool#bool#mode#bool`	Decode a PSR

Table 3: PSR operations.

of the program-counter is used to fetch instructions, but by the time a given instruction reaches the execute stage of the pipeline the program-counter will have a value that is eight bytes (two instructions) ahead of instruction's own address. In the HOL specification, the function `REG_READ` gives the state of the program-counter at the execute stage and `FETCH_PC` gives the state of the program-counter at the fetch stage:

```
REG_READ reg usr (W4 15) = (FETCH_PC reg) ADD32 (W32 8)
```

Further complication arises with program-counter access when dealing with instructions that require more than one cycle to execute. The program-counter is incremented on the first cycle of execution and hence subsequent access to the program-counter (by the same instruction) will yield a value of $pc + 12$, where pc is the address of the instruction being executed. After version 4 of the architecture, program-counter usage that gives rise to this type of behaviour is deemed to be unpredictable and is strongly discouraged.

Although such program-counter functionality made it easier to implement 3-stage pipelines, it has resulted in further complicating the implementation of later designs. For example, to be compatible, a 5-stage pipeline must simulate the program-counter behaviour of the 3-stage pipeline. This need not have been the case and could have been avoided by imposing a clean abstraction from the outset. There would be less of a problem if the programmer's model view of the program-counter were independent to that of an pipelined implementation i.e. by specifying that the program-counter `r15` always represents the address of the instruction being executed. It would then have been a requirement of the implementation to provide this straightforward functionality. In the case of pipelined designs this might simply involve using two registers: one of which is visible to the programmer (the address of the instruction being executed, which is incremented only at the end of instruction execution), and the other is a hidden register, which is used to fetch instructions and is incremented after an instruction is fetched.

3.5 Memory Organisation and Access

The ARM memory system may be viewed as a linear array of bytes, numbered from zero to $2^{32} - 1$. This could be modelled in HOL as a map from `w32` to `w8`, where `w8` is represents bytes. Instead the memory is modelled as a map from `w30` to `w32`, where `w30` is given the data type declaration:

```
Hol_datatype `w30 = W30 of num`;
```

The advantage of modelling the memory using word addressing is that, when loading and storing words, byte ordering is less significant and one need not rely on mappings between 4-tuples of bytes and words. This helps improve the performance of simulation, where words are the most common form of data to be accessed. The byte ordering is only of significance when accessing bytes or when loading data using misaligned memory addresses, see Figure 3. A 32-bit byte address is word aligned if, and only if, it is exactly four times greater than a 30-bit word address. When loading a word from a misaligned address $4n + q$, for 30-bit value n and $1 \leq q \leq 3$, the word with address n is fetched and is either rotated $8q$ places left with the big-endian scheme (to ensure that the q^{th} byte is the most-significant) or rotated $8q$ places right with the little-endian scheme (to ensure that the q^{th} byte is the least-significant). When storing a word using a misaligned address, word alignment is simply forced i.e. no word transformation takes place.

Table 4 describes the functions used to model the data access operations of the ARM architecture. The memory read operations produce a pair consisting of a word (in the case of byte access this is a zero extended byte value); and a boolean flag, which indicates whether or not the operation generated an exception. For example,

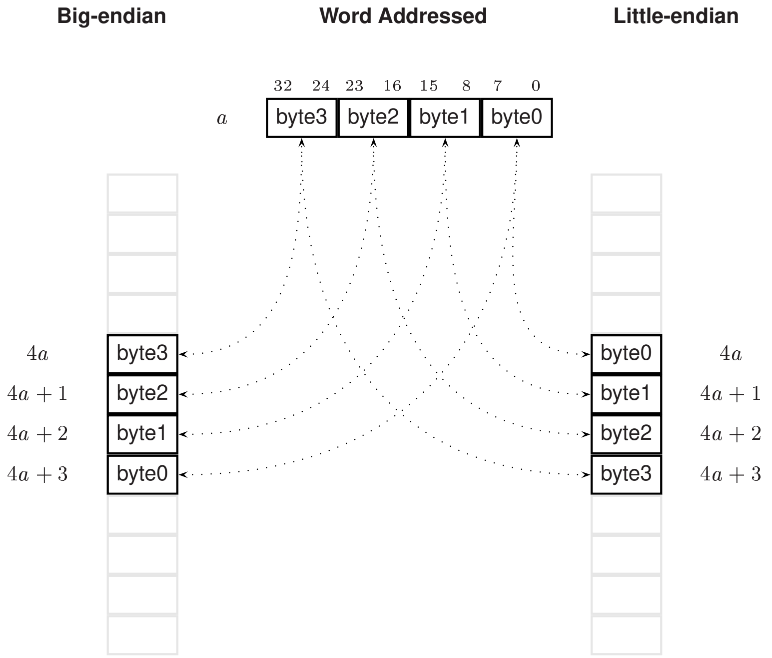

Figure 3: Little- and big-endian memory organisations.

Operation	Type	Meaning / Use
`WORD_ALIGN`	`w32→w32`	Force word alignment by clearing the two least significant bits.
`W32_W30`	`w32→w30`	Convert a 32-bit word of value n into a 30-bit word of value $\left\lfloor \frac{n}{4} \right\rfloor$
`MEM_READ_BYTE`	`(w30→w32)→w32→w32#bool`	Models loading a byte from memory
`MEM_READ_WORD`	`(w30→w32)→w32→w32#bool`	Models loading a word from memory
`MEM_WRITE_BYTE`	`(w30→w32)→w32→w32→(w30→w32)#bool`	Models storing a byte to memory
`MEM_WRITE_WORD`	`(w30→w32)→w32→w32→(w30→w32)#bool`	Models storing a word to memory

Table 4: Memory access operations.

Exception	Mode on entry	Vector address
Reset	Supervisor	0x00000000
Undefined instruction	Undefined	0x00000004
Software interrupt (SWI)	Supervisor	0x00000008
Prefetch abort (instruction fetch memory fault)	Abort	0x0000000C
Data abort (data access memory fault)	Abort	0x00000010
- reserved -	-	0x00000014
IRQ (normal interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

Table 5: Exception vector addresses.

```
(word,abort) = MEM_READ_WORD mem (W32 17)
```

models fetching a word from the misaligned 32-bit address 17. The first component `word` evaluates to the forth word in memory `mem` rotated eight places left, if `BIG_ENDIAN` is true, and eight places right otherwise. The `abort` flag evaluates to the constant value `ALIGN_CHECK`, see Section 3.2.

Memory write operations also produce a pair, this time consisting of the updated memory, and a flag to indicate the exception status. For example,

```
(mem',abort) = MEM_WRITE_BYTE mem word (W32 17)
```

models storing the least-significant byte (bits 0–7) of `word` in the memory `mem` at the address 17. The word at address 4 is altered: if `BIG_ENDIAN` is true then the third byte (bits 16–23) is overwritten, otherwise the second byte (bits 8–15) is overwritten. The updated memory is `mem'`, and the `abort` flag is the exception status. With respect to raising exceptions, the definitions of the memory access operations may be tailored to correspond with a given implementation. For example, to consider the case of a memory management unit that has access constraints to ensure system code is protected. The only architecture specific exception is loading a word using a misaligned address with `ALIGN_CHECK` set.

3.6 Exceptions and Interrupts

The specification presented here makes little or no attempt to model the intricacies of exception handling in an accurate way i.e. their precise timing and priorities. Seven types of exception are considered and these are listed in Table 5. The processor's response to a given exception type (as numbered from 0 to 7) is specified by the function `EXCEPTION` defined in Figure 4. The effect of the exception on the processor's state is as follows:

- The memory does not change state.
- Register 14 is assigned the value of the program-counter plus four (i.e. the address of the following instruction).
- The program-counter is assigned the vector address associated with the given exception.
- The CPSR is stored in the SPSR associated with the given exception.
- The CPSR mode bits are changed to correspond with the mode of the given exception.
- The CPSR is updated to mask out IRQ interrupts (bit 7 is set).

```
⊢def EXCEPTION (ARM mem reg psr) n =
      let cpsr = CPSR_READ psr in
      let fiq' = if (n = 1) ∨ (n = 7) then T else BIT32 6 cpsr
      and (mode',pc') =
           if n = 1 then (svc,W32 0)  else
           if n = 2 then (und,W32 4)  else
           if n = 3 then (svc,W32 8)  else
           if n = 4 then (abt,W32 12) else
           if n = 5 then (abt,W32 16) else
           if n = 6 then (irq,W32 24) else
           if n = 7 then (fiq,W32 28) else ARB in
     let reg' = REG_WRITE reg mode' 14 ((FETCH_PC reg) ADD32 (W32 4)) in
       ARM mem (REG_WRITE reg' usr 15 pc')
         (CPSR_WRITE (SPSR_WRITE psr mode' cpsr) (SET_IFMODE T fiq' mode' cpsr))
```

Figure 4: Exception handling specification.

- If the exception is a reset or a FIQ interrupt then the CPSR is updated to mask out FIQ interrupts.

Jumping to the specified vector addresses will activate exception handling code—this should deal with and return from the exception in a acceptable manner. The nature and requirements of this code is not specified here.

3.7 Bit-vector Operations

In order to specify the ARM architecture a number of word operations are defined, see Table 6. The first six operations are defined over the natural numbers and are primarily used to decode words. For example,

```
BITS 8 4 321 = 20
DIVMOD_2EXP 4 321 = (20,1)
```

because

$$321_{10} = 101000001_2 \text{ and}$$
$$20_{10} = 10100_2.$$

The operations over words can be expressed using natural number arithmetic, for example, the following axioms are used:

```
⊢def ONE_COMP32 (W32 x) = W32 (2 EXP 32 - 1 - x)
⊢def ADD32 (W32 x) (W32 y) = W32 ((x + y) MOD 2 EXP 32)
⊢def LSL32 (W32 x) n = W32 ((x * 2 EXP n) MOD 2 EXP 32)
```

If appropriate, theorems are partially evaluated prior to use, for example:

```
⊢def ONE_COMP32 (W32 x) = W32 (4294967295 - x)
```

The bitwise operations `AND32`, `OR32` and `EOR32` are all defined using a function:

```
BITWISE32:(bool→bool→bool)→w32→w32→w32
```

This function takes a boolean operation and uses it in comparing the oddness of the two arguments at powers of two.

