Formal Verification of the ARM6 Micro-architecture

Anthony Fox
Computer Laboratory, University of Cambridge

November 11, 2002

Abstract

This report describes the formal verification of the ARM6 micro-architecture using the HOL theorem prover. The correctness of the microprocessor design compares the micro-architecture with an abstract, target instruction set semantics. Data and temporal abstraction maps are used to formally relate the state spaces and to capture the timing behaviour of the processor. The verification is carried out in HOL and *one-step* theorems are used to provide the framework for the proof of correctness. This report also describes the formal specification of the ARM6's three stage pipelined micro-architecture.

Contents

1 Introduction

This report describes the formal verification of the ARM6 micro-architecture in HOL. This work forms part of a collaborative EPSRC funded project on the formal specification and verification of the ARM6. The project has been carried out with the support of ARM Ltd. Graham Birtwistle's group at Leeds have produced detailed ML specifications of the ARM architecture (Dominic Pajak) and the ARM6 micro-architecture (Daniel Schostak). At Cambridge, formal verification work has been undertaken using the HOL system. The Leeds specifications have been used as the basis for the production of compact and executable HOL models. The ARM instruction set architecture model is presented in [13]. Some simplifications have been made (with respect to the actual ARM6 and Daniel's high-fidelity specification) for the purposes of the initial verification attempt described here. Nevertheless, the verification is substantial and there is scope for carrying out additional more complete verifications in future. The approach used for the formal verification is based on work done at Swansea [17, 16, 14, 11], which has been formalised in HOL at Cambridge [12]. This approach provides a general and structured framework for carrying out processor verifications. However, before now only small scale (toy) case studies have been considered. This project aimed to apply these methods to a commercial processor design and, in doing so, assess the suitability of HOL for this task.

By developing executable models, it has been possible to test the HOL specifications with machine code generated by a GNU assembler. This has allowed sanity checks and extensive debugging to occur before carrying out the formal verification. The HOL specification of the ARM6 micro-architecture is presented in this report. However, it is *not* the intention of this report to give a fully extensive account of the ARM instruction set or the ARM6 micro-architecture. Rather, the functionality of the ARM6 is discussed to give a feel for the nature of the design and the correctness issues that arise as a consequence. The HOL specification of the ARM6 is presented in the appendix, primarily to show how the design is modelled in HOL and to illustrate the complexity of the design. Little attempt is made to explain the intricacies of the definitions therein. Two slightly different versions of the design (no-clobber and data forwarding) are presented and these represent two approaches to solving the problem of code invalidating the pipeline state (self-modifying code).

Section 2 provides a brief overview of processor verification work that was undertaken in the 1980s and 1990s. The algebraic approach to processor verification, which was developed at Swansea and used for the ARM6 verification, is presented in Section 3. This section provides an abstract definition of correctness and introduces the one-step theorems, which are used as the basis for the verification. An overview of the ARM instruction set is given in Section 4. The aim of this section is to highlight that ARM is a rich RISC architecture designed for writing compact, general purpose software for embedded systems: it is not a 'toy' architecture. The HOL specification of the architecture is documented in [13]. Modifications have been made to this specification, both in the course of the verification and as subsequent rationalizations; these changes are outlined in Section 5. The ARM6 micro-architecture is presented in Section 6: the data path is discussed in Section 6.1 and the pipeline control is described in Section 6.2. The formal verification is then discussed in Section 7. The HOL specification of the micro-architecture is given in the appendix.

2 Related Work

Several verifications of complete processors were undertaken during the 1980s and early 1990s. Examples include TAMARACK [24], SECD [3, 15], the partial verification of Viper [7, 8, 9], and Hunt's FM8501 and FM9001 [21, 22, 23] and Windley [36, 38]. All these processors were simple uniprocessor fetch-decode-execute engines specifically designed for formal verification. Following this work, Miller and Srivas verified the implementation of some of the instructions of a simple real processor called AAMP5 [27, 28]. A simplified version of the ARM2 processor was verified in [20].

Processors became much more complex from the later 1980s due to the addition of such features as complex multi-stage pipelines, out-of-order execution and co-processors. Architectures like Alpha, MIPS, Pentium, and PowerPC were considered too complex for complete formal verification and the introduction of parallelism made their specifications that much harder. As a result, the main thrust of hardware verification research work has shifted from one-off proofs of whole processors to focusing on the analysis of fragments (such as buses, caches and pipelines) using a variety of techniques including theorem proving, temporal logic model checking and process algebra.

Various models of correctness have been applied to pipelined designs and a framework for categorizing these has been proposed in [1]. Burch and Dill [6] developed a pipeline flushing abstraction to show the equivalence between an instruction set architecture and a pipelined implementation. The ideas have been further developed by Burch, [5], Windley and Burch, [37], and Skakkebaek *et al.* [33], for pipelined microprocessors. Further developments to an out-of-order processor core have been made by Damm and Pnueli [10] and McMillan [26]. Sawada and Hunt [31] give a formal proof of a pipeline with exceptions which requires invariants between successive pipeline stages. Aagaard and Leeser [2] developed the transaction technique (mirroring how an instruction is decoded and executed down the pipeline stages) method to cater for pipelines with hazards which has been taken up by Launchbury's group at Oregon. The verification of Hennessey and Patterson's widely used pedagogical DLX RISC processor has been studied, using HOL, by Tahar and Kumar [34, 30, 35]. All the above used mechanized proof tools.

3 The Formal Verification of Processor Designs

This section outlines the approach used in the formal verification of the ARM6. A detailed account of this *algebraic framework* can be found in [12].

3.1 Approach

This section formalises, in an abstract setting, a definition of *correctness*. This definition can be applied to the formal verification of pipelined microprocessor designs (such as the ARM6). The approach is based on comparing two models:

1. The processor's micro-architecture (MA); and
2. The processor's instruction set architecture (ISA).

These two models occupy different levels of data and temporal abstraction, and this is made apparent when one formalises *state* and *time*. Correctness requires there to be a correspondence in behaviour between these two models. The MA and ISA are modelled using *state functions*; these are maps of the form $f : time \to state \to state$. The set $time$ is required to be countable, and consequently it is simply assumed that $time = \mathbb{N}$. The set of states $state$ is called the *state space*, and this is a non-empty set. The function f formally specifies the required behaviour of a design at some level of temporal and data abstraction: the state at time t, from the *pre-initial*[1] state a, is $f\ t\ a$. At the ISA level, the state will contain only entities visible to the programmer (typically, memory and registers) and each state transition marks the execution of a single instruction. At the MA level, the state contains components from the implementation (for example, the pipeline state) and each state transition normally marks one processor clock cycle. Correctness requires there to be a correspondence between sequences of ISA and MA states.

Correctness is formalised in HOL with the following definition:

```
⊢def CORRECT (spec:num→α→α) (impl:num→β→β) (imm:β→num→num) (abs:β→α) =
      IMMERSION imm ∧
      DATA_ABSTRACTION abs (spec 0) (impl 0) ∧
      ∀t (a:β). spec t (abs a) = abs (impl (imm a t) a)
```

---

[1] The initialisation function $f\ 0$ maps arbitrary, pre-initialised states to valid initial states.

The state function *impl* is the *implementation* (modelling the MA) and *spec* is the *specification* (modelling the ISA). The state spaces are related using a data abstraction `abs`. Temporal abstraction relates the *clocks* of the two systems, and this is the rôle of the *immersion* `imm`. Note that the temporal abstraction is a function of the pre-initialised state of the implementation. The `DATA_ABSTRACTION` and `IMMERSION` conditions are captured using the following definitions:

```
⊢def DATA_ABSTRACTION abs init init’ = ∀a. ∃b. abs (init’ b) = init a
⊢def IMMERSION imm = (∀a. imm a 0 = 0) ∧ ∀a t1 t2. t1 < t2 ⇒ imm a t1 < imm a t2
```

The data abstraction criterion ensures that the implementation is complete with respect to the specification i.e. each initial state of the specification is an abstraction of an initial implementation state. An immersion is a monotonic increasing function from time, at the specification level, to time at the implementation level. The immersion is parameterized by the pre-initialised state of the implementation. The state functions *impl* and *spec* are required to be related with the condition:

$$\forall t\ a.\ \ spec\, t\ (abs\, a) = abs(impl\ (imm\, a\, t)\ a)$$

This condition ensures that after executing t instruction at the specification level, the processor state at time $imm\, a\ t$ is equivalent to the specification state, modulo applications of the data abstraction abs.

Correctness is defined with respect to the maps abs and imm, but it is possible for correctness to hold with respect to different abstractions. A less prescriptive presentation of correctness is:

```
⊢def IS_CORRECT spec impl = ∃imm abs. CORRECT spec impl imm abs
```

The verification of correctness then involves selecting suitable witnesses for the maps imm and abs.

The correctness framework described above is that used in the verification of the ARM6. This framework can be further enriched, for example, to accommodate input and output, and to deal with the temporal characteristics of superscalar processors. For a fuller account the reader is referred to [11, 12].

3.2 One-Step Theorems

This section describes an approach to the verification of correctness. This involves restricting the way in which state functions and immersions are defined. The following *one-step* theorem is used:

ONE_STEP_THM

```
⊢ ∀spec impl imm abs.
       UNIFORM imm impl ∧ TCON spec ∧ TCON_IMMERSION impl imm ⇒
         (CORRECT spec impl imm abs =
          DATA_ABSTRACTION abs (spec 0) (impl 0) ∧
          (∀a. spec 0 (abs a) = abs (impl (imm a 0) a)) ∧
           ∀a. spec 1 (abs a) = abs (impl (imm a 1) a))
```

This theorem reduces the verification of correctness to a goal in which t has been specialised to the times 0 and 1. This avoids the need to carry out an explicit induction over time when verifying the main correctness goal. The conditions on the specification, implementation and immersion require them to be deterministic. An immersion is *uniform* with respect to a state function if, and only if, each interval $imm\, a\ (t+1) - imm\, a\ t$ is a function of the implementation's state at time $imm\, a\ t$. This is expressed with the following definitions:

```
⊢def UNIFORM imm f = ∃dur. UIMMERSION imm f dur
⊢def UIMMERSION imm f dur =
        (∀a. 0 < dur a) ∧ (∀a. imm a 0 = 0) ∧
        ∀a t. imm a (SUC t) = dur (f (imm a t) a) + imm a t
```

The *time-consistency* property is defined by:

```
⊢def TCON f = ∀t1 t2 a. f (t1 + t2) a = f t1 (f t2 a)
```

Time-consistent state functions have primitive recursive definitions i.e. they can be modelled with *initialisation* and *next state* functions. Furthermore, a state function f is time-consistent if, and only if, the initialisation function $(f\ 0)$ is an identity map on the range of f. This invariance property forms the basis of the one-step theorem. Time-consistency is also defined with respect to an immersion:

```
⊢def TCON_IMMERSION f imm =
        ∀t1 t2 a.
          f (imm (f (imm a t2) a) t1 + imm a t2) a =
          f (imm (f (imm a t2) a) t1) (f (imm a t2) a)
```

Here initialisation is only required to be an identity map at times given by the immersion.

It is often the case that $\mathit{spec}\,0\ a = a$ by definition, therefore proving `TCON spec` is normally trivial. Immersions can be constructed so as to be uniform, thus this becomes a relatively simple condition to prove. Nevertheless, `ONE_STEP_THM` would be of little use if were not possible to verify the `TCON_IMMERSION` property without resorting to an explicit induction over time. The following theorem is used:

TC_IMMERSION_ONE_STEP_THM

```
⊢ ∀f imm. IS_IMAP f ∧ UNIFORM imm f ⇒
         (TCON_IMMERSION f imm =
          (∀a. f 0 (f (imm a 0) a) = f (imm a 0) a) ∧
           ∀a. f 0 (f (imm a 1) a) = f (imm a 1) a)
```

The constraint `IS_IMAP` requires the state functions to be *iterated maps*, this is defined by:

```
⊢def IS_IMAP f = ∃init next. IMAP f init next
⊢def IMAP f init next =
        (∀a. f 0 a = init a) ∧ ∀t a. f (SUC t) a = next (f t a)
```

The one-step theorems are, therefore, founded on the use of primitive recursive state functions. Using this approach, the verification of a processor, modelled by an iterated map impl, with respect to an ISA, modelled by an iterated map spec, is dominated by proving that the following set of equations hold for all processor states a:

$$\mathit{impl}\,0\ (\mathit{impl}\ (\mathit{imm}\,a\ 0)\ a) = \mathit{impl}\ (\mathit{imm}\,a\ 0)\ a \tag{1}$$

$$\mathit{impl}\,0\ (\mathit{impl}\ (\mathit{imm}\,a\ 1)\ a) = \mathit{impl}\ (\mathit{imm}\,a\ 1)\ a \tag{2}$$

$$\mathit{spec}\,0\ (\mathit{abs}\,a) = \mathit{abs}(\mathit{impl}\ (\mathit{imm}\,a\ 0)\ a) \tag{3}$$

$$\mathit{spec}\,1\ (\mathit{abs}\,a) = \mathit{abs}(\mathit{impl}\ (\mathit{imm}\,a\ 1)\ a). \tag{4}$$

Equations 1 and 3 are likely to be fairly easy to verify. In the case of the ARM6, the functions impl and spec are complex and the state space is large. This means that verifying equations 2 and 4 is very involved but, nonetheless, conceptually simple.

4 The ARM Architecture

This section gives a brief overview of Version 3 of the ARM 32-bit RISC architecture. The aim here is to introduce the main features of the architecture, rather than going into the full details of the instruction set semantics. A HOL specification of the ARM instruction set architecture is presented in [13]. The official ARM reference manual is [32].

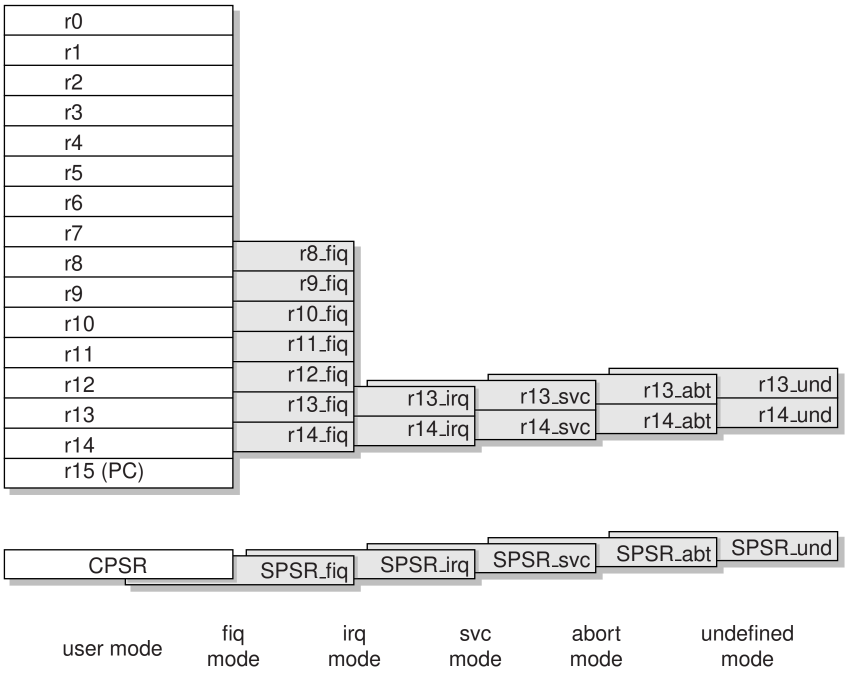


Figure 1: ARM's visible registers.

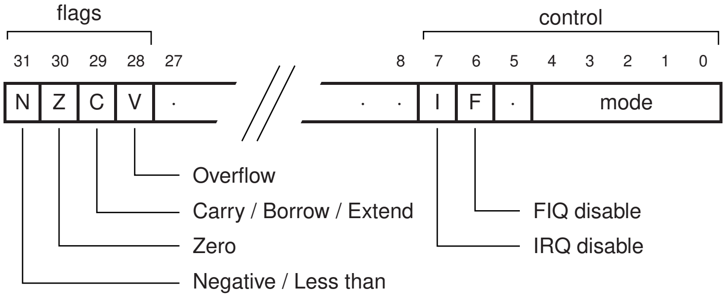


Figure 2: Format of the CPSR.

4.1 Modes and Registers

The ARM architecture supports a number of *operating modes*, allowing privileged access to different register banks: see Figure 1. The ARM6 provides six modes: usr (user), fiq (fast interrupt), irq (standard interrupt), svc (supervisor call), abt (abort), and und (undefined instruction). Each mode provides access to sixteen general purpose registers, some of which are shared with other modes. For example, registers r0 through to r7 and the program counter r15 are the same for all modes. The CPSR (Current Processor Status Register) is a special purpose register used to keep track of the operating mode, interrupt status, and four condition code flags NZCV: see Figure 2. The CPSR is accessible in all modes, and the privileged modes (non-user modes) have their own SPSR (Saved Processor Status Register), and these are used to store the CPSR when entering the privileged mode. Register r14 in each mode is used as a *link* register, this stores the program counter address when taking an exception or sub-routine call.

Class	Instructions
Branch and Branch with Link	B, BL
Data Processing	ADD, ADC, SUB, SBC, RSB, RSC, CMP, CMN, AND, ORR, EOR, MOV, MVN, BIC, TST, TEQ
Multiply and Multiply Accumulate	MUL, MLA
PSR Transfer	MRS, MSR
Single Data Transfer	LDR, STR
Block Data Transfer	LDM, STM
Single Data Swap	SWP
Software Interrupt and Exceptions	SWI

Table 1: The ARM Instruction Classes.

4.2 Memory

In the ARM architecture, memory may be viewed as a linear array of 2^{32} bytes. Data items can be accessed either as bytes or as 32-bit words. When accessing words, address-alignment and byte-ordering are of significance.

4.3 Instruction Classes

The architecture has eight principle instruction classes: see Table 1. Instruction codes that do not correspond with a valid instruction form the *undefined* instruction class. Holes in the instruction space allow the instruction set to be extended for future generations of processors. Undefined instructions are detected at run time and they cause the processor to raise an undefined instruction exception. In the ARM architecture, *all* instructions are conditionally executed at run time. Instructions that are not executed constitute the dynamic class of *unexecuted* instructions. All of the instructions listed in Table 1 are potentially members of this class.

Data processing instructions are used to perform logical and arithmetic operations, and data transfer instructions are used to transfer data between registers and memory. Both of these instruction classes are very flexible. For example, with single data transfers:

- A single byte or word can be transferred.
- Memory addresses are computed using a base address (stored in a register) and an offset.
- The offset can be added (up) or subtracted (down) from the base address.
- Pre- and post-indexing is possible.
- Base register write-back is allowed, facilitating auto-indexing.
- When loading a word from a mis-aligned memory address, the word is rotated right by 8, 16 or 24 places.

With data processing instruction:

- Eight arithmetic and eight logical operations are available.
- One of the source operands can be
    - an immediate value,

  - a register value,
  - a register value shifted by an immediate value,
  - a register value shifted by a register value, or
  - a register value shifted right one place and extended with the carry bit.

  With the exception of shifting by a register value, the offset for a data transfer can be any of the above.

- The condition code flags NZCV can be set.
- A mechanism for retuning from exceptions (changing mode) is available.

Five types of shifting are possible: Logical shift left (LSL), Logical shift right (LSR), Arithmetic shift right (ASR), Rotate right (ROR) and Rotate right extend (RRX).

4.4 Features not Modelled

The following features have not been tackled:

- Thumb instructions.
- Co-processor instructions.
- External interrupts.

The Thumb instruction set can be viewed as a compressed form of a subset of the ARM instruction set. Thumb instructions are 16-bits long (which allows for a higher code density) and compliant ARM processors can switch to and from Thumb mode (flagged with bit five of the CPSR) in order to execute these instructions. This instruction set was introduced with the ARM7 family of processors and so it is not considered here.

Co-processor instructions allow the instruction set to be extended through the addition of co-processors. A strategy for modelling and verifying ARM designs with co-processor instructions has yet to be devised.

The ARM architecture supports two types of external interrupts: standard and fast. These exceptions have not been modelled. Accurately modelling exceptions at the ISA level is not trivial because exceptions tend to expose implementation dependent (pipelined) behaviour. One must consider exception priority and order, with respect to other (possibly simultaneous) exceptions and the instruction sequence. The correctness of systems with input has been studied in an abstract setting, [11], but further work is needed in this area.

5 The ARM Instruction Set Architecture Specification

The HOL specification of the ARM instruction set is described in [13]. This specification was used as the basis for the ISA model used in the ARM6 verification, but a number of modifications have been made. The specification was ported to the latest version of HOL: the Kananaskis release, instead of Taupo. This release adds a number of new features, some of which have been used to refine parts of the ISA specification. Other changes were made in order to reduce the verification effort and these features may be reinstated for later verification attempts. This section outlines the main modifications.

5.1 Unpredictable Behaviour

Some ARM instructions give rise to unpredictable (processor dependent) behaviour. For example, the instruction

```
EOR r1, r15, r2 ASR r3
```

is unpredictable because the first operand is the program counter. This was modelled in HOL by using an arbitrary value `ARB` to represent an unspecified state. However, this approach is not suited to processor verification using the definition of correctness presented in Section 3.1. This is because the ARM6 will always be in some concrete state and it does not make sense to compare these processor states (by equality after data abstraction) with the ISA state `ARB`. For the purposes of the initial verification attempt the ISA model was changed so as to conform with ARM6 behaviour. Other approaches will be considered in due course.

5.2 Words

In [13] the model of 32-bit words (the principle data type for the ARM architecture) was not wholly satisfactory. Words were defined with the following data type declaration:

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

This approach is flawed in that term equality does not correspond with word equality. That is to say, the following one-to-one theorem should *not* be true, but it is:

```
⊢ (W32 a = W32 b) = (a = b)
```

This precludes a clean presentation of word theorems i.e. some key results (for example, the commutative ring properties of word arithmetic) must be presented as conditional equations or with the inclusion of casting maps. This problem has been addressed with the development of a theory of n-bit words, which has been added to the HOL system. Words are constructed as an equivalence type over the natural numbers and, as such, the following theorem holds for 32-bit words:

```
⊢ (w32 a = w32 b) = (a MOD (2 EXP 32) = b MOD (2 EXP 32))
```

where `w32:num→word32` maps numbers to words.[2] The new word theory supplies numerous (equational) theorems about the word data type, and these have been used in the verification of the ARM6. Operator overloading is also provided: see Table 2. The pretty-printing of word ground terms has also been made possible with the Kananaskis release of HOL, thus the term `w32 10` is displayed as `0xA`.

The same model of words is used at both the instruction set and micro-architecture levels of abstraction. This reduces the *semantic gap* between the two levels of abstraction, and the sharing of primitive functions between ISA and MA specifications is advantageous. The correctness definition covers both the *data* and *control* aspects of a design, and hence it is important to be able to verify the equivalence of word expressions. Having an executable model of words also provides a means to test specifications through the execution of real code.

---

[2] The function name `w32` is actually a pseudonym for `n2w` in `word32Theory`.

Operation	Function Name	Symbol	Type
One's Complement	`word_1comp`	`NOT`	`word32→word32`
Two's Complement	`word_2comp`	`~`	`word32→word32`
Addition	`word_add`	`+`	`word32→word32→word32`
Subtraction	`word_sub`	`-`	`word32→word32→word32`
Multiplication	`word_mul`	`*`	`word32→word32→word32`
Bitwise Conjunction	`word_and`	`&`	`word32→word32→word32`
Bitwise Disjunction	`word_or`	`|`	`word32→word32→word32`
Bitwise Exclusive Disjunction	`word_eor`	`#`	`word32→word32→word32`
Logical Shift Left	`word_lsl`	`<<`	`word32→num→word32`
Logical Shift Right	`word_lsr`	`>>>`	`word32→num→word32`
Arithmetic Shift Right	`word_asr`	`>>`	`word32→num→word32`
Rotate Right	`word_ror`	`#>>`	`word32→num→word32`
Rotate Right with Extend	`RRX`	None	`bool→word32→word32`

Table 2: Word operations.

5.3 Data Types

Some rationalizations have been made to the specification through making better use of data types. For example, some *nested if* statements have been replaced with *case* statements.

Of particular significance is the modelling of the general purpose registers. In the original ISA specification, the register bank state space was modelled using six maps, each corresponding with the registers of a given mode. This has been simplified by using one map from register names to 32-bit words. The program status registers have also been modelled using a single map. The following type gives names to each of the registers:

```
Hol_datatype ‘register = r0 | r1 | r2 | ... | r13_und | r14_und‘
Hol_datatype ‘psrs = CPSR | SPSR_fiq | ... | SPSR_abt | SPSR_und‘
```

Thus, the register bank is modelled with the type `register→word32` and the program status registers have type `psrs→word32`. Kananaskis provides a mechanism for type abbreviations, and this has been used as follows:

```
type_abbrev("mem", ‘‘:word30→word32‘‘)
type_abbrev("reg", ‘‘:register→word32‘‘)
type_abbrev("psr", ‘‘:psrs→word32‘‘)
```

This enables the ARM registers and main memory to be given the type names `mem`, `reg` and `psr`. Note that HOL will still *display* these types in a fully expanded form: see [18].

The following data type declaration is made:

```
Hol_datatype ‘exception = reset | undefined | software | address |
                          pabort | dabort | interrupt | fast‘
```

This enables exceptions to be named (improving readability) and, with Kananaskis, one can utilize the generated functions `num2exception` and `exception2num`. Note that the exception `address` is redundant with respect to Version 3/4 of the ARM instruction set—it is a legacy of ARM's 26-bit memory addressing. This exception type has been added to the type declaration because it provides a straightforwards means to compute exception vectors, which in turn simplifies the specifications.

Instruction classes are named by the following type declaration:

```
Hol_datatype ‘iclass = swp | mrs_msr | data_proc | reg_shift |
                       ldr | str | br | swi_ex | undef | unexec‘
```

The function `DECODE_INST:num→iclass` is used to decode instructions (at both levels of abstraction), and the ISA next state function is modified accordingly. The condition codes are also given a data type definition:

```
Hol_datatype ‘condition = EQ | CS | MI | VS | HI | GE | GT | AL‘
```

There are sixteen condition codes in total but the other eight are simply negations of the above. The definition of the ISA function `CONDITION_PASSED` is changed to be a case expression over the above data type.

5.4 Memory Access Operations

The following memory access operations are defined:

```
⊢def MEMREAD mem addr = mem (TO_W30 addr)
⊢def MEM_READ b = if b then MEM_READ_BYTE else MEM_READ_WORD
⊢def MEM_WRITE b = if b then MEM_WRITE_BYTE else MEM_WRITE_WORD
```

These functions help tidy up the presentation of the ISA specification. The function `MEMREAD` is used for word aligned memory access and is also used in the ARM6 specification, as is the function `MEM_WRITE`.

5.5 Features Dropped

The original ISA specification had a simplistic memory model, but memory aborts were modelled in the case of address mis-alignment. The modified specification drops all memory abort handling—it is assumed that abort exceptions are never raised. This means that the only exceptions that are handled are software interrupts and undefined instructions. Handling all of the exception types will require a more detailed memory model and a more sophisticated model of correctness.

It is assumed that little-endian word ordering is used throughout. This is a relatively minor change and verifying a version with big-endian ordering should not be difficult.

Finally, two instruction classes have been left out: multiply and multiply accumulate; and block data transfer. The main reason for this is that their verification would not be straightforward. Both of these instruction classes are implemented (by the ARM6) with an iterated machine cycle i.e. the same cycle is repeated until completion. Multiplication is implemented using a modified Booth's algorithm: the data path is used to carry out additions/subtractions and shifts. In most modern processors the ALU can perform multiplication directly and so the multiplication algorithm does not encroach upon the main processor control logic. Such an implementation would be readily verifiable (at the level of abstraction covered by this report) and, therefore, it is not significant dropping this instruction class. However, verifying the block data transfer instruction class remains a challenge.

6 The ARM6 Implementation

The ARM6 design is split into control and data path components. The data path is presented in Section 6.1 and the control components are introduced in Section 6.2. The pipeline behaviour explained in Section 6.2.2.

6.1 The Data Path

The data path performs various data processing operations throughout the course of an instruction's execution. Figure 3 shows the data path for the ARM6. The main units are the field extractor/extender, the shifter and the ALU. There are two data buses, labelled A and B, supplying the inputs to the ALU; with bus B passing through the shifter. The behaviour of the data path is determined by the control unit. This section gives details of this data path control logic—the pipeline control is covered in Section 6.2. Section 6.1.1 defines the state space of the data path, and the following sections outline the functionality of the data path.

6.1.1 The Data Path State Space

The state space of the data path is modelled in HOL using the following type declaration:

```
Hol_datatype `dp = DP of reg=>psr=>word32=>word32=>word32=>word32`
```

The components are named as follows:

```
DP reg psr areg din alua alub
```

The components are:

`reg`	`reg`	The register bank
`psr`	`psr`	The program status registers
`areg`	`word32`	The address register
`din`	`word32`	The data-in register
`alua`	`word32`	ALU source register A
`alub`	`word32`	ALU source register B.

The register bank and program status register types are taken from the ISA model (see [13]). However, the program counter register `r15` is treated differently: at the ISA level the program counter stores the address of the instruction being executed, whereas at this level the program counter is the address of the instruction being fetched. A program status register is conceptually a 32-bit word but with version 3/4 of the ARM ISA many of these bits are redundant. This means that the ARM6 does not use 32-bits in storing each status register. This detail is omitted in the implementation described here.

6.1.2 The Memory Interface

The main memory is accessed using the address register `areg`, which is the source/destination address for memory reads/writes. When writing to memory the data source is always bus B. When reading from memory the data is fed to the pipeline (see Section 6.2.2) and to the register `din`. Multiplexers are used to select the values used in updating the registers `din` and `areg`. The `din` register can either: (*i*) remain unchanged, (*ii*) take the value of the current instruction code `ireg`, or (*iii*) be updated with a word from memory. The `areg` register can take one of four values: (*i*) an exception vector, (*ii*) the value of register `r15`, (*iii*) the output of the ALU, or (*iv*) the output of the address increment.

The ARM6 data path actually contains a byte replicator between bus B and the memory, which is used when storing single bytes. This detail has been hidden within the definition of the function `MEM_WRITE`, which is taken from the modified ISA specification: see Section 5.4.

Table 3 lists the main functions that are responsible for the memory interface behaviour, and each of these functions is defined in Appendix A.1.

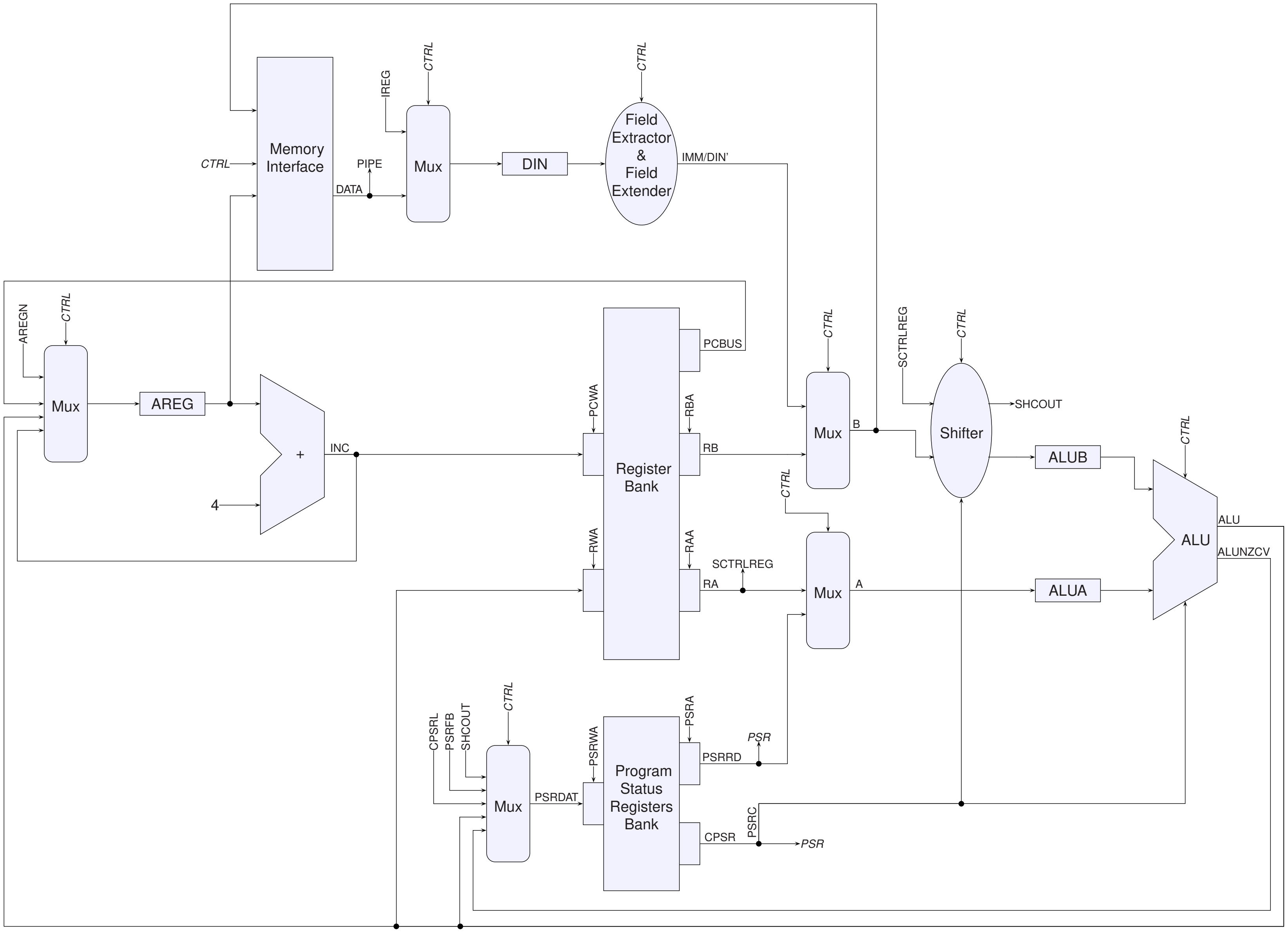


Figure 3: The ARM6 Data Path.