Operation	Type	Meaning / Use
`TIMES_2EXP m n`	`num→num→num`	Shift left, i.e. $n \times 2^m$
`DIV_2EXP m n`	`num→num→num`	Shift right, i.e. $\left\lfloor \frac{n}{2^m} \right\rfloor$
`MOD_2EXP m n`	`num→num→num`	Select least significant bits, i.e. $n \bmod 2^m$
`DIVMOD_2EXP m n`	`num→num→num#num`	Bitwise quotient and remainder, i.e. $(\left\lfloor \frac{n}{2^m} \right\rfloor, n \bmod 2^m)$
`BITS h l n`	`num→num→num→num`	Extract bit field, i.e. $\left\lfloor \frac{n}{2^l} \right\rfloor \bmod 2^{h-l+1}$
`BIT m n`	`num→num→bool`	Test bit, i.e. true if m^{th} bit of n is equal to one
`BIT32 n w`	`num→w32→bool`	Test bit
`LSB32 w`	`w32→bool`	Test least significant bit
`MSB32 w`	`w32→bool`	Test most significant bit
`W32_NUM w`	`w32→num`	Unsigned value
`ONE_COMP32 w`	`w32→w32`	Bitwise compliment
`TWO_COMP32 w`	`w32→w32`	Arithmetic negation
`w ADD32 v`	`w32→w32→w32`	Addition
`w MUL32 v`	`w32→w32→w32`	Multiplication
`w SUB32 v`	`w32→w32→w32`	Subtraction
`w AND32 v`	`w32→w32→w32`	Bitwise and
`w OR32 v`	`w32→w32→w32`	Bitwise or
`w EOR32 v`	`w32→w32→w32`	Bitwise exclusive or
`w LSL32 n`	`w32→num→w32`	Logical shift left (n places)
`w LSR32 n`	`w32→num→w32`	Logical shift right (n places)
`w ASR32 n`	`w32→num→w32`	Arithmetic shift right (n places)
`w ROR32 n`	`w32→num→w32`	Rotate right (n places)
`w ROL32 n`	`w32→num→w32`	Rotate left (n places)
`w RRX32 b`	`w32→bool→w32`	Shift right using extend bit b

Table 6: Primitive operations over the natural numbers and 32-bit words.

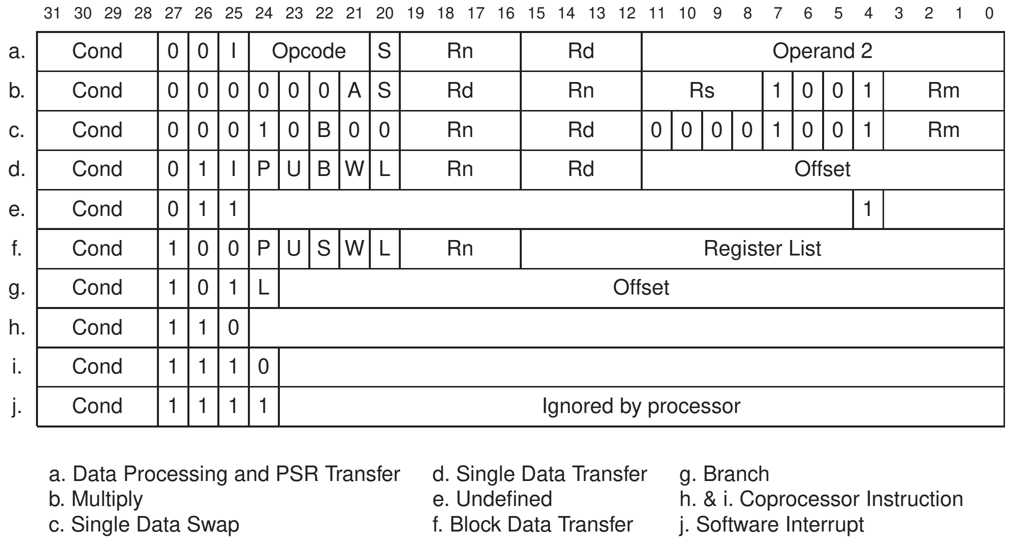

Figure 5: ARM instruction set formats.

4 The Instruction Set

In this section the bulk of the ARM instruction set is specified. Eight classes of instructions are considered; co-processor and thumb instructions are not covered.

4.1 Instruction Formats

The encoding of ARM instructions is shown in Figure 5. Instructions in the undefined instruction class, together with the co-processor classes (labelled e., h. and i.) are all trapped and raise the undefined instruction exception (see Section 3.6). Some instruction codes give unpredictable (implementation dependent) behaviour and others are not defined at all, but these do not raise an exception. These *holes* in the instruction-space have in some cases been used to add more instructions in later versions of the architecture. For the purposes of the HOL specification, the result of executing unpredictable instructions is expressed using `ARB`, which has HOL definition

$$\vdash_{def} \texttt{ARB = } \varepsilon\texttt{x. T}$$

In the case of state, this represents a value `x:state_ARM` such that `T` (truth) holds, which means that it denotes an arbitrary state.

The HOL specification will name the components of the ARM instructions as they appear in the literature (and figures in this report), but there are two notable exceptions due to overloading—the flags I and S must be called `Im` and `Sf` because `I` is the identity map and `S` is a combinator.

4.2 Instruction Summary

A complete list of the specified instructions is presented in Table 7. The third column provides an informal description of each instruction, mainly in terms of assignments to state components. No attempt is made in this report to desribe the full assembly code syntax for each class on instruction, this may be found in the references [10, 5].

Mnemonic	Instruction	Action
ADC	Add with carry	Rd := Rn + Op2 + Carry
ADD	Add	Rd := Rn + Op2
AND	AND	Rd := Rn AND Op2
B	Branch	R15 := address
BIC	Bit clear	Rd := Rn AND NOT Op2
BL	Branch with link	R14 := R15, R15 := address
CMN	Compare negative	CPSR flags := Rn + Op2
CMP	Compare	CPSR flags := Rn - Op2
EOR	Exclusive OR	Rd := (Rn AND NOT Op2) OR (Op2 AND NOT Rn)
LDM	Load multiple registers	Stack manipulation (Pop)
LDR	Load register from memory	Rd := (address)
MLA	Multiply accumulate	Rd := (Rm * Rs) + Rn
MOV	Move register or constant	Rd := Op2
MRS	Move register to PSR status/flags	Rn := PSR
MSR	Move register to PSR status/flags	PSR := Rm
MLA	Multiply	Rd := Rm * Rs
MVN	Move negated register	Rd := NOT Op2
ORR	ORR	Rd := Rn OR Op2
RSB	Reverse subtract	Rd := Op2 - Rn
RSC	Reverse subtract with carry	Rd := Op2 - Rn - 1 + Carry
RSB	Subtract with carry	Rd := Rn - Op2 -1 + Carry
STM	Store multiple	Stack manipulation (Push)
STR	Store register to memory	<address> := Rd
SUB	Subtract	Rd := Rn - Op2
SWI	Software interrupt	OS call
SWP	Swap register with memory	Rd := [Rn], [Rn] := Rm
TEQ	Test bitwise equality	CPSR flags := Rn EOR Op2
TST	Test bits	CPSR flags := Rn AND Op2

Table 7: The ARM instruction set.

Code	**Suffix**	**Flags**	**Interpretation**
0000	EQ	Z set	Equal / equals zero
0001	NE	Z clear	Not equal
0010	CS/HS	C set	Carry set / unsigned higher or same
0011	CC/LO	C clear	Carry clear / unsigned lower
0100	MI	N set	Minus / negative
0101	PL	N clear	Plus / positive or zero
0110	VS	V set	Overflow
0111	VC	V clear	No overflow
1000	HI	C set and Z clear	Unsigned higher
1001	LS	C clear and Z set	Unsigned lower or same
1010	GE	N equals V	Signed greater than or equal
1011	LT	N is not equal to V	Signed less than
1100	GT	Z clear and N equals V	Signed greater than
1101	LE	Z set and N is not equal to V	Signed less than or equal
1110	AL	any	Always
1111	NV	none	Never (use prohibited)

Table 8: ARM condition codes.

Figure 6: Branch and branch with link instruction encoding.

4.3 Conditional Execution

One of the characteristic features of the ARM architecture is that all instructions are conditionally executed. This facilitates more compact and efficient assembly code control structures by reducing the need for conditional branching. Bits 28-31 of an instruction store the condition code (see Figure 5) and this is interpreted according to Table 8. Conditional execution is specified in HOL using a predicate function:

`CONDITION_PASSED:bool→bool→bool→bool→num→bool`

Each instruction is executed if, and only if, `CONDITION_PASSED N Z C V Cond` is true, where `N`, `Z`, `C` and `V` are the condition flags (stored in the CPSR) and `Cond` is the instruction's condition code expressed as value in the range 0-14.

4.4 Branch and Branch with Link

This instruction class includes: regular branches (which, like all other instructions, are conditionally executed); and subroutine style branches, which store the return address in a link register. The encoding of these instructions is shown in Figure 6. This class of instruction is specified in HOL using the functions from Table 9. The function `BRANCH`, defined in Figure 7, specifies the state of the processor after executing a branch instruction:

- An external decode determines the operating mode and instruction code (see Section 5), and these are denoted by `mode` and `n` respectively.

Operation	Type	Meaning / Use
`BRANCH`	`state_ARM→mode→num→state_ARM`	Gives the state after executing a branch instruction
`DECODE_BRANCH`	`num→bool#num`	Decodes a branch giving the link flag and 24-bit signed offset
`SIGN_EX_OFFSET`	`num→w32`	Takes a 26-bit signed value and gives the 32-bit word equivalent

Table 9: Operations used to specify branch instructions.

```
⊢def BRANCH (ARM mem reg psr) mode n =
      let (L,offset) = DECODE_BRANCH n
      and pc = REG_READ usr reg 15 in
      let pc' = pc ADD32 (SIGN_EX_OFFSET (4*offset)) in
      if ¬(MSB32 pc = MSB32 pc') then ARB
      else
        let reg' = REG_WRITE usr reg 15 pc' in
        if L then
          ARM mem (REG_WRITE reg' mode 14 ((FETCH_PC reg) ADD32 (W32 4))) psr
        else
          ARM mem reg' psr
```

Figure 7: Branch and branch with link instruction execution.

- The 24-bit offset is shifted left two places, sign-extended and then added to the program-counter. Note that the current value of the program-counter is obtained using `REG_READ`, hence it is eight bytes ahead of the actual program-counter value.
- If the link form of the instruction is specified (`L` is set) then the address of the instruction following the branch is stored in register fourteen.

The memory and program status registers do not change state. An attempt to branch outside the 32-bit address-space may give unpredictable results—this is detected by checking whether or not the branch destination address has the same sign as the instruction's own address.

4.5 Data Processing

There are sixteen different data processing instructions, and their encoding is illustrated in Figure 8. The operation is determined by the 4-bit opcode field. A three-address format is employed: the destination register and first operand are expressed directly; and the second operand is computed using one of three methods. The operations used to specify the data processing instructions are listed in Table 10. The function `DATA_PROCESSING`, defined in Figure 9, specifies the state of the processor after executing a data processing instruction:

- An external decode (see Section 5) determines the operating mode and instruction code `n`, together with the status of the carry-out `C` and overflow `V` flags.
- The memory does not change state.
- The first operand `rn` is the state of register `Rn`.
- The pair `(C_s,op2)`, consists of the carry out from the shifter and the second operand, and is computed using the function `ADDR_MODE1`.

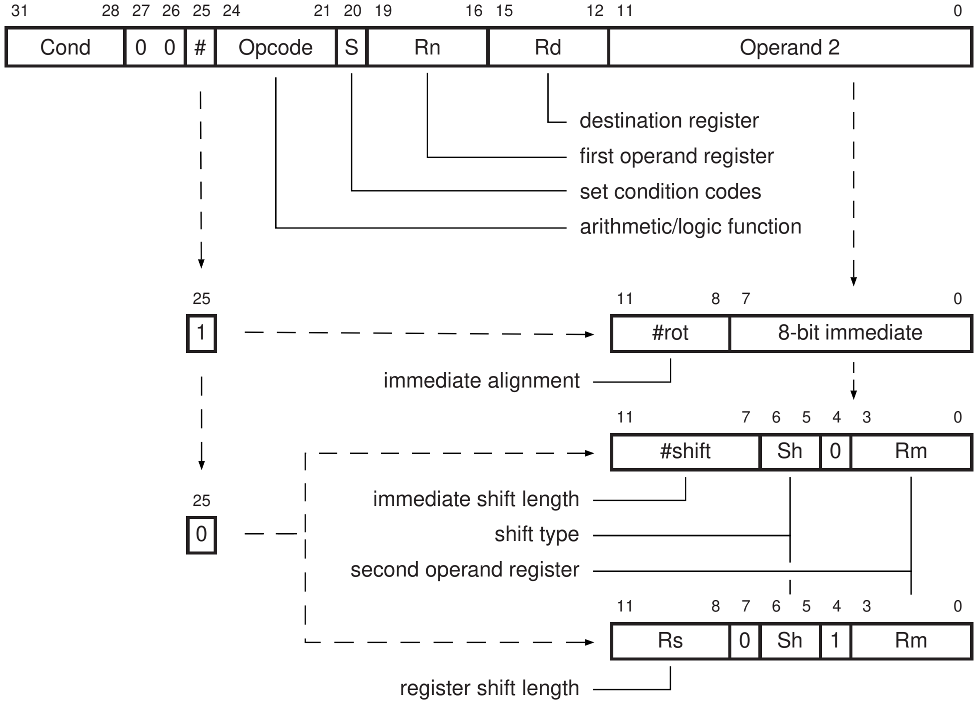

Figure 8: Data processing instruction encoding.

Operation	Type	Meaning / Use
DATA_PROCESSING	state_ARM→bool→bool→mode→num→state_ARM	Gives the state after executing a data processing instruction
DECODE_DATAP	num→bool#num#bool#num#num#num	Decodes the data processing instructions into six fields
ADDR_MODE1	reg→mode→bool→bool→num→bool#w32	Computes the carry out from the shifter and second source operand, using the register bank state, current mode, carry flag status, immediate flag and 12-bit operand fields. This function makes use of the following three functions
IMMEDIATE	bool→num→bool#w32	Compute immediate shift operand
SHIFT_IMMEDIATE	reg→mode→bool→num→bool#w32	Compute shift by immediate operand
SHIFT_REGISTER	reg→mode→bool→num→bool#w32	Compute shift by register operand
ALU_arithmetic	num→w32→w32→bool→bool#bool#bool#bool#w32	Gives the flag status and result for the execution of an arithmetic operation. Takes the opcode, two operands and the carry flag status
ALU_logical	num→w32→w32→bool→bool→bool#bool#bool#bool#w32	Gives the flag status and result for the execution of a logical operation. The overflow flag is not set by logical operations, so the current value must be provided
ARITHMETIC	num→bool	Holds true if the given opcode is that of an arithmetic (as opposed to logical) operation
TEST_OR_COMP	num→bool	Holds true if the given opcode is that of a test or comparison operation. These operations set the condition flags without storing a result

Table 10: Operations used to specify data processing instructions.

```
⊢def DATA_PROCESSING (ARM mem reg psr) C V mode n =
    let (Im,opcode,Sf,Rn,Rd,opnd2) = DECODE_DATAP n in
    if ((¬Im ∧ BIT 4 opnd2 ∧
        ((Rn = 15) ∨ (BITS 11 8 opnd2 = 15) ∨ (BITS 3 0 opnd2 = 15))) ∨
       (Rd = 15) ∧ Sf ∧ ((mode = usr) ∨ (mode = sys))) then ARB
    else
      let (C_s,op2) = ADDR_MODE1 reg mode C Im opnd2
      and rn = REG_READ reg mode Rn in
      let (N,Z,C',V',res) = if ARITHMETIC opcode
                              then ALU_arithmetic opcode rn op2 C
                              else ALU_logical opcode rn op2 C_s V in
        ARM mem (if TEST_OR_COMP opcode then
                   INC_PC reg
                 else let reg' = REG_WRITE reg mode Rd res in
                   if Rd = 15 then reg' else INC_PC reg')
          (if Sf then
             CPSR_WRITE psr
               (if Rd = 15 then SPSR_READ psr mode
                           else SET_NZCV N Z C' V' (CPSR_READ psr))
           else psr)
```

Figure 9: Data processing instruction execution.

- The ALU output `(N,Z,C',V',res)` is computed by the function `ALU_arithmetic` if the operation is arithmetic (SUB, ADD, RSB, ADC, SBC, RSC, CMP or CMN) and by the function `ALU_logic` otherwise (AND, EOR, ORR, MOV, BIC, MVN, TST and TEQ).
- If the operation is a test or comparison (CMP, CMN, TST or TEQ) then `Sf` must hold (see below), the program-counter is incremented and no other register is updated.
- If the operation is not a test or comparison then the result `res` will be written to the register `Rd`, and if this register is not `r15` then the program-counter will also be incremented.
- If the `Sf` flag is set then, if `Rd = 15` then the CPSR will be updated to the current mode's SPSR value, otherwise the CPSR flags are set in accordance with the output of the ALU.

Unpredictable behaviour arises when shifting by a register value when one of the source registers is the program-counter; and when trying to restore a SPSR (`Sf` set and `Rd=15`) when in the user or system mode.

A fair amount of complexity is hidden under the definition of `ADDR_MODE1`, which computes the second operand. For example, in the shift by immediate/register functions, four types of shifting are encoded using the `Sh` field (bits 5 and 6) but four more variants are available for the cases when the shift amount is zero.

4.6 PSR Transfer

This class of instruction includes the MRS instruction, which copies a PSR register to a general purpose register, and the MSR instruction, which is used to update a PSR using a register or immediate value. The encoding for these instructions is shown in Figures 10 and 11 respectively. The operations used to define this class of instruction are listed in Table 11.

The function `MRS` is defined in Figure 12 and it specifies the state after executing an MRS instruction: if the `R` bit is set then the SPSR is read and stored in register `Rd`, otherwise the CPSR is read and stored. The program-counter is always incremented. This operation is unpredictable when selecting an SPSR read in the user or system modes, or if the destination register is the program-counter.

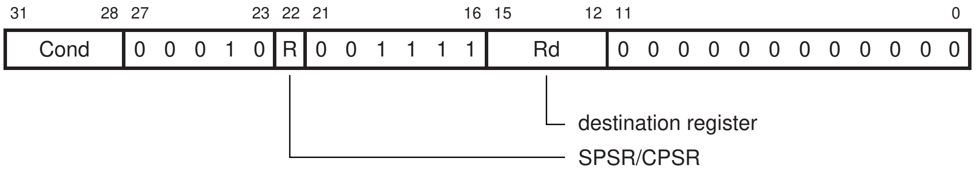

Figure 10: MRS instruction encoding.

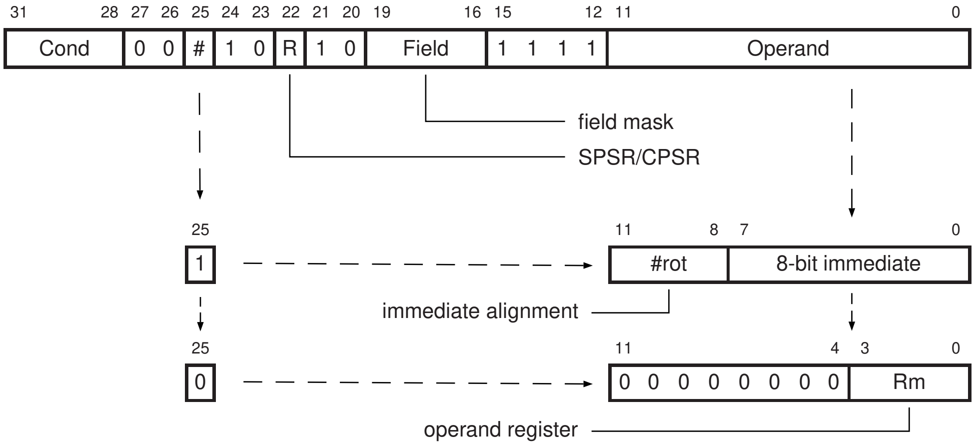

Figure 11: MSR instruction encoding.