Function	Description
`AREG`	Selects the value to be written to the AREG latch.
`AREGN1`	Gives the exception vector to be taken on the next cycle.
`DIN`	Selects the value to be written to the DIN latch.
`DINWRITE`	Indicates whether the DIN latch is to be updated.
`NBW`	Indicates whether the next memory access is for a byte or word.
`NRW`	Indicates whether the next memory access is a read or write.

Table 3: Functions used in specifying the memory interface.

Function	Description
`PCWA`	Controls whether the incementor bus updates r15.
`RAA`	Determines which register is written to the RA bus.
`RBA`	Determines which register is written to the RB bus.
`RWA`	Controls the (ALU output) update of the Register Bank.

Table 4: Functions used in specifying general purpose register access.

6.1.3 The Field Extractor/Extender

The field extractor/extender takes input from the register `din` and the result is fed to the bus B multiplexer (Section 6.1.6). The unit performs the task of extracting (and if necessary extending) an immediate field from the instruction register (for example, when accessing the branch offset). It is also used to select the appropriate bits from `din` when executing byte load (or swap) instructions. The field extractor/extender is modelled in HOL using the function `FIELD`, which is defined in Appendix A.2.

6.1.4 The General Purpose Registers

Three read ports provide access to the general purpose registers, the outputs are labelled `PCBUS`, `RA` and `RB`. The program counter bus gives direct access to register `r15` and is used when setting `areg` just prior to an instruction fetch. There are two write ports: one is used to update the program counter (after incrementing the address), and the other is connected to the ALU output. Table 4 lists the functions used in accessing the registers, and they are defined in Appendix A.3.

6.1.5 The Program Status Registers

The program status registers are accessed using two read ports and one write port. One of the read ports is dedicated to the CPSR, which is fed to the control unit. The CPSR status flags and mode influence the execution of instructions. The carry flag is also fed to the shifter and the ALU. The write port is used to update either the CPSR or an SPSR. A multiplexer is used to construct a new PSR word using data from the shifter (i.e. the carry out), the ALU and the control unit. Table 5 lists the functions used in accessing the PSRs, and they are defined in Appendix A.4.

6.1.6 The Data Buses and Shifter

There are two buses: bus A takes a value from a general purpose register or from a program status register; bus B passes through the shifter and takes a value from the field extractor/extender (Section 6.1.3) or from the register bank. The 32-bit bus values are stored in registers `alua` and `alub`

Function	Description
`PSRA`	Indicates whether the CPSR or an SPSR is written to the PSRRD bus.
`PSRDAT`	Selects the value to be written to the PSRDAT bus.
`PSRFBWRITE`	Controls the PSRFB latch update.
`PSRWA`	Controls the updating of the Program Status Register Bank.

Table 5: Functions used in specifying program status register access.

Function	Description
`BUSA`	Selects the value to be written on bus A.
`BUSB`	Selects the value to be written on bus B.
`SCTRLREGWRITE`	Controls the SCTRLREG latch update.
`SHIFTER`	Gives the output of the barrel shifter, which includes a carry out.

Table 6: Functions used in specifying the shifter and bus behaviour.

respectively. The shifter also outputs a carry flag, which is fed to the control unit and is used in updating the PSRs (Section 6.1.5). The (barrel) shifter can perform one of five types of shift operation (LSL, LSR, ASR, ROR or RRX). A carry in is input from the PSR unit. The shift amount is either: zero; an immediate value (taken from `ireg` in the control unit); a register shift amount (taken from `sctrlreg` in the control unit); or it is an alignment amount (constructed from `oareg` in the control unit). The last case is used when loading bytes or when reading from a mis-aligned memory address. Table 6 lists the functions used to specify the shifter and bus behaviour, and they are defined in Appendix A.4.

6.1.7 The ALU

The ALU performs arithmetic and logical operations on 32-bit words. The operands are read from `alua` and `alub`. The ALU also takes the CPSR carry flag, which is needed for executing the ADC and SBC instructions. The unit outputs a 32-bit word together with four status flags NZCV. The ALU is used to: (*i*) execute the operations from the data processing instruction class; (*ii*) simply select `alua` or `alub`; (*iii*) implement the auto indexing for data transfers; and (*iv*) compute branch destination and link register values. Table 7 lists the functions used in specifying the ALU behaviour, and they are defined in Appendix A.6.

6.2 The Control Unit

The behaviour of the pipeline and data path is determined by the control unit. The state space of this unit is given in Section 6.2.1. The ARM6 has a three stage pipeline with fetch, decode and execute stages. The fetch and decode stages always take one machine cycle to complete. Some

Function	Description
`ALU6`	Gives the output of the ALU, which includes the NZCV flags.
`ALUAWRITE`	Controls the ALUA latch update.
`ALUBWRITE`	Controls the ALUB latch update.

Table 7: Functions used in specifying the ALU behaviour.

instructions require more than a single machine cycle to execute, and this means that the fetch and decode stages do not proceed on all cycles. Details of this pipeline behaviour are given in Section 6.2.2.

6.2.1 The Control State Space

The state space of the control unit is modelled in HOL using the following type declaration:

```
Hol_datatype `ctrl = CTRL of word32=>bool=>word32=>bool=>word32=>bool=>word32=>word32=>
                             bool=>bool=>bool=>iclass=>iseq=>
                             num=>bool=>bool=>word32=>word32=>num`
```

The components are named as follows:

```
CTRL pipea pipeaval pipeb pipebval ireg iregval apipea apipeb
     ointstart onewinst opipebll nxtic nxtis
     aregn nbw nrw sctrlreg psrfb oareg
```

The first eight components give the state of the pipeline:

`pipea`	`word32`	Stores a fetched instruction code
`pipeaval`	`bool`	Indicates whether `pipea` has been invalidated
`pipeb`	`word32`	Stores an instruction code prior to decode
`pipebval`	`bool`	Indicates whether `pipeb` has been invalidated
`ireg`	`word32`	Stores the instruction code after decode
`iregval`	`bool`	Indicates whether `ireg` has been invalidated
`apipea`	`word32`	The source memory location for `pipea`
`apipeb`	`word32`	The source memory location for `pipeb`.

The components `apipea` and `apipeb` are not taken from the actual ARM6 design. They have been added in order to implement correct behaviour (see Section 3.1) when writing to the memory addresses `pc` and `pc-4`: see Examples 3 and 4 in Section 6.2.2.

The next five components control the pipeline behaviour:

`ointstart`	`bool`	Indicates whether an exception has been raised
`onewinst`	`bool`	Indicates whether a new instruction has been decoded
`opipebll`	`bool`	Controls whether an instruction will be fetched
`nxtic`	`iclass`	The class of the next instruction to be executed
`nxtis`	`iseq`	The next instruction sequence (for multi-cycle execution).

The remaining components relate to the data path control:

`aregn`	`num`	An exception vector
`nbw`	`bool`	Controls whether memory access is for a byte or a word
`nrw`	`bool`	Controls whether memory access is read or write
`sctrlreg`	`word32`	Stores a shift value, which is read from the register bank
`psrfb`	`word32`	Stores a word read from the PSRs
`oareg`	`num`	The two least-significant bits of the address register.

Note that `aregn` is naturally a 3-bit word[3] and `oareg` is a 2-bit word, but for convenience natural numbers are used. The instruction class type `iclass` is defined at the ISA level: see Section 5. The instruction sequence type is declared with:

```
Hol_datatype `iseq = t3 | t4 | t5 | t6`
```

These values could be encoded using 2-bit words.

---

[3] In fact a single bit would suffice here because only two types of exception are modelled: undefined instructions and software interrupts.

6.2.2 The Pipeline

The behaviour of the pipeline is explained in this section with the aid of some sample code. This code has been designed to demonstrate some of the intricacies of the design, and is otherwise without purpose. In fact, it is inadvisable to write code of the form found in Examples 3 and 4.[4] The pipeline has three stages: fetch, decode and execute. A *single-cycle* instruction is in the execute stage for just one cycle, and in the pipeline for three cycles in total. Multi-cycle instructions require more than a single cycle at the execute stage; consequently they are in the pipeline for more than three cycles. The distinction between single- and multi-cycle instructions should become clear in the following examples.

Instructions are fetched from the current program counter address `pc`, which is register `r15`. Therefore, `pc` is *not* the address of the instruction being *executed*. In general, the instruction being executed is from address `pc-8` and the instruction being decoded is from address `pc-4`. This is further complicated with multi-cycle instructions because the program counter is incremented on the first execute cycle, which means that the execute and decode instructions then correspond with addresses `pc-12` and `pc-8` respectively.

Example 1. Consider the following fragment of ARM code:

```
a: sub   pc, pc, #4         @ 0xE24FF004
b: swp   r0, r1, [r2]       @ 0xE1020091
c: add   r0, r1, r2, ROR r3 @ 0xE0810372
d: b     a                  @ 0xEAFFFFFB
e: mvn   r0, #3             @ 0xE3E00003
f: cmp   r0, r1             @ 0xE1500001
```

The letters to the left of the colon are instruction labels. The commented hexadecimal numbers to the right (after the @ symbol) are the 32-bit encodings for each of the respective instructions. When executing the code in HOL these values are used to track the flow of instructions through the pipeline. The instructions have the following affect:

1. The subtract (`sub`): This instruction causes a branch to address `pc-4`. This is the address of the second instruction, and so this is effectively a no-op instruction.

2. The swap (`swp`): This takes four cycles to execute; involving a read from, and write to, main memory.

3. The add (`add`): This takes two cycles to execute. The fist cycle is used to store register `r3` in the control unit component `sctrlreg`.

4. The branch (`b`): The pipeline is returned to the initial state i.e. the pipeline state prior to executing the `sub` instruction. The `mvn` and `cmp` instructions, having been fetched, are not decoded or executed.

Figure 4 shows the pipeline flow for this sequence of instructions. Instruction mnemonics and labels are shown on the *y*-axis and time increases along the *x*-axis. Each block represents the operation (fetch, decode or execute) performed on an instruction. Multi-cycle instructions are emphasised by marking the additional execute stages with a lighter colour. Observe that during the execution of the `swp` and `add` instructions, a fetch occurs on the first execute cycle and a decode occurs on the last execute cycle. This explains the significance of the component `pipeb`, which stores an

---

[4] It is good programming practise, when writing assembly code, to ensure that the program and data address spaces are clearly demarcated. Ignoring this guideline leads to the production of obfuscated code and in some cases it will expose (the pipeline) details of a particular processor implementation.

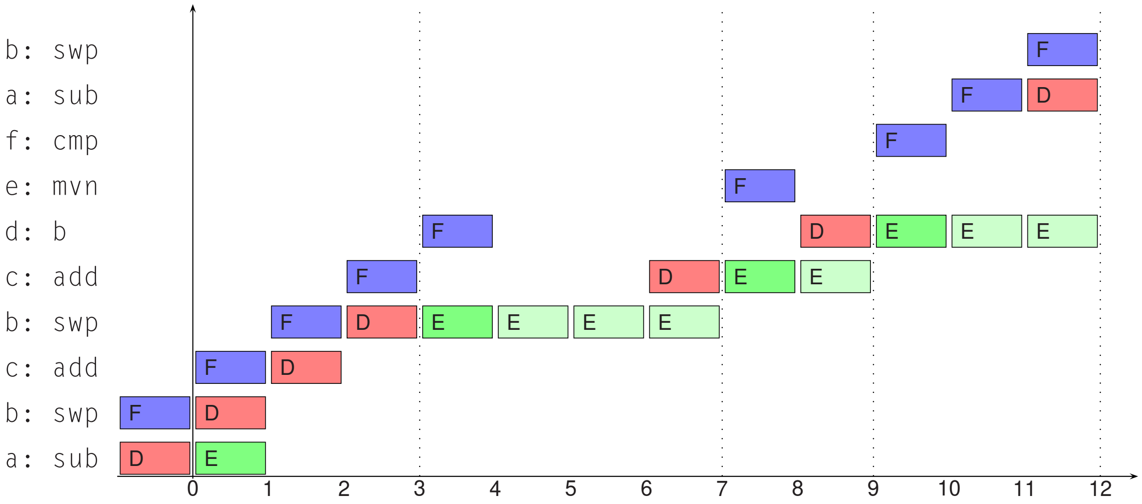


Figure 4: Pipeline flow for Example 1.

State \ Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12
pipea,pipeaval	b,T	c,F	b,T	c,T	d,T	d,T	d,T	d,T	e,T	e,T	f,T	a,T	b,T
pipeb,pipebval	b,T	c,F	b,T	c,T	c,T	c,T	c,T	d,T	d,T	e,T	f,T	a,T	b,T
ireg,iregval	a,T	b,F	c,F	b,T	b,T	b,T	b,T	c,T	c,T	d,T	d,T	d,T	a,T
ointstart	F	F	F	F	F	F	F	F	F	F	F	F	F
onewinst	T	T	T	T	F	F	F	T	F	T	F	F	T
opipebll	T	T	T	T	F	F	F	T	F	T	T	T	T
nxtic	`data_proc`	`swp`	`reg_shift`	`swp`	`swp`	`swp`	`swp`	`reg_shift`	`reg_shift`	`br`	`br`	`br`	`data_proc`
nxtis	`t3`	`t3`	`t3`	`t3`	`t4`	`t5`	`t6`	`t3`	`t4`	`t3`	`t4`	`t5`	`t3`

Table 8: The pipeline behaviour for Example 1.

instruction code prior to decode. (If the fetch and decode were always simultaneous then `pipeb` would be redundant.) A fetch occurs on each execute cycle of the branch. With single-cycle instructions fetch and decode occur simultaneously with the execute. From Figure 4 it is clear that all but the first instruction is multi-cycle. The first instruction (`sub`) is single-cycle, but a further two cycles are needed to re-fill the pipeline after `pc` is modified. Strictly speaking, these extra cycles are deemed not to constitute part of the instruction's execute stage but they are of significance from a correctness standpoint.

Table 8 shows the pipeline behaviour with respect to the first thirteen components of the control unit. The instruction labels have been used, in place of the instruction codes, to indicate the state of the components `pipea`, `pipeb` and `ireg`. Observe that the components `pipeaval`, `pipebval` and `iregval` are used to implement the re-filling of the pipeline (tagging invalidated instruction codes) after the `sub` instruction writes to register `pc`. This differs from the branch instruction, which takes three cycles to *execute*, re-filling the pipeline in the process.

The states are grouped together into blocks, with the cycle at the start of each block underlined. This grouping corresponds with the temporal abstraction used in verifying the design. The underlined cycles are specified by an immersion, which gives the times at which data abstraction yields ISA states. These states are characterised by the fact that the pipeline is ready for the first execute cycle of the instruction in `ireg`.