Operation	Type	Meaning / Use
`MRS`	`state_ARM→mode→num→state_ARM`	Gives the state after executing an MRS instruction
`DECODE_MRS`	`num→bool#num`	Decodes an MRS instruction
`MSR`	`state_ARM→mode→num→state_ARM`	Gives the state after executing an MSR instruction
`DECODE_MSR`	`num→bool#bool#bool#bool#num#num`	Decodes an MSR instruction into six fields
`SPLIT_WORD`	`w32→w8#w8#w8#w8`	Split a word into four bytes
`CONCAT_BYTES`	`w8→w8→w8→w8→w32`	Construct a word from four bytes (highest byte first)

Table 11: Operations used to specify PSR transfer instructions.

```
⊢def MRS (ARM mem reg psr) mode n =
     let (R,Rd) = DECODE_MRS n in
     if (R ∧ ((mode = usr) ∨ (mode = sys))) ∨ (Rd = 15) then ARB
     else
       let word = if R then SPSR_READ psr mode else CPSR_READ psr in
         ARM mem (INC_PC (REG_WRITE reg mode Rd word)) psr
```

Figure 12: MRS instruction execution.

```
⊢def MSR (ARM mem reg psr) mode n =
  let (Im,R,bit19,bit16,Rm,opnd) = DECODE_MSR n in
  if (Im ∧ bit16) ∨ (R ∧ ((mode = usr) ∨ (mode = sys))) ∨ (¬Im ∧ (Rm = 15)) then ARB
  else
    let xpsr = if R then SPSR_READ psr mode else CPSR_READ psr
    and  src = if Im then SND (IMMEDIATE ARB opnd) else REG_READ reg mode Rm in
    let (x3,x2,x1,x0) = SPLIT_WORD xpsr
    and (s3,s2,s1,s0) = SPLIT_WORD src in
    let xpsr' = CONCAT_BYTES (if bit19 then s3 else x3) x2 x1
                         (if bit16 ∧ ¬(mode = usr) then s0 else x0) in
      ARM mem (INC_PC reg)
        (if R then SPSR_WRITE psr mode xpsr' else CPSR_WRITE psr xpsr')
```

Figure 13: MSR instruction execution.

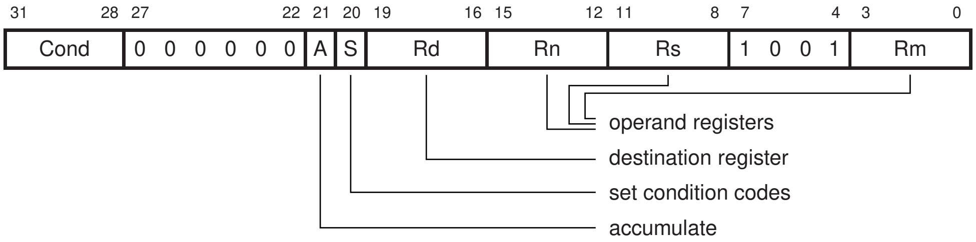

Figure 14: Multiply instruction encoding.

The function `MSR` is defined is Figure 13 and it specifies the state after executing an `MSR` instruction. When the `R` bit is set an `SPSR` update occurs, otherwise the `CPSR` value is modified. The source value is determined by the `Im` bit, if it is set then an immediate value is computed from the operand field `opnd` (using the function `IMMEDIATE` from Section 4.5), otherwise register `Rd` is read. The functions `SPLIT_WORD` and `CONCAT_BYTES` are used to construct a new `PSR` value by updating the control and/or flags parts of the status register. Bits 16 and 19 (of the field mask) are used to select the bytes to be updated, but the control bits can only be updated when not in user mode. `MSR` instructions are unpredictable with `SPSR` transfer in the user or system modes, when selecting control bit update with an immediate offset, or when the program-counter is the source register `Rm`. The main memory does not change state and the program-counter is incremented.

4.7 Multiply and Multiply-Accumulate

This class contains two instructions: multiply and multiply-accumulate. The encoding of these instructions is shown in Figure 14. The operations used to specify this class of instruction are listed in Table 12. The function `MUL_MLA`, defined in Figure 15, specifies the state of the processor after

Operation	**Type**	**Meaning / Use**
`MUL_MLA`	`state_ARM→mode→num→state_ARM`	Gives the state after executing a MUL or MLA instruction
`DECODE_MUL_MLA`	`num→bool#bool#num#num#num#num`	Decodes a MUL an MLA instructions
`ALU_multiply`	`bool→w32→w32→w32→bool#bool#bool#w32`	Gives the flag status and result of executing a MUL or MLA instruction

Table 12: Operations used to specify multiply and multiply-accumulate instructions.

```
⊢def MUL_MLA (ARM mem reg psr) mode n =
  let (A,Sf,Rd,Rn,Rs,Rm) = DECODE_MUL_MLA n in
  if (Rd = Rm) ∨ (Rd = 15) ∨ (A ∧ (Rn = 15)) ∨ (Rs = 15) ∨ (Rm = 15) then ARB
  else
    let rm = REG_READ reg mode Rm
    and rs = REG_READ reg mode Rs
    and rn = REG_READ reg mode Rn in
    let (N,Z,C,res) = ALU_multiply A rm rs rn in
      ARM mem (INC_PC (REG_WRITE reg mode res Rd))
          (if Sf then CPSR_WRITE psr (SET_NZC N Z C (CPSR_READ psr)) else psr)
```

Figure 15: Multiply instruction execution.

Operation	Type	Meaning / Use
`LDR_STR`	`state_ARM→bool→mode→num→state_ARM`	Gives the state after executing a LDR or STR instruction
`DECODE_LDR_STR`	`num→bool#bool#bool#bool#bool#bool#num#num#num`	Decodes LDR and STR instructions
`ADDR_MODE2`	`reg→mode→bool→bool→bool→bool→num→num→w32#w32`	Computes the memory address operand and write-back address

Table 13: Operations used to specify single data transfer instructions.

executing a multiply or multiply-accumulate instruction:

- The memory does not change state.
- The ALU output `(N,Z,C,res)` is computed using the function `ALU_multiply`. The multiplicand is register `Rm`, and the multiplier is register `Rs`. If the `A` bit is set then register `Rn` is added to the product. According to the ARM references, the carry-out flag `C` is assigned a "meaningless value", but in order to execute the HOL specification (see Section 6) it is set to false (though it may just as well be set true).
- The result is stored in register `Rd` and the program-counter is incremented.
- If `Sf` is set then the N, Z and C flags of the CPSR are updated.

This class of instruction is unpredictable when any of the destination or source registers is the program-counter, or when the destination register is the same as the multiplicand register `Rm`.

4.8 Single Data Transfer

This class includes: load instructions, which transfer data from memory to a register; and store instructions, which transfer data from a register to memory. The encoding for this class of instruction is shown in Figure 16. The operations used to specify this class of instruction are listed in Table 13.

The function `LDR_STR`, defined in Figure 17, specifies the state of the processor after executing a LDR or STR instruction:

- The carry-out flag `C` is determined by an external decode.
- Two addresses are computed using the function `ADDR_MODE2`: the address `addr` is used for the load/store, and `addr'` is the write-back address, which is used to update the base register `Rn`. If the `P` bit is set (for pre-indexing) then these two addresses will be the same.

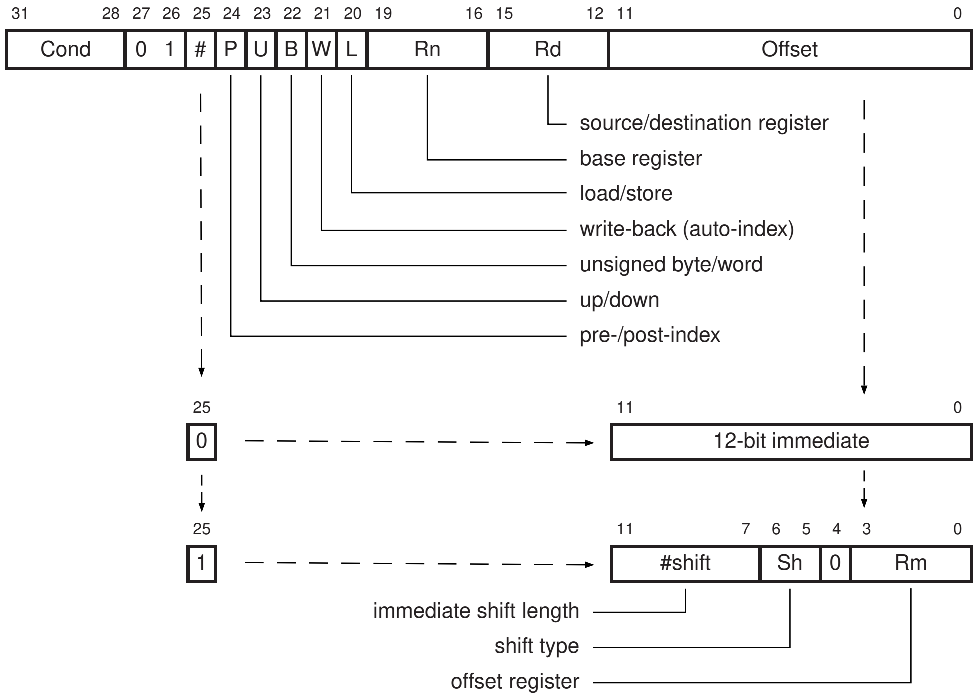

Figure 16: Single data transfer instruction encoding.

```
⊢def LDR_STR (ARM mem reg psr) C mode n =
    let (Im,P,U,B,W,L,Rn,Rd,offset) = DECODE_LDR_STR n in
    if (W ∧ (¬P ∨ (Rn = 15) ∨ (Rn = Rd))) ∨
       (Im ∧ (offset MOD 16 = 15)) ∨ (¬L ∧ (Rd = 15)) then ARB
    else
      let (addr,addr') = ADDR_MODE2 reg mode C Im P U Rn offset in
      let wb_reg = if W ∨ ¬P then REG_WRITE reg mode Rn addr' else reg in
      if L then
        let (data,r_abort) = if B then MEM_READ_BYTE mem addr
                                  else MEM_READ_WORD mem addr in
        if r_abort then
          EXCEPTION (ARM mem (INC_PC (if LATE_ABORT then wb_reg else reg)) psr) 5
        else let reg' = REG_WRITE mode wb_reg Rd data in
          ARM mem (if Rd = 15 then reg' else INC_PC reg') psr
      else
        let rd = REG_READ reg mode Rd in
        let (mem',w_abort) = if B then MEM_WRITE_BYTE mem rd addr
                                  else MEM_WRITE_WORD mem rd addr in
        if w_abort then
          EXCEPTION (ARM mem (INC_PC (if LATE_ABORT then wb_reg else reg)) psr) 5
        else
          ARM mem' (INC_PC wb_reg) psr
```

Figure 17: Single data transfer instruction execution.

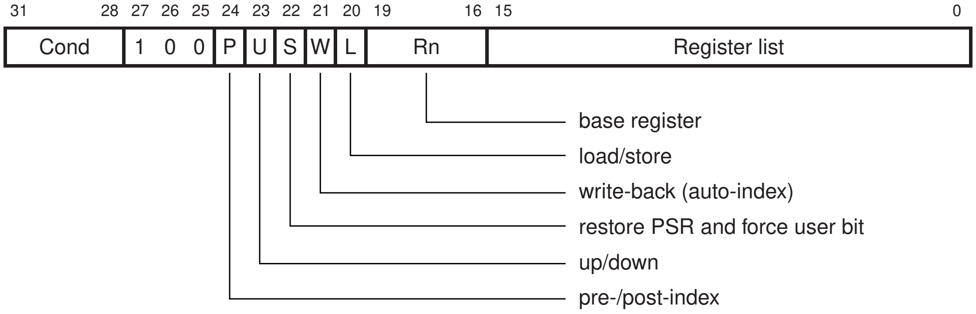

Figure 18: Block data transfer instruction encoding.

- Within `ADDR_MODE2` a base address b is obtained from register `Rn` and an offset δ is computed from the `offset` field, either as a 12-bit immediate value (when `Im` is set), or using the function `SHIFT_IMMEDIATE` from Section 4.5. If `U` is set (for up) then the write-back address is $b + \delta$, otherwise it is $b - \delta$.

- If the `W` bit is set (for write-back) or if `P` is not set (for post-indexing) then `Rn` is assigned the value of the write-back address `addr'`.

- If it is LDR instruction (`L` is set) then either a byte (`B` set) or word is read from the address `addr`. If a data abort does not occur then this data is stored in register `Rd`. If the destination register is not `r15` then the program-counter is incremented. If there was a data abort (`r_abort` set) then an exception of type five is raised. In such cases the constant `LATE_ABORT` (see Section 3.2) dictates whether or not the register `Rn` is updated with the write-back address. Term-rewriting may be used to eliminate this constant from the specification.

- If it is a STR instruction (`L` is not set) then register `Rd` is read, and its value is then stored at address `addr`, either as a byte (`B` set) or as a word. The updated memory is denoted by `mem'`. If there is data abort then an exception of type five is raised. In each case the program-counter is incremented.

This class of instruction is unpredictable when using the program-counter as the offset, when storing the program-counter, and when the write-back address `Rn` is the same as the source/destination register `Rd`.

4.9 Block Data Transfer

This class of instruction is used to store and load blocks of registers to and from memory. Four different types of stack structure are supported, there are modes to force user bank transfer, LDM instructions can be used to effect mode changes and memory aborts can occur during transfers; consequently this is the most complex class of ARM instruction to specify. The encoding for this class of instruction is shown in Figure 18. The operations used to specify this class of instruction are listed in Table 13. The function `LDM_STM`, defined in Figure 19, specifies the state after executing an LDM or STM instruction:

- The variable `list` represents the list of registers that are to be processed. The functionality of an LDM instruction is influenced by whether or not the program-counter is in this list.

- The base address is read from register `Rn`.

Operation	Type	Meaning / Use
LDM_STM	state_ARM→mode→num→state_ARM	Gives the state after executing a LDM or STM instruction
DECODE_LDM_STM	num→bool#bool#bool#bool#bool#num#num list	Decodes LDR and STR instructions
LDM_LIST	(w30→w32)→reg→mode→bool→bool→w32→num list→ reg#bool#w32	Gives the register state, abort status and write-back address for an LDM instruction
STM_LIST	(w30→w32)→reg→mode→bool→bool→w32→num list→ (w30→w32)#bool#w32	Gives the memory state, abort status and write-back address for an STM instruction

Table 14: Operations used to specify block data transfer instructions.

```
⊢def LDM_STM (ARM mem reg psr) mode n =
     let (P,U,Sf,W,L,Rn,list) = DECODE_LDM_STM n in
     let   pc_in_list = BIT 15 n
     and base_in_list = BIT Rn n in
     if (LENGTH list = 0) ∨ (Rn = 15) ∨ (Sf ∧ ((mode = usr) ∨ (mode = sys))) ∨
        (W ∧ ¬L ∧ (base_in_list ∨ Sf)) then ARB
     else
       let rn = REG_READ reg mode Rn in
       if L then
         let mode' = if Sf ∧ ¬pc_in_list then usr else mode in
         let (reg',r_abort,rn') = LDM_LIST mem reg mode' P U rn list in
         let wb_reg = if W ∧ (¬base_in_list ∨ r_abort)
                        then REG_WRITE reg' mode Rn rn'
                        else reg' in
         let pc_reg = if pc_in_list ∧ ¬r_abort then wb_reg else INC_PC wb_reg in
         if r_abort then
           EXCEPTION (ARM mem pc_reg psr) 5
         else
           ARM mem pc_reg
               (if Sf ∧ pc_in_list then CPSR_WRITE psr (SPSR_READ psr mode) else psr)
       else
         let mode' = if Sf then usr else mode in
         let (mem',w_abort,rn') = STM_LIST mem reg mode' P U rn list in
         let reg' = INC_PC (if W then REG_WRITE reg mode Rn rn' else reg) in
         if w_abort then
           EXCEPTION (ARM mem reg' psr) 5
         else
           ARM mem' reg' psr
```

Figure 19: Block data transfer instruction execution.

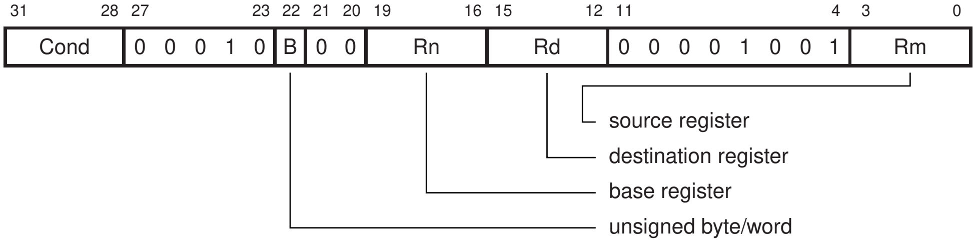

Figure 20: Swap instruction encoding.

- The mode used to access registers, denoted `mode'`, is determined by the `Sf` flag and, in the case of LDM instructions, by the presence or absence of the program-counter in the register list.
- The functions `LDM_LIST` and `STM_LIST` are used to compute the new register/memory state, abort status and the address for register write-back. The type of stack implemented by these operations is determined by the two flags `P` and `U`.
- Register write-back only occurs when the `W` bit is set. In the case of the LDM instruction, if the base register is in the list then the load takes precedence over the write-back, except when a data abort occurs, in which case the write-back occurs in order to facilitate recovery.
- The program-counter is incremented in each case, with the exception of an LDM instruction in which the program-counter in the list and when there is not a data abort.
- If a data abort occurs then an exception of type five is raised.
- If the program-counter is in the list and the `Sf` bit is set then an LDM instruction causes a mode change to occur (the SPSR is restored).

This class of instruction is unpredictable: when the length of the register list is zero; when the base register is `r15`; when changing mode in the user or system modes; and with write-back for the STM instruction when the base register is in the list or when the `Sf` bit set.

4.10 Single Data Swap

The swap instruction provides a way to exchange data between a register and memory in one cycle and without interruption, and this provides a mechanism to implement software semaphores. The encoding of this class of instruction is shown in Figure 20. The function `SWP`, defined in Figure 21, specifies the state of the processor after executing a SWP instruction:

- The `B` flag determines whether a byte or word is swapped.
- The memory address is read from the base register `Rn`.
- Register `Rm` is stored in memory and the old memory value is stored in register `Rd`.
- A data abort raises an exception of type five.
- The program-counter is always incremented.

This class of instruction is unpredictable if the any of the three register is `r15`, or if the base register is the same as the source or destination register. The source and destination registers may be the same.

```
⊢def SWP (ARM mem reg psr) mode n =
    let (B,Rn,Rd,Rm) = DECODE_SWP n in
    if (Rn = 15) ∨ (Rd = 15) ∨ (Rm = 15) ∨ (Rn = Rm) ∨ (Rn = Rd) then ARB
    else
      let (MEM_READ,MEM_WRITE) = if B then (MEM_READ_BYTE,MEM_WRITE_BYTE)
                                      else (MEM_READ_WORD,MEM_WRITE_WORD) in
      let rn = REG_READ reg mode Rn
      and rm = REG_READ reg mode Rm in
      let (word,r_abort) = MEM_READ mem rn
      and (mem',w_abort) = MEM_WRITE mem rm rn in
        if r_abort ∨ w_abort then
          EXCEPTION (ARM mem (INC_PC reg) psr) 5
        else
          ARM mem' (INC_PC (REG_WRITE reg mode Rd word)) psr
```

Figure 21: Swap instruction execution.

31 … 28	27 … 24	23 … 0
Cond	1 1 1 1	24-bit comment field (ignored by processor)

Figure 22: Software interrupt instruction encoding.

4.11 Software Interrupt

The software interrupt instruction is used to enter supervisor mode in a controlled manner. The encoding of this instruction is shown in Figure 22. The function `SWI`, defined in Figure 23, specifies the state after executing a SWI instruction, and this simply corresponds with an exception of type three.

```
⊢def SWI (ARM mem reg psr) mode n = EXCEPTION (ARM mem reg psr) 3
```

Figure 23: Software interrupt instruction execution.