State \ Cycle	0	1	2	3	4	5
pipea,pipeaval	b,T	c,T	d,T	e,T	0x4,T	0x8,T
pipeb,pipebval	b,T	c,T	d,T	e,T	0x4,T	0x8,T
ireg,iregval	a,T	b,T	c,T	c,T	c,T	0x4,T
ointstart	F	F	T	F	F	F
onewinst	T	T	T	F	F	T
opipebll	T	T	T	T	T	T
nxtic	data_proc	undef	swi_ex	swi_ex	swi_ex	data_proc
nxtis	t3	t3	t3	t4	t5	t3

Table 9: The pipeline behaviour for Example 2.

Example 2. This example considers the pipeline flow for the unexecuted and undefined instruction classes. When an undefined instruction code is encountered an exception is raised, causing a branch to memory address `0x4`. The following code fragment forms part of a stub exception handler:

```
0x0: movs  r15,#32     @ 0xE3B0F020 - Reset
0x4: movs  r15,r14     @ 0xE1B0F00E - Undefined Instruction
0x8: movs  r15,r14     @ 0xE1B0F00E - Software Interrupt
0xC: subs  r15,r14,#4  @ 0xE25EF004 - Prefetch Abort
```

This simplistic exception handler ignores each exception and returns to the link address, which is stored in register `r14`.

The code containing the unexecuted and undefined instructions is given below:

```
a: andnv r0, r0, r0             @ 0xF0000000
b: undef                        @ 0xE6000010
c: mvn   r0, #3                 @ 0xE3E00003
d: cmp   r0, r1                 @ 0xE1500001
e: str   r12, [r1,r5, ASR #6]!  @ 0xE7A1C345
```

The first instruction is never[5] executed and the second instruction has an undefined instruction code. Figure 5 shows the pipeline flow and Table 9 shows the state component trace for this example. The unexecuted instruction `andnv` is initially decoded and identified as a data processing instruction. When the instruction comes to being executed the actual instruction class (`ic`) is determined to be `unexec`. The value `ic` is computed by examining the condition code (bits 31–28 of the instruction register) and, as appropriate, performing tests on the program status flags. If the class is determined to be `unexec` then the execute stage does not proceed. The undefined instruction gives rise to an exception, which is handled by the `swi_ex` instruction class; the execution of which is shown in Figure 5 using different coloured boxes (labelled with an I). Exceptions have a similar behaviour to branch (with link) instructions and they take three cycles to execute. The control component `aregn` has value 1 at cycle two and the exception causes a branch to address `aregn*4`.

Example 3. This example has been devised to illustrate the execution of self-modifying code. The ARM6 has a von Neumann style architecture—program and data memories are not separated. This means that it is possible for a program to modify the memory at locations coincident with the code being executed. Therefore, one must consider what happens when instructions in the pipeline are overwritten in memory. The following code fragment contains store and swap instructions, both of which try to modify the code itself:

---

[5] Note that using the never (`nv`) condition code is strongly discouraged by ARM because it utilizes an area of the instruction space reserved for future use.

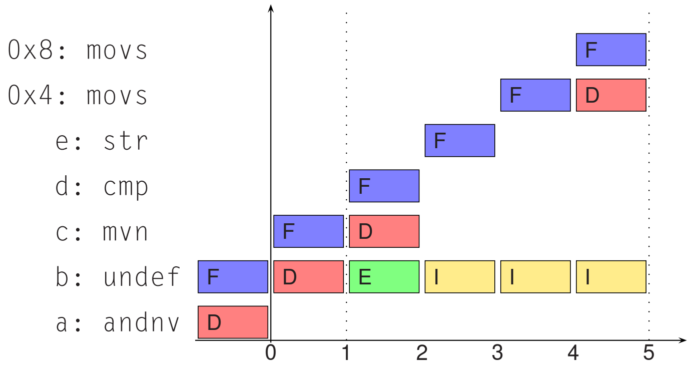


Figure 5: Pipeline flow for Example 2.

```
a: ldr   r0, f          @ 0xE59F000C
b: adr   r1, e          @ 0xE28F1004
c: swp   r2, r0, [r1]   @ 0xE1012090
d: str   r2, f          @ 0xE58F2000
e: cmp   r3, #1         @ 0xE3530001
f: mvn   r4, #2         @ 0xE3E04002
g: mov   r5, #3         @ 0xE3A05003
```

The instruction labelled `f` is first read into register `r0`. The *pseudo-instruction*[6] `adr` causes `r1` to take the address of the instruction with label `e`. The swap instruction stores the instruction code of `f` (which is in `r0`) at address `e`. The old instruction code at `e` is stored in `r2`. The store instruction then stores this instruction code at address `f`. At this point the instructions labelled `e` and `f` have been swapped in memory. The remaining instruction perform simple data-processing operations.

When this sequence of instructions is executed on an ARM6, the `cmp` and `mvn` instructions are executed in the original program order, even though the instructions are swapped in memory. This is because in both cases the instructions are fetched prior to updating the memory. The HOL model of the ARM instruction set architecture does not take account of this. In the abstract ISA model the `mvn` instruction is executed before the `cmp` instruction. Consequently there is an inconsistency in behaviour between the two levels of abstraction, preventing a formal verification of the design. This needs to be addressed, and a number of options have been considered:

1. Leave the ARM6 model unchanged and investigate one of the following:
    (a) Modify the ISA specification so as to implement (highly abstract) pipeline behaviour.
    (b) Modify the ISA specification—abstracting out details of instruction sequencing. This would require a small modification to the ISA model, but necessitate a more elaborate definition of correctness.
    (c) Deem such self-modifying code to be unpredictable. It would then be necessary to modify the correctness definition, so as to accommodate unpredictable behaviour.

---

[6] Pseudo-instructions are converted by the assembler into a real ARM instructions. In this example the `adr` becomes an `add` instruction.

2. Change the data behaviour of the ISA and ARM6 models. If the `swp` and `str` instructions do not write over (clobber) the following two instructions, then correctness holds. This involves minor changes to both specifications and preserves the ARM6 pipeline behaviour.

3. Change the pipeline behaviour of the ARM6, so as to correctly implement the abstract ISA behaviour. This has been implemented using data forwarding, and consequently it only affects the timing of `str` instructions when writing over the following instruction (one extra cycle is needed).

Options 2 and 3 are implemented with the no-clobber and data forwarding versions of the ARM6 respectively. These versions are very similar and the small differences in the control logic may be seen in Appendix C. Unless otherwise stated, this report will focus on the no-clobber implementation. Option 1(a) is not ideal because it would involve changing the state space of the ISA. Options 1(b) and 1(c) are promising and will be investigated in due course.

Figure 6 shows the pipeline flow for this example. In Figure 6(a) the instruction sequence corresponds with the actual ARM6 behaviour. This is also the pipeline flow for Option 2 above. In Figure 6(b) data forwarding is introduced, which implements the abstract ISA behaviour. Data forwarding is a technique used to resolve dependencies in pipelines—completed computations are used to update the pipeline state, thus avoiding the need to re-load (modified) data from its original source. In Figure 6(b) the boxes labelled **W-a** mark instances where the word being stored in memory is also forwarded to `pipea`. This is triggered when there is a match between the current address register (see Section 6.1.2) and `apipea`. This could be implemented in hardware without great difficulty.

The pipeline states for this example are shown in Table 10. The true ARM6 behaviour is shown in Table 10(a) and this can be contrasted with Table 10(b) which shows the data forwarding behaviour.

Example 4. In the previous example the instruction fetched from address `pc` (stored in `pipea`) is compromised by a memory store to the `pc` address. The following code fragment is used to illustrate a memory store to address `pc-4`.

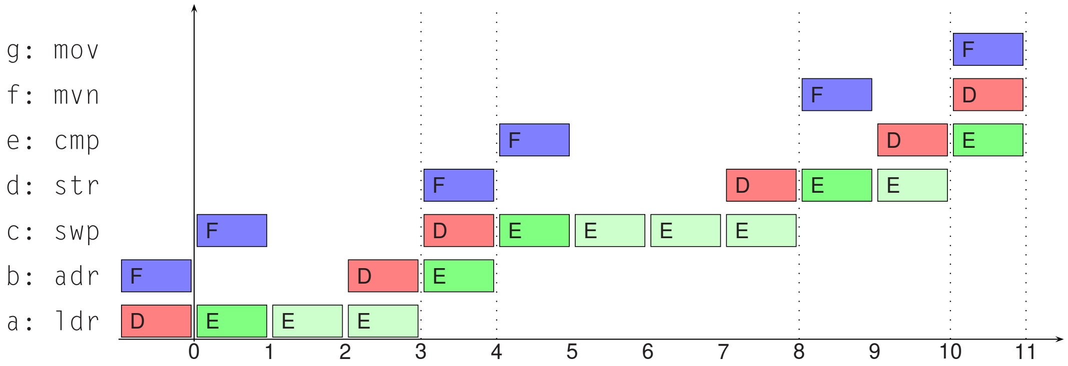
```
a: ldr   r0, e   @ 0xE59F0008
b: str   r0, c   @ 0xE50F0004
c: cmp   r3, #1  @ 0xE3530001
d: mvn   r4, #2  @ 0xE3E04002
e: mov   r5, #3  @ 0xE3A05003
```

In this example data forwarding does not provide a simple fix. This is because the `cmp` instruction (labelled `c`) is decoded simultaneously with the memory store; however it then becomes necessary to decode the `mov` instruction, as this is forwarded to `pipeb` replacing the `cmp`. The original ARM6 pipeline behaviour is modified with the introduction of an extra cycle. In this cycle the `mov` decode is carried out and the `mvn` instruction is also re-fetched. The fetch may seem unnecessary (because the required instruction code is already in the pipeline) but it keeps the control logic fairly simple.

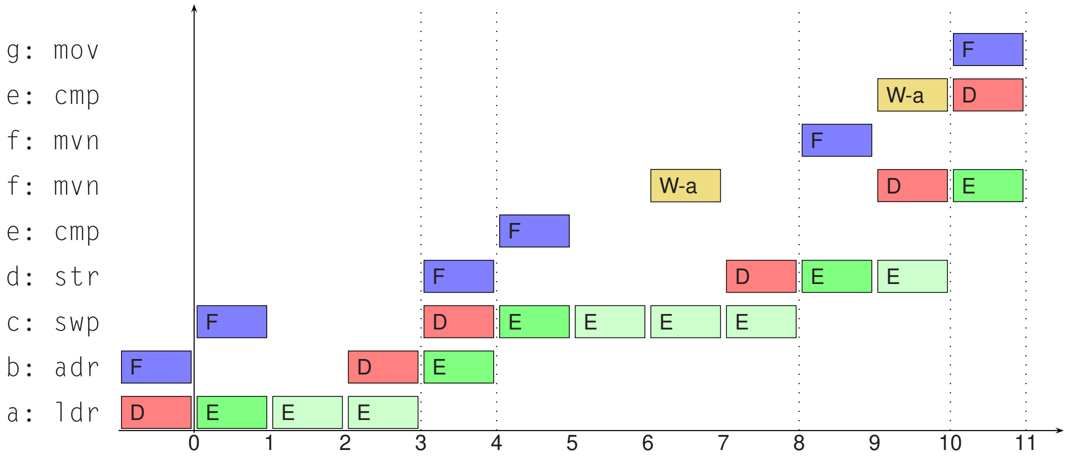
The pipeline flow for this example, with and without data forwarding, is shown in Figure 7. The box labelled **W-b** shows data forwarding to `pipeb`. Table 11 gives the state changes for this example.

6.2.3 Formal Specification

The functions used to specify the ARM6 pipeline control are listed in Table 12. The formal specification of these functions is given in Appendix B. Most of these functions are inherently simple in definition and some have been further simplified by the absence of (most) exceptions, and by omitting the multiply and block data transfer instruction classes. Daniel Schostak's phase level



(a) Actual ARM6 behaviour.



(b) With data forwarding.

Figure 6: Pipeline flow for Example 3.

State \ Cycle	0	1	2	3	4	5	6	7	8	9	10	11
pipea,pipeaval	b,T	c,T	c,T	c,T	d,T	e,T	e,T	e,T	e,T	f,T	f,T	g,T
pipeb,pipebval	b,T	b,T	b,T	c,T	d,T	d,T	d,T	d,T	e,T	e,T	f,T	g,T
ireg,iregval	a,T	a,T	a,T	b,T	c,T	c,T	c,T	c,T	d,T	d,T	e,T	f,T
ointstart	F	F	F	F	F	F	F	F	F	F	F	F
onewinst	T	F	F	T	T	F	F	F	T	F	T	T
opipebll	T	F	F	T	T	F	F	F	T	F	T	T
nxtic	ldr	ldr	ldr	data_proc	swp	swp	swp	swp	str	str	data_proc	data_proc
nxtis	t3	t4	t5	t3	t3	t4	t5	t6	t3	t4	t3	t3

(a) Actual ARM6 behaviour.

State \ Cycle	0	1	2	3	4	5	6	7	8	9	10	11
pipea,pipeaval	b,T	c,T	c,T	c,T	d,T	e,T	e,T	[f],T	[f],T	f,T	[e],T	g,T
pipeb,pipebval	b,T	b,T	b,T	c,T	d,T	d,T	d,T	d,T	[f],T	[f],T	[e],T	g,T
ireg,iregval	a,T	a,T	a,T	b,T	c,T	c,T	c,T	c,T	d,T	d,T	f,T	[e],T
ointstart	F	F	F	F	F	F	F	F	F	F	F	F
onewinst	T	F	F	T	T	F	F	F	T	F	T	T
opipebll	T	F	F	T	T	F	F	F	T	F	T	T
nxtic	ldr	ldr	ldr	data_proc	swp	swp	swp	swp	str	str	data_proc	data_proc
nxtis	t3	t4	t5	t3	t3	t4	t5	t6	t3	t4	t3	t3

(b) With data forwarding.

Table 10: The pipeline behaviour for Example 3.

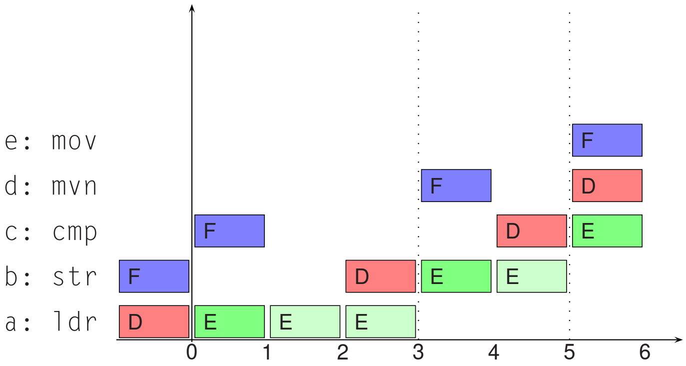
State \ Cycle	0	1	2	3	4	5	6
pipea,pipeaval	b,T	c,T	c,T	c,T	d,T	d,T	e,T
pipeb,pipebval	b,T	b,T	b,T	c,T	c,T	d,T	e,T
ireg,iregval	a,T	a,T	a,T	b,T	b,T	c,T	d,T
ointstart	F	F	F	F	F	F	F
onewinst	T	F	F	T	F	T	T
opipebll	T	F	F	T	F	T	T
nxtic	ldr	ldr	ldr	str	str	data_proc	data_proc
nxtis	t3	t4	t5	t3	t4	t3	t3

(a) Actual ARM6 behaviour.