5 State and Next-State Functions

Section 4 gave details of the semantics of the main classes of ARM instruction. This section gives a semantics for the entire instruction set architecture. The ARM architecture is specified by an iterated map state function:

```
⊢def (STATE_ARM 0 a = a) ∧
      STATE_ARM (SUC t) a = NEXT_ARM (STATE_ARM t a)
```

An initialisation function is not defined but one should be aware that in practice the memory and PSR registers need to be suitably initialised, see Section 6. The next-state function is defined in Figure 24:

- An instruction is fetched using the word-aligned program-counter value. If a pre-fetch abort occurs then an exception of type four is raised.
- The CPSR and current instruction are decoded. If the instruction's execute condition is not met then the program-counter is simply incremented, otherwise the instruction is executed by the appropriate function from Sections 4.4 to 4.11.

```
⊢def NEXT_ARM (ARM mem reg psr) =
      let (inst,p_abort) = MEM_READ_WORD mem (WORD_ALIGN (FETCH_PC reg)) in
      if p_abort then
        EXCEPTION (ARM mem reg psr) 4
      else
        let (N,Z,C,V,mode,valid_mode) = DECODE_PSR (CPSR_READ psr) in
        if ¬valid_mode then
          ARB
        else
          let n = W32_NUM inst in
          let (cond,bits2726,bit25,bit24,bit23,bit22,
               bit21,bit20,bits118,bits75,bit4) = DECODE_INST n in
          if ¬(CONDITION_PASSED N Z C V cond) then
            ARM mem (INC_PC reg) psr
          else
            if bits2726 = 0 then
              if bit24 ∧ ¬bit23 ∧ ¬bit20 then
                if ¬bit25 ∧ ¬bit21 ∧ (bits118 = 0) ∧ (bits75 = 4) ∧ bit4 then
                  SWP (ARM mem reg psr) mode n
                else
                  if bit21 then
                    MSR (ARM mem reg psr) mode n
                  else
                    MRS (ARM mem reg psr) mode n
              else
                if ¬bit25 ∧ ¬bit24 ∧ ¬bit23 ∧ ¬bit22 ∧ (bits75 = 4) ∧ bit4 then
                  MUL_MLA (ARM mem reg psr) mode n
                else
                  DATA_PROCESSING (ARM mem reg psr) C V mode n
            else
              if bits2726 = 1 then
                if bit25 ∧ bit4 then
                  EXCEPTION (ARM mem reg psr) 2
                else
                  LDR_STR (ARM mem reg psr) C mode n
              else
                if bits2726 = 2 then
                  if bit25 then
                    BRANCH (ARM mem reg psr) mode n
                  else
                    LDM_STM (ARM mem reg psr) mode n
                else
                  if bit25 ∧ bit24 then
                    SWI (ARM mem reg psr) mode n
                  else
                    EXCEPTION (ARM mem reg psr) 2
```

Figure 24: The ARM next-state function.

- The decoding does not trap all of the holes in the instruction set i.e. it does not assert the precise format for each instruction class. For example, if the bits of an MRS instruction are not set exactly in accordance with Figure 10 from page 21 then an undefined instruction trap ought to be raised, but this is not asserted here. Instead the instruction is executed as if it were an MRS instruction even though, strictly speaking, it is not. The definition, as given, is adequate provided the code is restricted to version 3 of the architecture.

This specification does not consider external control signals (such as reset, IRQ or FIQ) because, at this level of abstraction, attempting to model the interrupt behaviour will severely complicate the specification. The main reason for this is the granularity of the clock, which ensures that each stage of instruction execution is indivisible (unlike those of a pipelined design) and therefore each instruction is considered in isolation. Exceptions on the other hand expose the stepwise and parallel manner in which instructions are executed in microprocessor designs. When an exception occurs in an ARM6 processor:

- It may occur in combination with other exceptions, which either occur at precisely the same time or at slightly different times but all corresponding with the same cycle t with respect to the definition of `STATE_ARM`.
- Three instructions may be effected.
- It is important to know the precise stage of instruction execution (memory access instructions all require a number of cycles to execute).

In other words, just as the program-counter behaviour is tainted with pipelined functionality (see Section 3.4), so is exception/interrupt behaviour, but in this case the abstraction process will adversely effect the readability of the specification. Mathematical tools exist for modelling control signals at different levels of abstraction (by, for example, applying sampling functions to streams of input value) but this topic is beyond the intended scope of this report, see [2].

6 Executing the Specification

One of the advantages of specifying the ARM architecture with a next-state function is that it is possible to execute the specification, either symbolically or using ground (variable free) terms. This means that is possible to test the specification by running small programs. If ground terms are used then the most efficient way to run the specification is by using the HOL library “computeLib”. First it is necessary to define a sensible initial state, for example:

```
ARM MEMORY (REG (λx. W32 0) (λx. W32 0) (λx. W32 0) (λx. W32 0) (λx. W32 0)
      (λx. W32 0)) (PSR (W32 211) (λx. W32 16))
```

where `MEMORY` is a suitably defined memory. The general purpose registers all hold zero, the CPSR register holds 211 (supervisor mode with interrupts disabled), and the SPSRs hold the value sixteen (user mode).

The most effective way to define the memory is to make use of an ARM assembler, such as the portable GNU assembler `as`, available at `www.gnu.org/software/binutils/`. For example, the following trace was generated using the command `as -aln file.s`:

```
 1                    .text
 2                    .align 0
 3
 4 0000 20F0B0E3        movs  r15,#32     @ Reset
 5 0004 0EF0B0E1        movs  r15,r14     @ Undefined Instruction
 6 0008 0EF0B0E1        movs  r15,r14     @ Software Interrupt
 7 000c 04F05EE2        subs  r15,r14,#4  @ Prefetch Abort
 8 0010 08F05EE2        subs  r15,r14,#8  @ Data Abort
 9 0014 0EF0B0E1        movs  r15,r14     @ - Reserved -
10 0018 04F05EE2        subs  r15,r14,#4  @ IRQ
11 001c 04F05EE2        subs  r15,r14,#4  @ FIQ
12 0020 0800A0E3        mov   r0, #8
13 0024 0B10A0E3        mov   r1, #11
14 0028 1C0090E8        ldmia r0, {r2,r3,r4}
```

The source assembly code is shown to the right, with the line numbers shown in the leftmost column. The third column is the encoding of each instruction i.e. ARM machine code, and the second column is the memory address for each instruction. The machine code is presented in a little-endian manner, with each pair of hexadecimal digits representing one byte. For example, the instruction on line 14 is converted into a 32-bit word as follows:

`1C0090E8` ↦ (00011100, 00000000, 10010000, 11101000) ↦ 11101000100100000000000000011100 ↦ `W32 3901751324`

Hence, the memory is a function in which:

```
W30 0  ↦ W32 3820023840
W30 1  ↦ W32 3786469390
W30 2  ↦ W32 3786469390
W30 3  ↦ W32 3797872644
W30 4  ↦ W32 3797872648
W30 5  ↦ W32 3786469390
W30 6  ↦ W32 3797872644
W30 7  ↦ W32 3797872644
W30 8  ↦ W32 3818913800
W30 9  ↦ W32 3818917899
W30 10 ↦ W32 3901751324
```

There are many ways to define such a function in HOL. This program is very small and the efficiency of access is not that critical, therefore the first eleven words can be stored in a list, with instructions accessed using the function `EL:num→'a list→'a`. The first eight lines of code constitute a minimalistic exception handler, which simply ignores the exception and returns to the entry address. In the case of memory aborts and interrupts the instruction is retried (note that with data aborts the program-counter is incremented before the exception is raised, see Section 4.5). The reset exception jumps to the first line of the program. The result of executing this program for four cycles is as follows:

```
⊢ STATE_ARM 4 (ARM MEMORY RESET_REG RESET_PSR) =
       ARM MEM
         (REG
            (Sb
               (Sb
                  (Sb
                     (Sb (Sb (Sa (λx. W32 0) (W4 0,W32 8)) (W4 1,W32 11))
                        (W4 2,W32 3786469390)) (W4 3,W32 3797872644))
                  (W4 4,W32 3797872648)) (W4 15,W32 44)) (λx. W32 0)
            (λx. W32 0) (λx. W32 0) (λx. W32 0) (λx. W32 0))
         (PSR (W32 16) (λx. W32 16))
```

The registers 2, 3, and 4 have been loaded with the contents of the memory from addresses 8, 12, and 16, which is the expected result of executing the `ldmia` instruction. The mode is changed to user mode after the first instruction is executed—exiting from the reset exception handler.

This theorem was generated using `CBV_CONV`, which rewrites terms using a call-by-value strategy, similar to function evaluation in ML. On an 800MHz Athlon PC, each instruction takes about half a second to execute. The performance is strongly dependent upon the definitions of the operations in Table 6 on page 14. For example, the performance is impaired by explicitly using the following axiom:

```
⊢def BITS h l n = (n DIV 2 EXP l) MOD 2 EXP (SUC h-l)‘;
```

Instead the value n is converted to a binary form, where it is possible to write efficient operations for division and remainder by powers of two. HOL is used to prove that the optimised versions of these operations conform with their axiomatic definition.

Both `Sb` and `Sa` are pseudonyms for the function `SUBST`. These two functions are used as a means to produce normal forms for nested memory substitutions. The substitutions are ordered and redundancy, caused by one substitution superseding another, is eliminated. This means that although the program-counter is updated four times, only the resultant value of 44 is presented. This simplification is only appropriate when evaluating ground-terms.

The appendix contains examples of executing a range of instructions.

7 Future Work and Conclusions

This report has presented a HOL specification of the ARM instruction set architecture. The approach taken was to model eight classes of instruction using state transforming functions. Higher-order logic has been shown to be a good framework for modelling the architecture in a concise and unambiguous manner. The logic allows axiomatic definitions to be proved equivalent to optimised versions. Call-by-value term rewriting is used to simulate the execution of small programs, enabling the specification to be demonstrated and tested.

The specification constitutes an abstract model of the architecture, and this can be used as a basis for proving the correctness of microprocessor designs. The next research goal is to model a 3-stage pipelined ARM design and then prove its correctness. Ways must be found to manage the scale of this proof goal—a subset of the instruction set will be tackled first. Ways to incorporate and verify interrupt handling will also be investigated.

8 Acknowledgments

The approach taken in this work has been greatly influenced by research at the University of Wales Swansea on the algebraic specification of microprocessors [7, 4].

This work is supported by EPSRC grant GR/N13135, “Formal Specification and Verification of ARM6”. This project is in collaboration with the University of Leeds School of Computing: Graham Birtwistle, Keith Hobley, Dominic Pajak and Daniel Schostak. Thanks must go to them for their input, and to Mike Gordon for reading a draft of this report.