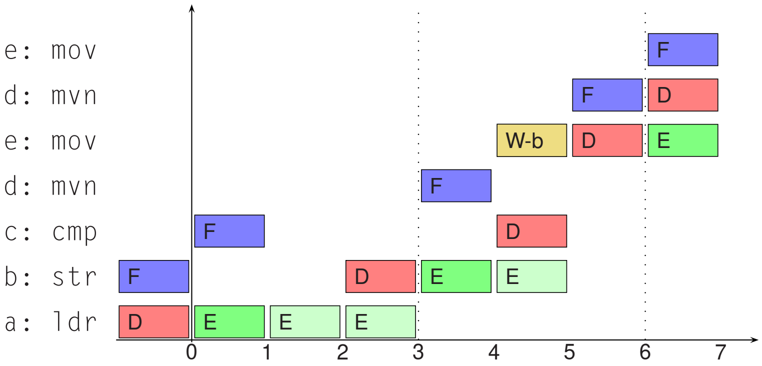
State \ Cycle	0	1	2	3	4	5	6	7
pipea,pipeaval	b,T	c,T	c,T	c,T	d,T	d,T	d,T	e,T
pipeb,pipebval	b,T	b,T	b,T	c,T	c,T	e,T	d,T	e,T
ireg,iregval	a,T	a,T	a,T	b,T	b,T	c,F	e,T	d,T
ointstart	F	F	F	F	F	F	F	F
onewinst	T	F	F	T	F	T	T	T
opipebll	T	F	F	T	F	T	T	T
nxtic	ldr	ldr	ldr	str	str	data_proc	data_proc	data_proc
nxtis	t3	t4	t5	t3	t4	t3	t3	t3

(b) With data forwarding.

Table 11: The pipeline behaviour for Example 4.



(a) Actual ARM6 behaviour.



(b) With data forwarding.

Figure 7: Pipeline flow for Example 4.

Function	Description
`ABORTINST`	Indicates whether the instruction in execute has failed its condition code or has been invalidated.
`IC`	Gives the instruction class for the current instruction cycle.
`INTSEQ`	Indicates whether an exception vector is to be taken on the next instruction cycle.
`IREGVAL`	Indicates whether the next execute stage has been invalidated by a direct change to r15.
`IS`	Gives the instruction sequence for the current instruction cycle.
`NEWINST`	Indicates whether a new instruction is to be executed.
`NXTIC`	Gives the instruction class for the next instruction cycle. (Provided the condition is passed.)
`NXTIS`	Gives the instruction sequence for the next instruction cycle.
`PCCHANGE`	Indicates whether r15 is changed (other than by a branch or exception vector) in the current instruction cycle.
`PIPEALL`	Indicates whether the result of instruction fetch is to be latched to PIPEA.
`PIPEAVAL`	Indicates whether instruction fetch has been invalidated by a direct change to r15.
`PIPEAWRITE`	Controls the PIPEA latch update.
`PIPEBLL`	Indicates whether the instruction in predecode is to be preseved in the current instruction cycle.
`PIPEBWRITE`	Controls the PIPEB latch update.
`PIPECWRITE`	Controls the PIPEC latch update.
`PIPESTATAWRITE`	Controls the PIPEAVAL latch update.
`PIPESTATBWRITE`	Controls the PIPEBVAL latch update.
`PIPESTATIREGWRITE`	Controls the IREGVAL latch update.

Table 12: Functions used to specify the pipeline control.

specification is far less abstract but one of the objectives of his work was to provide formal, high fidelity models.

6.3 The ARM6 State Function

The ARM6 state space is modelled in HOL with the following type declaration:

```
Hol_datatype ‘state_ARM6 = ARM6 of mem=>dp=>ctrl‘
```

where `mem` is a type abbreviation for `word30`→`word32`. The ARM6 is modelled with the function `STATE_ARM6`, which is defined by:

$$\vdash_{def} \texttt{STATE\_ARM6 0 a = INIT\_ARM6 a} \wedge$$
$$\texttt{STATE\_ARM6 (SUC t) a = NEXT\_ARM6 (STATE\_ARM6 t a)}$$

The functions `INIT_ARM6` and `NEXT_ARM6` are specified in Appendix C, and they are discussed in the following sections.

6.3.1 The Initialisation Function

The initialisation function is critical to the verification of a design when using the one-step theorems of Section 3.2. If the initialisation function is suitably defined then time-consistency (with respect to an immersion) will hold. The process of defining the initialisation function can be likened to developing an invariant for an inductive proof. Not all implementation states are valid and the primary job of the initialisation function is to ensure that the implementation starts in a suitable state. There must be at least one initial state for each state at the ISA level (this is the `DATA_ABSTRACTION` condition). Non-initial states will either be completely erroneous (the processor should never exhibit such states) or states that the machine passes through in the course of instruction execution. Examples of these two cases are:

- Erroneous: any state with `nxtic = data_proc`, `iregval = T`, `onewinst = F` and `nxtis = t5`.
- Intermediate: a state with `nxtic = reg_shift`, `iregval = T`, `onewinst = F` and `nxtis = t4` and the rest of the state corresponding with one cycle through the execution of a `reg_shift` instruction.

The first case is erroneous because it is inconsistent with control logic i.e. the next instruction sequence is never `t5` for a data processing instruction. The second case *could* be considered initial (with the immersion defined accordingly) but to do so would add unnecessary complexity i.e. the initialisation function would not have a simple definition and more cases would have to be considered in the verification. The initialisation function forces the processor to be in a state for which `nxtis = t3` and the pipeline is set so as to be consistent with the program counter value. In this way, the initial states are those at the beginning of the blocks in Tables 8-11. It is the job of the duration map to define how many cycles there are between two initial states. This corresponds with the number of cycles taken at the execute stage, which is a function of the instruction register.

The function `INIT_ARM6` is the same for the no-clobber and data forwarding implementations. After initialisation, the instruction about to executed is in the `ireg` register and this is from memory address `pc-0x8`. The rest of the state is consistent with having fetched and decoded this instruction. The instruction at address `pc-0x4` is fetched and the next instruction to be fetched is from address `pc`.

6.3.2 The Next State Function

The next state function `NEXT_ARM6` is specified in Appendix C. Two versions are presented and these correspond with Options 2 and 3 from page 22. The next state function is decomposed into two functions: `PHASE1` and `PHASE2`. By the end of the first phase the ALU registers (`alua` and `alub`) have the right values. The field extractor/extender and shifter have both been utilized by this point. In the second phase the ALU becomes active and results are stored as appropriate. Instruction decode also occurs in the second phase.

The next state function brings together all of the control logic and data path operations defined in Sections 6.1 and 6.2.

7 The ARM6 Verification

In Section 7.1 the correctness of the ARM6 is formally stated in terms of a data abstraction (Section 7.1.1) and temporal abstraction (Section 7.1.2). The formal verification of this correctness statement is discussed in Section 7.2.

7.1 Correctness Statement

The following correctness statement has been verified:

```
⊢ CORRECT STATE_ARM STATE_ARM6 IMM_ARM6 ABS_ARM6
```

The immersion `IMM_ARM6` and data abstraction `ABS_ARM6` are defined in Appendix D. The ISA state function `STATE_ARM` was originally defined in [13] but modifications have been made since, as outlined in Section 5. This correctness statement has been verified for two pairs of state functions i.e. for the no-clobber and data forwarding implementations. The no-clobber version requires the ISA specification to be modified so as to avoid writing over the pipeline state; see Section 6.2.2.

7.1.1 Data Abstraction

The data abstraction `ABS_ARM6` is specified in Appendix D. The map is essentially a projection, extracting out the memory `mem`, the general purpose registers `reg` and program status registers `psr`. The later two are taken from the data path unit. However, the program counter is decremented by eight, and this adjusts for the pipeline. That is to say, the ARM6 is about to execute the instruction from address `pc-0x8` and not that from address `pc`. The ISA model executes the instruction from address `pc` and adds eight to this value (as appropriate) when executing an instruction. This (offset by eight) relationship between the two program counters must be preserved in order for correctness to hold.

An alternative approach would have been to mimic the MA program counter behaviour at the ISA level. This is illustrated in the following table:

	ISA model	MA model
Execute instruction	`pc`	`pc-8`
Reading during execution	`pc+8`	`pc`
	`pc+12`	`pc+4`
Writing during execution	`pc`	`pc+8`
Data abstraction	`pc↦pc-8`	`pc↦pc`

The data abstraction is a pure projection if one adopts the MA behaviour at the ISA level. However, very little is gained by this approach (at the ISA level one must modify `pc` write instead of `pc` read) and it runs contrary to the spirit of the ARM architectural reference.

The data abstraction is surjective (and the initialisation function never modifies `mem`, `reg` or `psr`) and therefore it is possible to verify the `DATA_ABSTRACTION` condition: see Section 7.2.2. This mean that the ARM6 implements all of the ISA behaviour.

7.1.2 Temporal Abstraction

The immersion `IMM_ARM6` is specified in Appendix D. This function is defined so as to be uniform with respect to `STATE_ARM6` using a duration map `DUR_ARM6`. The duration map gives a value from one to six cycles. The data forwarding version has a slightly more complicated duration map—an extra cycle is needed for `str` instructions that write to `pipeb`. In order to detect such cases the ISA functions `DECODE_LDR_STR` and `ADDR_MODE2` are used to determine the destination address for the store.

7.2 Formal Verification

The formal verification is structured around proving the following results:

1. The immersion `IMM_ARM6` is uniform with respect to `STATE_ARM6`.
2. The data abstraction `ABS_ARM6` meets the `DATA_ABSTRACTION` condition.
3. The state function `STATE_ARM` is time-consistent, and the state function `STATE_ARM6` is time-consistent with respect to `IMM_ARM6`.
4. The state function `STATE_ARM6` is correct with respect to `STATE_ARM`, `IMM_ARM6` and `ABS_ARM6`.

These goals are discussed in the following sections.

7.2.1 Uniformity

The duration map `DUR_ARM6` is used as the witness in proving

```
⊢ UNIFORM IMM_ARM6 STATE_ARM6
```

It is necessary to prove

```
⊢ ∀a. 0 < DUR_ARM6 a
```

and this can be verified in HOL using the simplification tactic `RW_TAC`.

7.2.2 Completeness

For each ISA state `ARM mem reg psr` the processor state

```
ARM mem (DP (ADD8_PC reg) psr areg din alua alub) ctrl
```

is a suitable witness for the `DATA_ABSTRACTION` property. Free variables are used for components other than `mem`, `reg` and `psr`. The function `ADD8_PC` is defined by:

```
⊢def ADD8_PC reg = SUBST reg (r15,reg r15 + 0x8)
```

This function is the inverse of `SUB8_PC`:

```
⊢ ∀r. SUB8_PC (ADD8_PC r) = r
```

This theorem was not verifiable when using the old 32-bit type definition; see Section 5.2. This meant that it was impossible to prove the data abstraction condition (because it was false) without resorting to modifying the definitions of `SUB8_PC` and `ADD8_PC`. The development of `word32Theory` has been of great use in this regard.

7.2.3 Time-Consistency

The state function `STATE_ARM` does not have an initialisation function and therefore the following theorem can be used to derive time-consistency:

```
⊢ ∀f next. IMAP f I next ⇒ TCON f
```

where `I` is the identity map. This theorem states that any iterated map whose initialisation function is the identity map is time-consistent.

Establishing the time-consistency of `STATE_ARM6` with respect to `IMM_ARM6` is more involved. The theorem `TC_IMMERSION_ONE_STEP_THEOREM` is used to derive the following equivalence:

```
⊢ TCON_IMMERSION STATE_ARM6 IMM_ARM6 =
       (∀a. STATE_ARM6 0 (STATE_ARM6 (IMM_ARM6 a 0) a) = STATE_ARM6 (IMM_ARM6 a 0) a) ∧
        ∀a. STATE_ARM6 0 (STATE_ARM6 (IMM_ARM6 a 1) a) = STATE_ARM6 (IMM_ARM6 a 1) a
```

Thus, time-consistency can be verified by term rewriting (with appropriate case splitting) using the definitions of `STATE_ARM6` and `IMM_ARM6`, which are in turn defined using `NEXT_ARM6`, `INIT_ARM6` and `DUR_ARM6`. The cycle zero case can be discharged quite simply: the initialisation map `INIT_ARM6` is applied twice on the left hand side and once on the right hand side. The results are equal because the three ISA components do not change during initialisation. The cycle one case requires the next state function to be applied between one to six times. This is because

IMM_ARM6_ONE

```
⊢ ∀a. IMM_ARM6 a 1 = DUR_ARM6 (INIT_ARM6 a)
```

Therefore, the number of cases is primarily dictated by the definition of `DUR_ARM6`, and these cases are shown in Figure 8. The main branches correspond with the instruction classes and whether a branch (*pc*-write) occurs. The function `NEXT_ARM6` is complex and naïve multi-cycle expansion of the time-consistency goal is impracticable. A number of steps are taken to structure the proof and limit the amount of rewriting, and these are discussed below. HOL is an interactive proof tool and the *goalstack* was used to manage the proof effort.

The overall structure of the (cycle one) proof is as follows:

1. The goal is re-arranged to be of the form:

   ```
   ⊢ ∀b. (STATE_ARM6 (DUR_ARM6 (INIT_ARM6 b)) b = a) ⇒ (INIT_ARM6 a = a)
   ```

   This prevents the state from being evaluated twice i.e. on both sides of the equality. Note that the theorem `IMM_ARM6_ONE` is also applied.

2. The main cases are introduced manually using the `Cases_on` tactic. Each case provides a context (for the evaluation) in the form of a list of assumptions.

3. The bulk of the evaluation is carried out using the *simplifier* tactic `ASM_SIMP_TAC`. The simplification set (*simpset*) used contains the conversion `CBV_CONV`, which is in turn given a computation set (*compset*) containing definitions from the specification and this then carries out call-by-value conversion. This provides fast, symbolic execution of the HOL specification.

4. The previous stage generates a set of *leaf* goals; these are re-arranged (to undo the first stage) and the initialisation function is applied. Thus, a set of goals of the form $t_1 = t_2$ are produced, where t_1 and t_2 are terms denoting state classes. Each state class is expressed in terms of primitive operations from the specification i.e. the 32-bit word and register/memory access operations. These remaining goals is discharged using the tactic `RW_TAC`, which is supplied with a set of lemmas. This tactic automatically carries out further case splitting as required.

When evaluating each case (Stage 3), a subset of the definitions from the specification are added to the *compset*. The definitions added are predominantly those listed in Table 12 because these control the flow of instructions through the pipeline. The conversion `CBV_CONV` is fairly crude as a rewriting engine—it is not really designed as a general purpose rewriting tool and is best suited to evaluating ground terms. Therefore, the execution is carried out in single cycle stages so as to periodically utilize the more general tactic `ASM_SIMP_TAC`, which will rewrite using the assumption list (context). Trade-offs arise in selecting the set of definitions to rewrite with—a certain amount of experimentation occurred at this point in the proof effort. The aim is to limit the size (and increase readability—for the interactive proof) of terms, and yet at the same time to avoid carrying out superfluous rewriting. Only certain parts of the state space are of real relevance when determining time-consistency. The components of most significance are the memory and the program counter because these determine the state of the pipeline during initialisation. The rewriting is, therefore, structured around evaluating these components.