Appendix

Instruction Execution Examples

Unless otherwise indicated, it is assumed that the registers and the initial eight words of memory are set in accordance with Section 6.

Branching

Code fragment:

```
(W32 32) 0800A0E3        mov   r0, #8
                       label:
(W32 36) 020050E2        subs  r0, r0, #2
(W32 40) FDFFFF1A        bne   label
```

Result:

```
⊢ STATE_ARM 4 (ARM MEMORY RESET_REG RESET_PSR) =
       ARM MEMORY
         (REG (Sb (Sa (λx. W32 0) (W4 0,W32 6)) (W4 15,W32 36)) (λx. W32 0)
            (λx. W32 0) (λx. W32 0) (λx. W32 0) (λx. W32 0))
         (PSR (W32 536870928) (λx. W32 16))
```

- The CPSR carry-out bit is set by the subtraction.
- The program-counter points to the `subs` instruction.

Code fragment:

```
(W32 32) 0800A0E3        mov   r0, #8
                       label:
(W32 36) 020050E2        subs  r0, r0, #2
(W32 40) FDFFFF1B        blne   label
```

Result:

```
⊢ STATE_ARM 4 (ARM MEMORY RESET_REG RESET_PSR) =
       ARM MEMORY
         (REG
            (Sb (Sb (Sa (λx. W32 0) (W4 0,W32 6)) (W4 14,W32 44))
               (W4 15,W32 36)) (λx. W32 0) (λx. W32 0) (λx. W32 0) (λx. W32 0)
            (λx. W32 0)) (PSR (W32 536870928) (λx. W32 16))
```

- The link register has value 44.

Data Processing

Code fragment:

```
(W32 32) AA00E0E3        mvn   r0, #0xaa
(W32 36) BB10E0E3        mvn   r1, #0xbb
(W32 40) CC20E0E3        mvn   r2, #0xcc
(W32 44) DD30E0E3        mvn   r3, #0xdd
(W32 48) 002092E0        adds  r2, r2, r0
(W32 52) 0130B3E0        adcs  r3, r3, r1
```

Result:

```
⊢ STATE_ARM 7 (ARM MEMORY RESET_REG RESET_PSR) =
       ARM MEMORY
         (REG
            (Sb
               (Sb
                  (Sb
                     (Sb (Sa (λx. W32 0) (W4 0,W32 4294967125))
                        (W4 1,W32 4294967108)) (W4 2,W32 4294966920))
                  (W4 3,W32 4294966887)) (W4 15,W32 56)) (λx. W32 0)
            (λx. W32 0) (λx. W32 0) (λx. W32 0) (λx. W32 0))
         (PSR (W32 2684354576) (λx. W32 16))
```

- The words `r1,r0` and `r3,r2` are loaded with 64-bit integer values `0xFFFFFF44FFFFFF55` and `0xFFFFFF22FFFFFF33` respectively.
- The 65-bit sum `0x1FFFFFE67FFFFFE88` is computed with the 64-bit result stored in `r3,r2` and the 65th bit forming the carry-out. The result is negative, so the N bit of the CPSR is also set.

Code fragment:

```
(W32 32) 0C00A0E3        mov   r0, #12
(W32 36) 000180E0        add   r0, r0, r0, LSL #2
```

Result:

```
⊢ STATE_ARM 3 (ARM MEMORY RESET_REG RESET_PSR) =
       ARM MEMORY
         (REG (Sb (Sa (λx. W32 0) (W4 0,W32 60)) (W4 15,W32 40)) (λx. W32 0)
            (λx. W32 0) (λx. W32 0) (λx. W32 0) (λx. W32 0))
         (PSR (W32 16) (λx. W32 16))
```

- The register `r0` is loaded with 12, and this then multiplied by five by using an add with shift left.

Code fragment:

```
(W32 32) 0C00A0E3        mov   r0, #12
(W32 36) 0A10A0E3        mov   r1, #10
(W32 40) 012080E1        orr   r2, r0, r1
(W32 44) 013000E0        and   r3, r0, r1
(W32 48) 014020E0        eor   r4, r0, r1
(W32 52) 0150C0E1        bic   r5, r0, r1
```

Result:

```
⊢ STATE_ARM 7 (ARM MEMORY RESET_REG RESET_PSR) =
       ARM MEMORY
         (REG
            (Sb
               (Sb
                  (Sb
                     (Sb
                        (Sb (Sb (Sa (λx. W32 0) (W4 0,W32 12)) (W4 1,W32 10))
                           (W4 2,W32 14)) (W4 3,W32 8)) (W4 4,W32 6))
                  (W4 5,W32 4)) (W4 15,W32 56)) (λx. W32 0) (λx. W32 0)
            (λx. W32 0) (λx. W32 0) (λx. W32 0)) (PSR (W32 16) (λx. W32 16))
```

- The bitwise operations are tested with the values 1100_2 and 1010_2.

PSR Transfer

Code fragment:

```
(w32  0) 20F0A0E3        mov   r15,#32     @ Reset
            :  :          :
(w32 28) 04F05EE2        subs  r15,r14,#4  @ FIQ
(w32 32) 0F02A0E3        mov   r0, #0xf0000000
(w32 36) 120080E3        orr   r0, r0, #0x12
(w32 40) 00F029E1        msr   CPSR, r0
```

Result:

```
⊢ STATE_ARM 4 (ARM MEMORY RESET_REG RESET_PSR) =
     ARM MEMORY
       (REG (Sb (Sa (λx. W32 0) (W4 0,W32 4026531858)) (W4 15,W32 44))
          (λx. W32 0) (λx. W32 0) (λx. W32 0) (λx. W32 0) (λx. W32 0))
       (PSR (W32 4026531858) (λx. W32 16))
```

- The `mov` instruction (at address zero) replaces the usual `movs` instruction—this means that the code is executed in supervisor mode.
- The resultant CPSR value is $4026531858 = \mathrm{F0000012}_{16}$. All the flags are set and IRQ mode is selected.

Code fragment:

```
(w32  0) 20F0A0E3        mov   r15,#32     @ Reset
            :  :          :
(w32 28) 04F05EE2        subs  r15,r14,#4  @ FIQ
(w32 32) ED00E0E3        mov   r0, #0xffffff12
(w32 36) 00F021E1        msr   CPSR_c, r0
(w32 40) 00100FE1        mrs   r1, CPSR
(w32 44) 00204FE1        mrs   r2, SPSR
(w32 48) 0EF268E3        msr   SPSR_f, #0xe0000000
```

Result:

```
⊢ STATE_ARM 6 (ARM MEMORY RESET_REG RESET_PSR) =
     ARM MEMORY
       (REG
          (Sb
             (Sb (Sb (Sa (λx. W32 0) (W4 0,W32 4294967058)) (W4 1,W32 18))
                (W4 2,W32 16)) (W4 15,W32 52)) (λx. W32 0) (λx. W32 0)
          (λx. W32 0) (λx. W32 0) (λx. W32 0))
       (PSR (W32 18) (Sa (λx. W32 16) (spsr_irq,W32 3758096400)))
```

- The `mov` command at address 32 is converted, by the assembler, into a `mvn` command.
- Register `r1` takes the CPSR value (which is changed to IRQ mode).
- Register `r2` takes the SPSR value for the IRQ mode.
- The SPSR flags are then set.

Multiply and Multiply-Accumulate

Code fragment:

```
(W32 32) 0A00A0E3        mov   r0, #10
(W32 36) 1410A0E3        mov   r1, #20
(W32 40) 1E20A0E3        mov   r2, #30
(W32 44) 900103E0        mul   r3, r0, r1
(W32 48) 902124E0        mla   r4, r0, r1, r2
```

Result:

```
⊢ STATE_ARM 6 (ARM MEMORY RESET_REG RESET_PSR) =
       ARM MEMORY
         (REG
            (Sb
               (Sb
                  (Sb
                     (Sb (Sb (Sa (λx. W32 0) (W4 0,W32 10)) (W4 1,W32 20))
                        (W4 2,W32 30)) (W4 3,W32 200)) (W4 4,W32 230))
               (W4 15,W32 52)) (λx. W32 0) (λx. W32 0) (λx. W32 0) (λx. W32 0)
            (λx. W32 0)) (PSR (W32 16) (λx. W32 16))
```

- The product is 230, and when summed with the accumulator `r2` this gives 230.

Single Data Transfer

Code fragment:

```
(W32 32) 0800A0E3        mov   r0, #8
(W32 36) 4010A0E3        mov   r1, #64
(W32 40) A121B0E7        ldr   r2, [r0, r1, LSR #3]!
(W32 44) 0130D0E4        ldrb  r3, [r0], #1
(W32 48) 0140D0E4        ldrb  r4, [r0], #1
(W32 52) 0150D0E4        ldrb  r5, [r0], #1
(W32 56) 0060D0E5        ldrb  r6, [r0]
```

Result:

```
⊢ STATE_ARM 8 (ARM MEMORY RESET_REG RESET_PSR) =
       ARM MEMORY
         (REG
            (Sb
               (Sb
                  (Sb
                     (Sb
                        (Sb
                           (Sb
                              (Sb (Sa (λx. W32 0) (W4 0,W32 19)) (W4 1,W32 64))
                              (W4 2,W32 3797872648)) (W4 3,W32 8))
                        (W4 4,W32 240)) (W4 5,W32 94)) (W4 6,W32 226))
               (W4 15,W32 60)) (λx. W32 0) (λx. W32 0) (λx. W32 0) (λx. W32 0)
            (λx. W32 0)) (PSR (W32 16) (λx. W32 16))
```

- The base address is 8 and the offset is 8, therefore the load and write-back address is 16 (which is the exception vector for the data abort).
- Post-indexing is used to load the same four bytes, starting from address 16. The bytes are read using little-endian byte ordering. If big-endian ordering were used then the contents of registers three to six would be reversed.