In order to speed up the evaluation and increase readability, the tactic `ABBREV_TAC` is used: see [18]. This tactic replaces all instances of a sub-term in a goal with a variable and an assumption is added asserting the equality of this variable with the sub-term. For example, the program counter and instruction code are represented by the variables `pc` and `i`. This is achieved with

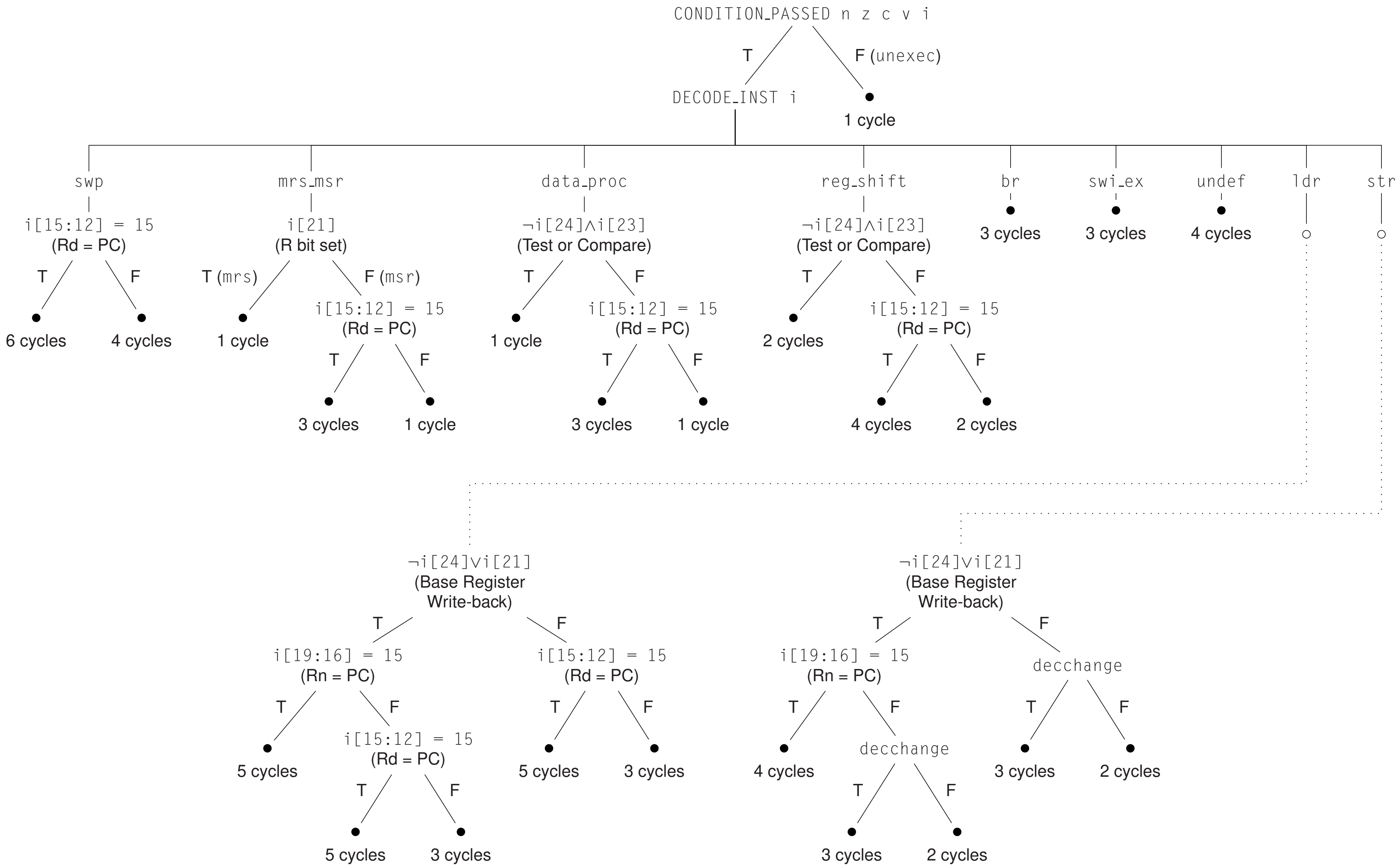


Figure 8: Manual case splits for the data forwarding ARM6. Without data forwarding the `str` instruction class does not case split on the `decchange`; the two cycle case prevails.

```
ABBREV_TAC ‘pc = REG_READ6 reg usr 15‘
```

and

```
ABBREV_TAC ‘i = MEMREAD mem (pc - w32 8)‘
```

For multi-cycle instruction classes, the sub-term representing the ALU output is abbreviated at the end of each cycle. This greatly reduces term sizes and increases evaluation times: the ALU sub-term is moderately large and occurs repeatedly because the ALU output propagates through the data path and control unit. The precise ALU output is of no significance with regards to time-consistency, and this means that the assumptions for these abbreviations can be discarded eventually.

The lemmas needed at Stage 4 only become apparent having completed Stage 3. It is at this point that large, unreadable terms can be a problem; for these sub-goals must be manually inspected and either: (*i*) a bug is spotted or (*ii*) a suitable lemma is constructed. The majority of lemmas that must be proved relate to the semantics of memory/register read and write operations. These operations are based on using the map substitution `SUBST`, which is defined by

```
⊢def SUBST m (a,w) b = (if a = b then w else m b)
```

For example, consider the following theorem:

MEM_WRITE_READ

```
⊢ ∀m d a1 a2 b.
    ¬ALIGN_EQ a1 a2 ⇒ (MEMREAD (MEM_WRITE b m d a1) a2 = MEMREAD m a2)
```

This theorem states that if two 32-bit word addresses, `a1` and `a2`, are not equal after word alignment (i.e. when ignoring the two least significant bits) then a write to address `a1` may be ignored when reading from address `a2`. In principle this is a simple theorem to prove, however, `a1` and `a2` are 32-bit words and the memory is indexed by 30-bit words. Consequently, one must reason about the equality of bit sub-strings. Also, the memory write covers byte and word storage.

There is a similar write-read theorem for registers:

REG_WRITE_READ

```
⊢ ∀r m m2 p n n2.
    n < 16 ∧ n2 < 16 ∧ ¬(n = n2) ⇒
      (REG_READ6 (REG_WRITE r m n d) m2 n2 = REG_READ6 r m2 n2)
```

Differing numbers (less than sixteen) access different registers, but it is not the case that a mode and number pair uniquely identifies a register. This is because the same register can be accessed in different modes. For example, the program counter is mode invariant. This theorem cannot be generalised for all values of `n` and `n2` because `REG_READ6` and `REG_WRITE` have a peculiar semantics, which is based on the assumption that out-of-bounds indices do not occur. For example:

```
⊢ ∀r. REG_READ6 r usr 16 = REG_READ6 r fiq 8
```

Once `n` is greater than 31 the register accessed by `REG_READ6` is effectively arbitrary. Indices are guaranteed to be in range by virtue of the fact that in use the index is either 14, 15 or a 4-bit field of the instruction code.

The theorems `MEM_WRITE_READ` and `REG_WRITE_READ` are used to establish that the pipeline state is consistent after re-initialisation. For example, if a branch or *pc*-write does not occur then `ireg` takes the value of `pipeb` as the pipeline shifts along a stage. Thus, ensuring that `ireg` is consistent involves proving that

```
⊢ MEMREAD m (REG_READ6 r usr 15 - 0x4) = MEMREAD m' (REG_READ6 r' usr 15 - 0x8)
```

where `m'` and `r'` are the states of the memory and register bank after instruction execution. The right-hand side represents the `ireg` state after re-initialisation and the left-hand side represents the old value of `pipeb` (the new `ireg` value). The terms are equivalent provided that reading the program counter from register bank `r'` gives an incremented program counter value (i.e. `REG_READ6 r usr 15 + 0x4`) and the memory `m'` is not altered at this new `pc` location. This is where the two theorems above come into play. Note that it is again necessary to be able to reason about word addition and subtraction. Similar goals have to be verified for the cases in which branches occur.

In essence, time-consistency involves verifying that the initialisation function `INIT_ARM6` acts as an invariant, and this entails reasoning about the control logic, in particular the pipeline control logic.

7.2.4 Correctness

The proof of correctness is very similar in structure to the proof of time-consistency. Correctness is reduced to the following goal using the one-step theorem:

```
⊢ (∀a. STATE_ARM 0 (ABS_ARM6 a) = ABS_ARM6 (STATE_ARM6 (IMM_ARM6 a 0) a)) ∧
   ∀a. STATE_ARM 1 (ABS_ARM6 a) = ABS_ARM6 (STATE_ARM6 (IMM_ARM6 a 1) a)
```

The cycle zero case is trivial to prove. The main cases for cycle one are those of Figure 8 but a few more cases are introduced at the leaves. Stage 1 of the time-consistency proof (re-arranging the goal) is not necessary because sub-terms are not repeated in the left and right hand sides of the correctness goal. The leaf goals are ISA state equivalences, with the primitive operations being word operations and read and write operations performed on the three state components: memory, registers and the program status registers. More definitions must be added to the evaluation *compset* because is now necessary to consider data aspects of the design i.e. the shifter and ALU units must be evaluated. Additional lemmas are needed to discharge the leaf goals. The state components are effectively reduced into a canonical form in order to establish their equivalence. For example, with registers the following two lemmas are used:

REG_WRITE_WRITE and REG_WRITE_COMMUTES

```
⊢ ∀r m n d d2. REG_WRITE (REG_WRITE r m n d) m n d2 = REG_WRITE r m n d2
⊢ ∀r m1 m2 n n2 d d2.
    n < 16 ∧ n2 < 16 ∧ ¬(n = n2) ⇒
      (REG_WRITE (REG_WRITE r m1 n2 d2) m2 n d = REG_WRITE (REG_WRITE r m2 n d) m1 n2 d2)
```

The second of these two lemmas in used with `n2` specialised to fifteen; this enables multiple program counter updates to be reduced to a single write. A series of lemmas are needed to equate operations performed on the program counter. For example:

```
⊢ ∀r m d. INC_PC (SUB8_PC (REG_WRITE r m 15 d)) = SUB8_PC (REG_WRITE r m 15 (d + 0x4))
```

Again, word arithmetic is needed in order to verify this theorem.

Various other lemmas were required to deal with some of the data manipulations performed by the ARM6. For example, the following lemma was needed to verify the load and swap instructions:

```
⊢ ∀h l a. w32 (SLICEw h l a) #>> l = w32 (BITSw h l a)
```

This theorem is needed to reason about reading a byte from memory. In the ISA specification the required byte is simply selected using the `BITSw` function, whereas in the ARM6 the field-extractor selects the required byte (keeping it in place) using `SLICEw` and this word is then rotated using the shifter. The operations `BITSw`, `SLICEw` and `#>>` (rotate right) are defined in terms of natural number division, mod, and addition. In practice, verifying results about these operation is often tedious to do in HOL. Having a well developed and extensive library of word theorems is clearly of great benefit here. Of course, there is always a temptation to cheat when such theorems are required. The ISA specification could have been modified (in a number of instances) to correspond with MA functionality. With the theorem above, logical shift right could have been used instead of rotate right: this would give the same result and be easier to verify in HOL. However, this path was resisted: the ISA remains abstract and the MA is true to the implementation. Verifying tricky word theorems is an important part of the formal verification. These proof obligations do not arise, as one might expect, through verifying the data aspects of the ISA (for example, when looking at the result of a data processing instruction) because here the operations performed are identical at both levels of abstraction. Instead, they are a consequence of relating the disparate ways in which the two level of abstraction (ISA and MA) make use of data operations in the course of instruction execution. At the ISA level succinctness and a clean presentation is important, whereas at the MA level physical organisation comes into play.

8 Summary and Future Work

The ARM6 micro-architecture has been modelled and formally verified in HOL. The processor's organisation is relatively simple, consisting of a three stage pipeline with multi-cycle execution. Nevertheless, formal verification is not trivial and subtle correctness issues must be considered. In particular, it remains to find the best methods to deal with (incorporate into the algebraic framework) implementation dependent behaviour i.e. the instruction flow (self-modifying code), exceptions and unpredictable instructions. It should not be necessary to modify the ISA specification or the MA implementation in a manner inconsistent with the spirit of the ARM references. Rather the correctness model should be adapted to reflect fully what is meant by correctness in the context of processor dependent behaviour. It has been demonstrated that the abstract definition of correctness presented in Section 3 is extensively suitable for ARM6 correctness. Furthermore, verification with the one-step theorems was shown to be a good approach: the initialisation function has a straightforward definition and it was not necessary to explicitly use an invariant.

It is clear that carrying out complete verifications of commercial micro-processors requires well developed tools and a good methodology. The HOL theorem prover enables one to:

- Formalise the abstract correctness model and one-step theorems.
- Model the ISA and MA in a concise manner.
- Carry out fast execution (for testing and verification) using `CBV_CONV`.
- Reason about word arithmetic and logic.
- Provide well structured proofs with the use of lemmas.

One of the disadvantages of using HOL is that the proof is not fully automated. Human interaction and expertise is required for the verification: one must manually direct the proof effort, spot bugs and supply necessary lemmas. However, it is worth noting that the structure of the main proofs are simple (case splitting, evaluation and application of lemmas) and the lemmas to be proved are not mathematically deep (equating memory substitutions). Therefore, for a given architecture, a library of theorems can be developed, which gives great potential for re-use and semi-automation. The structure of the proof varies little with similar designs. For example, the data forwarding version of the design was verified relatively quickly after completing the no-clobber version.

It is also of great advantage to be able to formalise and reason about correctness in an abstract setting. In this way, different implementations can be related by virtue of conforming with the same correctness condition. There is, therefore, no ambiguity as to precisely what is meant by correctness from one verification to the next. One correctness condition is not sufficient for all circumstances (for example, variants arise when one considers input, output, superscalar designs, error states and non-determinism), but it is clearly advantageous to develop a small family of correctness statements designed for microprocessor verification. It could be argued that the modifications made to the ARM6 implementations (the no-clobber and data forwarding versions) are an artifact of an over prescriptive correctness definition. Equally, one could also take the line that ARM should have taken a more purist and formal line with the ARM architecture i.e. stipulate that the program counter should always correspond with the address (in memory) of the instruction to be executed (and this relationship should be maintained throughout the course of instruction execution). The actual ARM6 violates this condition in two ways: (*i*) the instruction being executed is from the address `pc-0x8`; and (*ii*) it can execute an instruction that has been over-written in memory. In other words, ideally the ISA should be wholly implementation (pipeline) independent and each processor should be designed accordingly. However, it should be possible to modify our definition of correctness to more closely correspond with ARM's approach.

The formal verification brought about proof obligations in which the semantics of word operations was of importance. In particular, it was necessary to reason about 32-bit addition, subtraction and bit rotation, as well as bit field manipulations (extraction and concatenation). A HOL theory of bits and words was developed during the course of the verification and this theory should facilitate further specification and verification work (even outside the domain of processor verification). The word theory was developed with the ARM in mind, and so there is scope for extending the library with additional theorems and operations.

The verification was carried out on a modest desktop PC (800MHz) and processing power was not found to be a *major* problem. During testing the ISA could be executed at a rate of one or two instructions per second, and the MA took up to ten seconds per instruction. This is adequate for small scale testing and evaluation. The computation time for the verification is dominated by the time-consistency and correctness proofs; these involved approximately two and three million primitive HOL inferences respectively. The total CPU time for these two proofs was approximately fourteen minutes. However, using `CBV_CONV` in a proof is not as straightforward as it might be and a bespoke tactic for fast contextual evaluation would be beneficial. Were further instruction classes to be included (for example, the multiple data transfers) then the existing proof of correctness (for the instructions covered in this report) would be slowed due to the overhead of executing a more complex specification. The new instruction class would also have to be verified. The overall proof complexity is proportional to the number of instruction classes (because this primarily determines the number of cases) and so pipelined processor verifications of this nature appear to be quite tractable. The proof script was split into two main files: about 800 lines of lemmas and a thousand lines for the time-consistency and correctness proofs. It is important to find ways to ensure that the proof scripts are fairly small, presentable and hence manageable, otherwise it is all too easy to 'get lost' in the proof.

The ARM6 specification and verification took about six months (one person), with the time being split almost equally between specification and verification. The work on the theory of words represents almost half of the effort. The correctness problems with self-modifying code was identified prior to carrying out the verification. Two bugs were spotted in the course of the verification, but by happenstance they combined in a way that made the design correct—they simply represented an unrealistic implementation and this was rectified.

Future work will focus on the following:

- Changing the definition of correctness to encompass self-modifying code and more advanced memory models.
- Looking at ways to deal with unpredictable instructions.
- Incorporating external interrupts.
- Verification of the multiple data transfer instruction class.

Each of these will build upon the existing verification work. In the long term more advanced micro-architectures may be studied and there is potential to look at broader systems level correctness concerns, for example: bus protocols and multi-processor systems.

References

[1] Mark D. Aagaard, Byron Cook, Nancy A. Day, and Robert B. Jones. A framework for microprocessor correctness statements. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *Lecture Notes in Computer Science*, pages 433–448. Springer, 2001.

[2] Mark D. Aagaard and Miriam E. Leeser. Reasoning about pipelines with structural hazards. In Kumar and Kropf [25], pages 13–32.

[3] Graham Birtwistle and Brian Graham. Verifying SECD in HOL. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design*, pages 129–177, Amsterdam, 1990. IFIP WG 10.5, North-Holland.

[4] Graham Birtwistle and P. A. Subrahmanyam, editors. *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, 1988.

[5] Jerry R. Burch. Techniques for verifying superscalar microprocessors. In *33rd Design Automation Conference*, pages 552–557, 1996.

[6] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Proceedings of the 6th International Conference, CAV '94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80, Berlin, 1994. Springer-Verlag.

[7] Avra Cohn. A proof of correctness of the VIPER microprocessor: The first level. In Birtwistle and Subrahmanyam [4], pages 27–71.

[8] Avra Cohn. Correctness properties of the VIPER block model: The second level. In Graham Birtwistle and P. A. Subramanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 1–91. Springer-Verlag, 1989.

[9] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, June 1989.

[10] Werner Damm and Amir Pnueli. Verifying out-of-order executions. In Hon F. li and David K. Probst, editors, *Advances in Hardware Design and Verification: IFIP WG10.5 International Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 23–47, Montreal, Canada, 1997. Chapman & Hall.

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

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

[13] Anthony C. J. Fox. A HOL specification of the ARM instruction set architecture. Technical report, University of Cambridge, Computer Laboratory, June 2001.

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

[15] Brian T. Graham. *The SECD Microprocessor, A Verification Case Study*. Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Boston, 1992.

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

[17] Neal A. Harman and J. V. Tucker. Algebraic models and the correctness of microprocessors. In Milne and Pierre [29], pages 92–108.

[18] *The* HOL *System Reference for Kananaskis-1*, 2002.

[19] A. J. Hu and M. Y. Vardi, editors. *Computer Aided Verification (CAV 1998)*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[20] James K. Huggins and David Van Campenhout. Specification and verification of pipelining in the ARM2 RISC microprocessor. *ACM Transactions on Design Automation of Electronic Systems.*, 3(4):563–580, 1998.

[21] Warren A. Hunt, Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5:429–460, 1989.

[22] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor*, volume 795 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[23] Warren A. Hunt, Jr. and Bishop C. Brock. A formal HDL and its use in the FM9001 verification. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, pages 35–47. Prentice-Hall, 1992.

[24] Jeffrey J. Joyce. Formal verification and implementation of a microprocessor. In Birtwistle and Subrahmanyam [4], pages 129–157.

[25] Ramayya Kumar and Thomas Kropf, editors. *Proceedings of the 2nd International Conference, TPCD '94: Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume 901 of *Lecture Notes in Computer Science*. IFIP WG 10.2, Springer-Verlag, 1995.

[26] K. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In Hu and Vardi [19].

[27] Steven P. Miller and Mandayam K. Srivas. Formal verification of an avionics microprocessor. Technical Report SRI-CSL-95-4, Computer Science Laboratory, SRI International, Menlo Park, 1995.

[28] Steven P. Miller and Mandayam K. Srivas. Applying formal verification to the AAMP5 microprocessor: A case study in the industrial use of formal methods. *Formal Methods in Systems Design*, 8(2):153–188, March 1996.

[29] George J. Milne and Laurence Pierre, editors. *Correct Hardware Design and Verification Methods*, volume 683 of *Lecture Notes in Computer Science*. IFIP WG 10.2, Springer-Verlag, 1993.

[30] S. Tahar and R. Kumar. Formal Verification of Pipeline Conflicts in RISC-Processors. In *Proc. European Design Automation Conference (EURO-DAC94)*, pages 285–289, Grenoble, France, 1994. IEEE Computer Society Press.

[31] J. Sawada and W. Hunt. Processor verification with precise exceptions and speculative execution. In Hu and Vardi [19].

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

[33] Jens U. Skakkebaek, Robert B. Jones, and David L. Dill. Formal verification of out-of-order execution using incremental flushing. In *Computer Aided Verification*, pages 98–109, 1998.

[34] Sofiène Tahar and Ramayya Kumar. Implementing a methodology for formally verifying RISC processors in HOL. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 281–294. Springer-Verlag, 1993.

[35] Sofiéne Tahar and Ramayya Kumar. A practical methodology for the formal verification of RISC processors. *Formal Methods in System Design: An International Journal*, 13(2):159–225, September 1998.

[36] Phillip J. Windley. A theory of generic interpreters. In Milne and Pierre [29], pages 122–134.

[37] Phillip J. Windley and Jerry R. Burch. Mechanically checking a lemma used in an automatic verification tool. In Mandayam K. Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference, FMCAD '96: Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 362–376. Springer-Verlag, 1996.

[38] Phillip. J. Windley and Michael L. Coe. A correctness model for pipelined microprocessors. In Kumar and Kropf [25], pages 33–51.

A The Data Path Specification

A.1 The Memory Interface

```
⊢def AREG ic is ireg aregn inc reg15 aluout =
          if (is = t4) ∧ (ic = reg_shift) then
             if
               (¬BITw 24 ireg ∨ BITw 23 ireg) ∧ (BITSw 15 12 ireg = 15)
             then
               aluout
             else
               reg15
          else
             if (is = t4) ∧ ((ic = ldr) ∨ (ic = str)) then
                if
                  (¬BITw 24 ireg ∨ BITw 21 ireg) ∧ (BITSw 19 16 ireg = 15)
                then
                  aluout
                else
                  reg15
             else
                if (is = t5) ∧ (ic = ldr) ∨ (is = t6) ∧ (ic = swp) then
                  (if BITSw 15 12 ireg = 15 then aluout else reg15)
                else
                   if
                     (is = t3) ∧
                     ((ic = data_proc) ∧ (¬BITw 24 ireg ∨ BITw 23 ireg) ∧
                      (BITSw 15 12 ireg = 15) ∨
                      (ic = mrs_msr) ∧ ¬BITw 21 ireg ∧
                      (BITSw 15 12 ireg = 15) ∨ (ic = ldr) ∨ (ic = str) ∨
                      (ic = br)) ∨ (ic = swp)
                   then
                     aluout
                   else
                      if (is = t3) ∧ (ic = swi_ex) then
                        w32 (aregn * 4)
                      else
                        inc
⊢def AREGN1 intstart = if intstart then 1 else 2
⊢def DIN ic is ireg data =
         if ((ic = ldr) ∨ (ic = swp)) ∧ (is = t4) then data else ireg
⊢def DINWRITE ic is = ¬((ic = swp) ∧ (is = t5))
⊢def NBW ic is ireg =
         ¬(BITw 22 ireg ∧
           ((is = t3) ∧ ((ic = ldr) ∨ (ic = str) ∨ (ic = swp)) ∨
            (is = t4) ∧ (ic = swp)))
⊢def NRW ic is = (is = t3) ∧ (ic = str) ∨ (is = t4) ∧ (ic = swp)
```

Notes:

- See Section 6.1.2, page 13. These functions are listed in Table 3.
- `AREGN1` only has two cases: software interrupt or undefined instruction.
- `NBW` evaluates true for word access and false for byte access.
- `NRW` evaluates true for memory write and false for memory read.

A.2 The Field Extractor/Extender

```
⊢def FIELD ic is ireg oareg din =
       if is = t3 then
         if ic = br then
           SIGN_EX_OFFSET (BITSw 23 0 din)
         else
           if (ic = ldr) ∨ (ic = str) then
             w32 (BITSw 11 0 din)
           else
             if (ic = mrs_msr) ∨ (ic = data_proc) then
               w32 (BITSw 7 0 din)
             else
               ARB
       else
         if (is = t5) ∧ (ic = ldr) ∨ (is = t6) ∧ (ic = swp) then
           if ¬BITw 22 ireg then
             din
           else
             w32 (SLICEw (8 * oareg + 7) (8 * oareg) din)
         else
           ARB
```

Notes:

- See Section 6.1.3, page 15.
- The function `SIGN_EX_OFFSET` is defined in [13].
- `oareg` is required to be ≤ 3, and hence the application of slice selects a byte. This can implemented quite simply, without the need for the addition or multiplication.

A.3 The General Purpose Registers

```
⊢def PCWA ic is ireg =
        (is = t3) ∧
        (¬(ic = data_proc) ∨ (BITSw 24 23 ireg = 2) ∨
         ¬(BITSw 15 12 ireg = 15)) ∧
        (¬(ic = mrs_msr) ∨ BITw 21 ireg ∨ ¬(BITSw 15 12 ireg = 15)) ∧
        ¬(ic = undef) ∨ (ic = br) ∨ (ic = swi_ex)
⊢def RAA ic is ireg =
         if is = t3 then
            if (ic = data_proc) ∨ (ic = ldr) ∨ (ic = str) then
              BITSw 19 16 ireg
            else
               if ic = reg_shift then
                 BITSw 11 8 ireg
               else
                  if (ic = br) ∨ (ic = swi_ex) then 15 else ARB
         else
            if (is = t4) ∧ (ic = reg_shift) then
              BITSw 19 16 ireg
            else
              ARB
⊢def RBA ic is ireg =
         if
           (is = t3) ∧
           ((ic = data_proc) ∨ (ic = mrs_msr) ∨ (ic = ldr) ∨
            (ic = str)) ∨ (is = t4) ∧ (ic = reg_shift) ∨
           (is = t5) ∧ (ic = swp)
         then
           BITSw 3 0 ireg
         else
            if (is = t3) ∧ (ic = swp) then
              BITSw 19 16 ireg
            else
               if (is = t4) ∧ (ic = str) then
                 BITSw 15 12 ireg
               else
                  if (is = t5) ∧ ((ic = br) ∨ (ic = swi_ex)) then
                    14
                  else
                    ARB
⊢def RWA ic is ireg =
         if
           ((is = t3) ∧ (ic = data_proc) ∨
            (is = t4) ∧ (ic = reg_shift)) ∧
           (¬BITw 24 ireg ∨ BITw 23 ireg) ∨
           (is = t3) ∧ (ic = mrs_msr) ∧ ¬BITw 21 ireg ∨
           (is = t5) ∧ (ic = ldr) ∨ (is = t6) ∧ (ic = swp)
         then
           (T,BITSw 15 12 ireg)
         else
            if
              (is = t4) ∧ ((ic = ldr) ∨ (ic = str)) ∧
              (¬BITw 24 ireg ∨ BITw 21 ireg)
            then
              (T,BITSw 19 16 ireg)
            else
               if
                 ((is = t4) ∨ (is = t5)) ∧
                 ((ic = br) ∧ BITw 24 ireg ∨ (ic = swi_ex))
               then
                 (T,14)
               else
                 (F,ARB)
```

Notes:

- See Section 6.1.4, page 15. These functions are listed in Table 4.

- `PCWA` evaluates true when the program counter takes the value of the incremented address register.
- `RAA` and `RBA` give values in the range 0–15, or the value is unspecified (`ARB`).
- `RWA` gives a pair: the fist component is true when a register is to take the value of the ALU output; the second component is the index of the destination register, which is a number in the range 0–15.

A.4 The Program Status Registers

```
⊢def PSRA ic is ireg =
          (is = t3) ∧
          ((ic = swi_ex) ∨
           ¬BITw 22 ireg ∧
           ((ic = mrs_msr) ∨ (ic = data_proc) ∧ ¬BITw 20 ireg)) ∨
          (is = t4) ∧ ¬BITw 22 ireg ∧ (ic = reg_shift) ∧ ¬BITw 20 ireg
⊢def PSRDAT ic is ireg nbs aregn cpsrl psrfb alu sctlc =
          if
            BITw 20 ireg ∧
            ((is = t3) ∧ (ic = data_proc) ∨ (is = t4) ∧ (ic = reg_shift))
          then
            if BITSw 15 12 ireg = 15 then
              if USER nbs then cpsrl else psrfb
            else
              (let (n,z,c,v) = NZCV alu in
                 if
                   ¬BITw 23 ireg ∧ ¬BITw 22 ireg ∨
                   BITw 24 ireg ∧ BITw 23 ireg
                 then
                   SET_NZC n z sctlc cpsrl
                 else
                   SET_NZCV n z c v cpsrl)
          else
            if (is = t3) ∧ (ic = mrs_msr) then
              if USER nbs then
                if ¬BITw 22 ireg ∧ BITw 19 ireg then
                  w32 (SLICEw 31 28 (ALUOUT alu) + BITSw 27 0 psrfb)
                else
                  ARB
              else
                if BITw 19 ireg then
                  if BITw 16 ireg then
                    w32
                      (SLICEw 31 28 (ALUOUT alu) + SLICEw 27 8 psrfb +
                       BITSw 7 0 (ALUOUT alu))
                  else
                    w32 (SLICEw 31 28 (ALUOUT alu) + BITSw 27 0 psrfb)
                else
                  if BITw 16 ireg then
                    w32 (SLICEw 31 8 psrfb + BITSw 7 0 (ALUOUT alu))
                  else
                    ARB
            else
              if (is = t3) ∧ (ic = swi_ex) then
                SET_IFMODE T ((aregn = 0) ∨ (aregn = 7) ∨ BITw 6 cpsrl)
                  (exception2mode (num2exception aregn)) cpsrl
              else
                if (is = t4) ∧ (ic = swi_ex) then psrfb else ARB
⊢def PSRFBWRITE ic is = ¬((is = t4) ∧ (ic = swi_ex))
⊢def PSRWA ic is ireg nbs =
          if
            BITw 20 ireg ∧
            ((is = t3) ∧ (ic = data_proc) ∨
             (is = t4) ∧ (ic = reg_shift)) ∨ (is = t3) ∧ (ic = swi_ex)
          then
            (T,T)
          else
            if (is = t3) ∧ (ic = mrs_msr) then
              if
                ¬BITw 21 ireg ∨ ¬BITw 19 ireg ∧ ¬BITw 16 ireg ∨
                USER nbs ∧ (BITw 22 ireg ∨ ¬BITw 19 ireg ∧ BITw 16 ireg)
              then
                (F,ARB)
              else
                (T,¬BITw 22 ireg)
            else
              if (is = t4) ∧ (ic = swi_ex) then (T,F) else (F,ARB)
```

```
⊢def exception2mode e =
       case e of
         reset ↦ svc
       ‖ undefined ↦ und
       ‖ software ↦ svc
       ‖ address ↦ svc
       ‖ pabort ↦ abt
       ‖ dabort ↦ abt
       ‖ interrupt ↦ irq
       ‖ fast ↦ fiq
```

Notes:

- See Section 6.1.5, page 15. These functions are listed in Table 5.
- The functions `exception2mode` and `num2exception` are from the modified ISA specification: see Section 5.
- `PSRA` evaluates true if the CPSR is to be read and false if an SPSR is to be read.
- `nbs` is the current operating mode.
- The functions `SET_NZC`, `SET_NZCV` and `SET_IFMODE` are defined in [13].
- `alu` is a 5-tuple, and the functions `NZCV` and `ALUOUT` are used to extract the status flags and output word respectively.
- `PSRWA` gives a pair: the first component is true when a PSR is to be updated; the second component is true for CPSR update and false for SPSR update.

A.5 The Data Buses and Shifter

```
⊢def BUSA ic is psrrd ra =
        if is = t3 then
          if ic = mrs_msr then
            psrrd
          else
            if
              (ic = data_proc) ∨ (ic = ldr) ∨ (ic = str) ∨ (ic = br) ∨
              (ic = swi_ex)
            then
              ra
            else
              ARB
        else
          if (is = t4) ∧ (ic = reg_shift) then
            ra
          else
            if (is = t5) ∧ ((ic = br) ∨ (ic = swi_ex)) then
              0x3
            else
              ARB
⊢def BUSB ic is ireg din’ rb =
        if
          (is = t3) ∧
          ((ic = br) ∨
           BITw 25 ireg ∧ ((ic = data_proc) ∨ (ic = mrs_msr)) ∨
           ¬BITw 25 ireg ∧ ((ic = ldr) ∨ (ic = str))) ∨
          (is = t5) ∧ (ic = ldr) ∨ (is = t6) ∧ (ic = swp)
        then
          din’
        else
          rb
⊢def SCTRLREGWRITE ic is = (is = t3) ∧ (ic = reg_shift)
⊢def SHIFTER ic is ireg oareg sctrlreg busb c =
        if is = t3 then
          if BITw 25 ireg ∧ ((ic = data_proc) ∨ (ic = mrs_msr)) then
            ROR busb (2 * BITSw 11 8 ireg) c
          else
            if
              (ic = swp) ∨
              ¬BITw 25 ireg ∧ ((ic = ldr) ∨ (ic = str) ∨ (ic = mrs_msr))
            then
              LSL busb 0 c
            else
              if
                ¬BITw 25 ireg ∧ (ic = data_proc) ∨
                BITw 25 ireg ∧ ((ic = ldr) ∨ (ic = str))
              then
                SHIFT_IMMEDIATE2 (BITSw 11 7 ireg) (BITSw 6 5 ireg) busb c
              else
                if ic = br then LSL busb 2 c else ARB
        else
          if (is = t4) ∧ (ic = reg_shift) then
            SHIFT_REGISTER2 (BITSw 7 0 sctrlreg) (BITSw 6 5 ireg) busb c
          else
            if (is = t5) ∧ (ic = ldr) ∨ (is = t6) ∧ (ic = swp) then
              ROR busb (8 * oareg) c
            else
              if (is = t5) ∧ ((ic = br) ∨ (ic = swi_ex)) then
                LSL busb 0 c
              else
                ARB
```

Notes:

- See Section 6.1.6, page 15. These functions are listed in Table 6.
- `BUSA` gives value `0x3` for branches and exceptions, and this is used to compute the link register value (which is `NOT 0x3 + pc`).