Code fragment:

```
(W32 32) 0400A0E3        mov   r0, #4
(W32 36) 5010A0E3        mov   r1, #80
(W32 40) CD20A0E3        mov   r2, #0xcd
(W32 44) AB2C82E3        orr   r2, r2, #0xab00
(W32 48) 002581E6        str   r2, [r1], r0, LSL #10
(W32 52) 042001E4        str   r2, [r1], #-4
(W32 56) 042041E5        strb  r2, [r1, #-4]
```

Result:

```
⊢ STATE_ARM 8 (ARM MEMORY RESET_REG RESET_PSR) =
       ARM
         (Sb (Sb (Sa MEMORY (W30 20,W32 43981)) (W30 1042,W32 205))
            (W30 1044,W32 43981))
         (REG
            (Sb
               (Sb (Sb (Sa (λx. W32 0) (W4 0,W32 4)) (W4 1,W32 4172))
                  (W4 2,W32 43981)) (W4 15,W32 60)) (λx. W32 0) (λx. W32 0)
            (λx. W32 0) (λx. W32 0) (λx. W32 0)) (PSR (W32 16) (λx. W32 16))
```

- This example shows pre- and post-indexing with an upward offset.
- The byte is stored using little-endian byte-ordering. If big-endian ordering were used then address 1042 would contain the value 3439329280.

Block Data Transfer

Code fragment:

```
(W32  0) 20F0A0E3        mov   r15,#32     @ Reset
           :  :           :
(W32 28) 04F05EE2        subs  r15,r14,#4  @ FIQ
(W32 32) 1100A0E3        mov   r0, #0x11
(W32 36) 00F021E1        msr   CPSR_c, r0
(W32 40) FCFFF1E8        ldmia r1!, {r2-r15}^
```

Result:

```
⊢ STATE_ARM 4 (ARM MEMORY RESET_REG RESET_PSR) =
       ARM MEMORY
         (REG
            (Sb
               (Sb
                  (Sb
                     (Sb
                        (Sb
                           (Sb
                              (Sb
                                 (Sb (Sa (λx. W32 0) (W4 0,W32 17))
                                    (W4 1,W32 56)) (W4 2,W32 3818975264))
                              (W4 3,W32 3786469390)) (W4 4,W32 3786469390))
                        (W4 5,W32 3797872644)) (W4 6,W32 3797872648))
                  (W4 7,W32 3786469390)) (W4 15,W32 0))
            (Sb
               (Sb
                  (Sb
                     (Sb
                        (Sb
                           (Sb (Sa (λx. W32 0) (r8_fiq,W32 3797872644))
                              (r9_fiq,W32 3797872644))
                           (r10_fiq,W32 3818913809)) (r11_fiq,W32 3777097728))
                     (r12_fiq,W32 3908173820)) (r13_fiq,W32 0))
               (r14_fiq,W32 0)) (λx. W32 0) (λx. W32 0) (λx. W32 0)
            (λx. W32 0)) (PSR (W32 16) (λx. W32 16))
```

- The mode is changed to `FIQ`, and then the exception handler is loaded into the registers for this mode.
- The program-counter is in the register list and, becaause the `Sf` bit is set (achieved by suffixing the command with a ^), a change to user mode occurs after the load.
- This toy program will eventually lead to unpredictable behaviour because on cycle eight the `ldm` instruction is encountered again but this time in user mode, where the mode change is not possible.

Code fragment:

```
(W32  0) 20F0A0E3        mov   r15,#32     @ Reset
           :  :           :
(W32 28) 04F05EE2        subs  r15,r14,#4  @ FIQ
(W32 32) 1100A0E3        mov   r0, #0x11
(W32 36) 00F021E1        msr   CPSR_c, r0
(W32 40) C07FF1E8        ldmia r1!, {r6-r14}^
```

Result:

```
⊢ STATE_ARM 4 (ARM MEMORY RESET_REG RESET_PSR) =
       ARM MEMORY
         (REG
            (Sb
               (Sb
                  (Sb
                     (Sb
                        (Sb
                           (Sb
                              (Sb
                                 (Sb
                                    (Sb
                                       (Sb
                                          (Sb (Sa (λx. W32 0) (W4 0,W32 17))
                                             (W4 1,W32 36))
                                          (W4 6,W32 3818975264))
                                       (W4 7,W32 3786469390))
                                    (W4 8,W32 3786469390))
                                 (W4 9,W32 3797872644)) (W4 10,W32 3797872648))
                           (W4 11,W32 3786469390)) (W4 12,W32 3797872644))
                     (W4 13,W32 3797872644)) (W4 14,W32 3818913809))
               (W4 15,W32 44)) (λx. W32 0) (λx. W32 0) (λx. W32 0) (λx. W32 0)
            (λx. W32 0)) (PSR (W32 17) (λx. W32 16))
```

- This is the same program as before, but with a different register list. The program-counter is not in the list.
- This time the mode change does not occur but a user bank transfer is forced.

Code fragment:

```
(W32 32) 0800A0E3        mov   r0, #8
(W32 36) 7F00B0E8        ldmia r0!, {r0-r6}
(W32 40) 7F0021E9        stmdb r1!, {r0-r6}
```

Result:

```
⊢ STATE_ARM 4 (ARM MEMORY RESET_REG RESET_PSR) =
       ARM
         (Sb
            (Sb
               (Sb
                  (Sb
                     (Sb
                        (Sb (Sa MEMORY (W30 949468154,W32 3818913800))
                           (W30 949468155,W32 3797872644))
                        (W30 949468156,W32 3797872644))
                     (W30 949468157,W32 3786469390))
                  (W30 949468158,W32 3797872648))
               (W30 949468159,W32 3797872644)) (W30 949468160,W32 3786469390))
         (REG
            (Sb
               (Sb
                  (Sb
                     (Sb
                        (Sb
                           (Sb
                              (Sb (Sa (λx. W32 0) (W4 0,W32 3786469390))
                                 (W4 1,W32 3797872612)) (W4 2,W32 3797872648))
                           (W4 3,W32 3786469390)) (W4 4,W32 3797872644))
                     (W4 5,W32 3797872644)) (W4 6,W32 3818913800))
               (W4 15,W32 44)) (λx. W32 0) (λx. W32 0) (λx. W32 0) (λx. W32 0)
            (λx. W32 0)) (PSR (W32 16) (λx. W32 16))
```

- The `ldm` register write-back is overwritten.

Single Data Swap

Code fragment:

```
(W32 32) 0800A0E3        mov   r0, #8
(W32 36) 1010A0E3        mov   r1, #16
(W32 40) 1820A0E3        mov   r2, #24
(W32 44) 910002E1        swp   r0, r1, [r2]
```

Result:

```
⊢ STATE_ARM 5 (ARM MEMORY RESET_REG RESET_PSR) =
       ARM (Sa MEMORY (W30 6,W32 16))
         (REG
            (Sb
               (Sb (Sb (Sa (λx. W32 0) (W4 0,W32 3797872644)) (W4 1,W32 16))
                  (W4 2,W32 24)) (W4 15,W32 48)) (λx. W32 0) (λx. W32 0)
            (λx. W32 0) (λx. W32 0) (λx. W32 0)) (PSR (W32 16) (λx. W32 16))
```

- Register zero is overwritten with the memory contents from address 24. At the same time register one is stored at this address.
- In practice this is not possible because the exception handler will be protected.

Code fragment:

```
(W32 32) 1100A0E3        mov   r0, #0x11
(W32 36) 110C80E3        orr   r0, r0, #0x1100
(W32 40) 1810A0E3        mov   r1, #24
(W32 44) 900041E1        swpb  r0, r0, [r1]
```

Result:

```
⊢ STATE_ARM 5 (ARM MEMORY RESET_REG RESET_PSR) =
       ARM (Sa MEMORY (W30 6,W32 3797872657))
          (REG
             (Sb (Sb (Sa (λx. W32 0) (W4 0,W32 4)) (W4 1,W32 24))
                (W4 15,W32 48)) (λx. W32 0) (λx. W32 0) (λx. W32 0) (λx. W32 0)
             (λx. W32 0)) (PSR (W32 16) (λx. W32 16))
```

- The first byte of the word at address six is altered.
- The old byte value is stored (zero extended) in register zero.

Software Interrupt

Code fragment:

```
(W32 32) 000000EF        swi
```

Result:

```
⊢ STATE_ARM 2 (ARM MEMORY RESET_REG RESET_PSR) =
       ARM MEMORY
         (REG (Sa (λx. W32 0) (W4 15,W32 8)) (λx. W32 0) (λx. W32 0)
            (Sa (λx. W32 0) (r14_svc,W32 36)) (λx. W32 0) (λx. W32 0))
         (PSR (W32 147) (Sa (λx. W32 16) (spsr_svc,W32 16)))
```

- Supervisor mode is entered and IRQs are disabled.

Code fragment:

```
(W32 32) 000000EF        swi
(W32 36) 0A00E0E2        rsc   r0, r0, #10
```

Result:

```
⊢ STATE_ARM 4 (ARM MEMORY RESET_REG RESET_PSR) =
       ARM MEMORY
         (REG (Sb (Sa (λx. W32 0) (W4 0,W32 4294967285)) (W4 15,W32 40))
            (λx. W32 0) (λx. W32 0) (Sa (λx. W32 0) (r14_svc,W32 36))
            (λx. W32 0) (λx. W32 0))
         (PSR (W32 16) (Sa (λx. W32 16) (spsr_svc,W32 16)))
```

- The exception handler simply returns to the next instruction.

References

[1] Alonzo Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940.

[2] Anthony C. J. Fox. *Algebraic Models for Advanced Microprocessors*. PhD thesis, University of Wales Swansea, 1998.

[3] Anthony C. J. Fox. An algebraic framework for modelling and verifying microprocessors using HOL. Technical Report 512, University of Cambridge, Computer Laboratory, April 2001.

[4] Anthony C. J. Fox and Neal A. Harman. Algebraic models of correctness for microprocessors. *Formal Aspects of Computing*, 12(4):298–312, 2000.

[5] Steve Furber. *ARM: system-on-chip architecture*. Addison-Wesley, second edition, 2000.

[6] Mike J. C. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh* LCF*: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[7] Neal A. Harman and John V. Tucker. Algebraic models of microprocessors: Architecture and organisation. *Acta Informatica*, 33(5):421–456, 1996.

[8] Dominic Pajak. ARM6 instruction set architecture specification. Technical report, University of Leeds, School of Computing, 2001. In preparation.

[9] Larry Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*, volume Cambridge Tracts in Theoretical Computer Science 2. Cambridge University Press, 1987.

[10] David Seal, editor. *ARM Architectural Reference Manual.* Addison-Wesley, second edition, 2000.