- `SHIFTER` gives a pair: the first component is the carry out; the second component is the output word.
- The functions `LSL`, `ROR`, `SHIFT_IMMEDIATE2` and `SHIFT_REGISTER2` are taken from the modified ISA specification.
- `LSL busb 0 c = (c,busb).`

A.6 The ALU

```
⊢def ALU6 ic is ireg alua alub c =
        if
          (ic = data_proc) ∧ (is = t3) ∨ (ic = reg_shift) ∧ (is = t4)
        then
          ALU (BITSw 24 21 ireg) alua alub c
        else
          if (ic = mrs_msr) ∧ (is = t3) then
            ALU_logic (if BITw 21 ireg then alub else alua)
          else
            if (ic = ldr) ∨ (ic = str) then
              if (is = t3) ∧ ¬BITw 24 ireg then
                ALU_logic alua
              else
                if (is = t3) ∨ (is = t4) then
                  if BITw 23 ireg then ADD alua alub F else SUB alua alub T
                else
                  if (is = t5) ∧ (ic = ldr) then ALU_logic alub else ARB
            else
              if (is = t3) ∧ (ic = br) then
                ADD alua alub F
              else
                if (ic = br) ∨ (ic = swi_ex) then
                  if is = t4 then
                    ALU_logic alua
                  else
                    if is = t5 then ADD (NOT alua) alub F else ARB
                else
                  if ic = swp then ALU_logic alub else ARB
⊢def ALUAWRITE ic is =
        ¬((is = t4) ∧
          ((ic = ldr) ∨ (ic = str) ∨ (ic = br) ∨ (ic = swi_ex)))
⊢def ALUBWRITE ic is =
        ¬((is = t4) ∧ ((ic = ldr) ∨ (ic = str) ∨ (ic = swp)))
```

Notes:

- See Section 6.1.7, page 16. These functions are listed in Table 7.
- `ALU6` gives a 5-tuple `(n,z,c,v,aluout)`.
- The functions `ALU`, `ALU_logic` and `ADD` are defined in [13].

B Pipeline Control Specification

```
⊢def ABORTINST iregval onewinst ointstart ireg n z c v =
        ¬iregval ∨
        onewinst ∧ ¬ointstart ∧
        ¬CONDITION_PASSED n z c v (BITSw 31 28 ireg)
⊢def ALIGN_EQ a b = (WORD_ALIGN a = WORD_ALIGN b)
⊢def IC abortinst nxtic = (if abortinst then unexec else nxtic)
⊢def INTSEQ ic is = (ic = undef) ∧ (is = t3)
⊢def IREGVAL pipecval pcchange = pipecval ∧ ¬pcchange
⊢def IS abortinst nxtis = (if abortinst then ARB else nxtis)
⊢def NEWINST ic is intstart =
        intstart ∨ (ic = data_proc) ∨ (ic = mrs_msr) ∨ (ic = unexec) ∨
        ((ic = str) ∨ (ic = reg_shift)) ∧ (is = t4) ∨
        ((ic = ldr) ∨ (ic = br) ∨ (ic = swi_ex)) ∧ (is = t5) ∨
        (ic = swp) ∧ (is = t6)
⊢def NXTIC intstart newinst ic ireg =
        if intstart then
          swi_ex
        else
          if ¬newinst then ic else DECODE_INST (w2n ireg)
⊢def NXTIS ic is newinst =
        if newinst then
          t3
        else
          if
            (is = t3) ∧
            ((ic = reg_shift) ∨ (ic = ldr) ∨ (ic = str) ∨ (ic = br) ∨
             (ic = swi_ex) ∨ (ic = swp))
          then
            t4
          else
            if
              (is = t4) ∧
              ((ic = ldr) ∨ (ic = br) ∨ (ic = swi_ex) ∨ (ic = swp))
            then
              t5
            else
              if (is = t5) ∧ (ic = swp) then t6 else ARB
⊢def PCCHANGE rwa = (let (w,a) = rwa in w ∧ (a = 15))
⊢def PIPEALL opipebll = opipebll
⊢def PIPEAVAL pcchange = ¬pcchange
⊢def PIPEAWRITE pipeall = pipeall
⊢def PIPEBLL newinst ic = newinst ∨ (ic = br) ∨ (ic = swi_ex)
⊢def PIPEBWRITE pipebll = pipebll
⊢def PIPECHANGE areg apipea apipeb =
        ALIGN_EQ areg apipea ∨ ALIGN_EQ areg apipeb
⊢def PIPECWRITE newinst = newinst
⊢def PIPESTATAWRITE pipeall pcchange = pipeall ∨ pcchange
⊢def PIPESTATBWRITE pipebll pcchange = pipebll ∨ pcchange
⊢def PIPESTATIREGWRITE newinst pcchange = newinst ∨ pcchange
```

Definitions for the data forwarding implementation:

```
⊢def IREGVAL pipecval pcchange decchange =
        pipecval ∧ ¬(pcchange ∨ decchange)
⊢def PIPEBWRITE pipebll pipebchange = pipebll ∧ ¬pipebchange
⊢def PIPECHANGE nrw pcchange areg pipe =
        nrw ∧ ¬pcchange ∧ ALIGN_EQ areg pipe
```

Notes:

- See Section 6.2.3, page 22. These functions are listed in Table 12.
- The functions `CONDITION_PASSED`, `DECODE_INST`, `WORD_ALIGN` and `SET_BYTE` are taken from the modified ISA specification.

C The Initialisation and Next State Functions

```
⊢def REG_READ6 reg mode n =
       if n = 15 then FETCH_PC reg else REG_READ reg mode n
```

```
⊢def INIT_ARM6
       (ARM6 mem (DP reg psr areg din alua alub)
          (CTRL pipea pipeaval pipeb pipebval ireg iregval apipea apipeb
             ointstart onewinst opipebll nxtic nxtis aregn nbw nrw
             sctrlreg psrfb oareg)) =
     (let pc = REG_READ6 reg usr 15 in
        ARM6 mem (DP reg psr pc (MEMREAD mem (pc - 0x8)) alua alub)
          (CTRL (MEMREAD mem (pc - 0x4)) T (MEMREAD mem (pc - 0x4)) T
             (MEMREAD mem (pc - 0x8)) T (pc - 0x4) (pc - 0x4) F T T
             (DECODE_INST (w2n (MEMREAD mem (pc - 0x8)))) t3 2 nbw F
             sctrlreg psrfb oareg))
```

```
⊢def NEXT_ARM6 = PHASE2 o PHASE1
```

Notes:

- See Section 6.3, page 26.
- The function `REG_READ6` is used to access registers. This is essentially the same as the ISA function `REG_READ` but it differs when reading the program counter. The ISA version adds eight to the program counter (to compensate for the pipeline), whereas at the MA level the program counter is naturally running ahead.
- The initialisation function `INIT_ARM6` preserves the ISA state components `mem`, `reg` and `psr`.
- The address register `areg` holds the `pc` value (this is the address of the next instruction to be fetched).
- Both the instruction register `ireg` and data-in register `din` holds the current instruction code (from address `pc-0x8`). This ensures that the field extractor/extender processes the instruction code.
- The pipeline contains the instructions from addresses `pc-0x8` and `pc-0x4` and these values are flagged as being valid.
- The next instruction sequence is `t3` and the next instruction class is the decoding of the instruction register.
- The `nrw` bit is clear because a memory read (instruction fetch) is going to occur. The `nbw` bit can have any value because it does not affect memory reads.
- The phase functions are defined in the following sections.

C.1 Version 1 (No-Clobber)

```
⊢def PHASE1 (ARM6 mem (DP reg psr areg din alua alub)
          (CTRL pipea pipeaval pipeb pipebval ireg iregval apipea apipeb
                ointstart onewinst opipebll nxtic nxtis aregn nbw nrw sctrlreg psrfb oareg)) =
    let cpsr = CPSR_READ psr
    in
    let (n,z,c,v,nbs) = DECODE_PSR cpsr
    in
    let abortinst = ABORTINST iregval onewinst ointstart ireg n z c v
    in
    let ic = IC abortinst nxtic
    and is = IS abortinst nxtis
    in
    let pcwa = PCWA ic is ireg
    and rwa = RWA ic is ireg
    in
    let intseq = INTSEQ ic is
    and pcchange = PCCHANGE rwa
    in
    let newinst = NEWINST ic is intseq
    in
    let pipeall = PIPEALL opipebll
    and pipebll = PIPEBLL newinst ic
    and pipec = if PIPECWRITE newinst then pipeb else ireg
    and pipecval = pipebval
    in
    let psrrd = if PSRA ic is ireg then cpsr else SPSR_READ psr nbs
    in
    let psrfb' = if PSRFBWRITE ic is then psrrd else psrfb
    in
    let raa = RAA ic is ireg
    and rba = RBA ic is ireg
    in
    let ra = REG_READ6 reg nbs raa
    and rb = REG_READ6 reg nbs rba
    and din' = FIELD ic is ireg oareg din
    in
    let busa = BUSA ic is psrrd ra
    and busb = BUSB ic is ireg din' rb
    and sctrlreg' = if SCTRLREGWRITE ic is then ra else sctrlreg
    in
    let shifter = SHIFTER ic is ireg oareg sctrlreg busb c
    in
    let shcout = FST shifter
    and shout = SND shifter
    in
    let alua' = if ALUAWRITE ic is then busa else alua
    and alub' = if ALUBWRITE ic is then shout else alub
    in
  (busb,c,shcout,rwa,cpsr,nbs,ic,is,pcwa,pcchange,pipeall,pipec,pipecval,
  ARM6 mem (DP reg psr areg din alua' alub')
    (CTRL pipea pipeaval pipeb pipebval ireg iregval apipea apipeb
       intseq newinst pipebll nxtic nxtis aregn nbw nrw sctrlreg'
       psrfb' oareg))
```

Notes:

- The functions `CPSR_READ`, `SPSR_READ` and `DECODE_PSR` are taken from the ISA specification.

```
⊢def PHASE2 (busb,c,shcout,rwa,cpsr,nbs,ic,is,pcwa,pcchange,pipeall,pipec,pipecval,
             ARM6 mem (DP reg psr areg din alua alub)
              (CTRL pipea pipeaval pipeb pipebval ireg iregval apipea
                 apipeb intseq newinst pipebll nxtic nxtis aregn nbw nrw
                 sctrlreg psrfb oareg)) =
    let alu = ALU6 ic is ireg alua alub c
    in
    let aluout = ALUOUT alu
    and inc = areg + 0x4
    and pcbus = REG_READ6 reg usr 15
    and psrwa = PSRWA ic is ireg nbs
    in
    let psrdat = PSRDAT ic is ireg nbs aregn cpsr psrfb alu shcout
    and data = if nrw then ARB else MEMREAD mem areg
    in
    let mem' = if nrw ∧ (pcchange ∨ ¬PIPECHANGE areg apipea apipeb)
                  then MEMWRITE mem busb areg nbw else mem
    and reg' = if pcwa then REG_WRITE reg nbs 15 inc else reg
    and psr' = if FST psrwa then
                  if SND psrwa then CPSR_WRITE psr psrdat else SPSR_WRITE psr nbs psrdat
               else psr
    in
    let reg'' = if FST rwa then REG_WRITE reg' nbs (SND rwa) aluout else reg'
    in
    let oareg' = BITSw 1 0 areg
    and areg' = AREG ic is ireg aregn inc pcbus aluout
    and pipea' = if PIPEAWRITE pipeall then data else pipea
    and apipea' = if PIPEAWRITE pipeall then areg else apipea
    in
    let pipeb' = if PIPEBWRITE pipebll then pipea' else pipeb
    and apipeb' = if PIPEBWRITE pipebll then apipea' else apipeb
    and pipeaval' = if PIPESTATAWRITE pipeall pcchange then PIPEAVAL pcchange else pipeaval
    in
    let pipebval' = if PIPESTATBWRITE pipebll pcchange then pipeaval' else pipebval
    and iregval' = if PIPESTATIREGWRITE newinst pcchange then IREGVAL pipecval pcchange
                                                       else iregval
    in
    let nxtic' = NXTIC intseq newinst ic pipec
    and nxtis' = NXTIS ic is newinst
    and din' = if DINWRITE ic is then DIN ic is pipec data else din
    and aregn' = AREGN1 intseq
    and nbw'  = NBW ic is ireg
    and nrw'  = NRW ic is
    in
   ARM6 mem' (DP reg'' psr' areg' din' alua alub)
     (CTRL pipea' pipeaval' pipeb' pipebval' pipec iregval' apipea'
        apipeb' intseq newinst pipebll nxtic' nxtis' aregn' nbw' nrw'
        sctrlreg psrfb oareg')
```

Notes:

- The functions `CPSR_WRITE`, `SPSR_WRITE` and `REG_WRITE` are taken from the ISA specification.
- In order to avoid writing over the pipeline, an extra condition is added to the memory write condition i.e. `pcchange ∨ ¬PIPECHANGE areg apipea apipeb`. Removing this condition would make the specification more faithful to the actual ARM6 behaviour.

C.2 Version 2 (Data Forwarding)

```
⊢def PHASE1 (ARM6 mem (DP reg psr areg din alua alub)
           (CTRL pipea pipeaval pipeb pipebval ireg iregval apipea apipeb
                 ointstart onewinst opipebll nxtic nxtis aregn nbw nrw sctrlreg psrfb oareg)) =
     let cpsr = CPSR_READ psr
     in
     let (n,z,c,v,nbs) = DECODE_PSR cpsr
     in
     let abortinst = ABORTINST iregval onewinst ointstart ireg n z c v
     in
     let ic = IC abortinst nxtic
     and is = IS abortinst nxtis
     in
     let pcwa = PCWA ic is ireg
     and rwa = RWA ic is ireg
     in
     let intseq = INTSEQ ic is
     and pcchange = PCCHANGE rwa
     in
     let pipeachange = PIPECHANGE nrw pcchange areg apipea
     and pipebchange = PIPECHANGE nrw pcchange areg apipeb
     and newinst = NEWINST ic is intseq
     in
     let pipecwrite = PIPECWRITE newinst
     in
     let pipeall = PIPEALL opipebll
     and pipebll = PIPEBLL newinst ic
     and pipec = if pipecwrite then pipeb else ireg
     and pipecval = pipebval
     and decchange = pipecwrite ∧ pipebchange
     in
     let psrrd = if PSRA ic is ireg then cpsr else SPSR_READ psr nbs
     in
     let psrfb' = if PSRFBWRITE ic is then psrrd else psrfb
     in
     let raa = RAA ic is ireg
     and rba = RBA ic is ireg
     in
     let ra = REG_READ6 reg nbs raa
     and rb = REG_READ6 reg nbs rba
     and din' = FIELD ic is ireg oareg din
     in
     let busa = BUSA ic is psrrd ra
     and busb = BUSB ic is ireg din' rb
     and sctrlreg' = if SCTRLREGWRITE ic is then ra else sctrlreg
     in
     let shifter = SHIFTER ic is ireg oareg sctrlreg busb c
     in
     let shcout = FST shifter
     and shout = SND shifter
     in
     let alua' = if ALUAWRITE ic is then busa else alua
     and alub' = if ALUBWRITE ic is then shout else alub
     in
   (busb,c,shcout,rwa,cpsr,nbs,ic,is,pcwa,pcchange,
    pipeachange,pipebchange,decchange,pipeall,pipec,pipecval,
    ARM6 mem (DP reg psr areg din alua' alub')
      (CTRL pipea pipeaval pipeb pipebval ireg iregval apipea apipeb
         intseq newinst pipebll nxtic nxtis aregn nbw nrw sctrlreg' psrfb' oareg))
```

Notes:

- This definition is similar to that of version 1. In order to implement the data forwarding, three additional values are computed: `pipeachange`, `pipebchange` and `decchange`. The first two are determined by comparing `apipea` and `apipeb` with `areg` (the address of the memory write). The value `decchange` indicates whether or not `pipeb` is going to be overwritten in parallel with a decode of the old `pipeb`.

```
⊢def PHASE2 (busb,c,shcout,rwa,cpsr,nbs,ic,is,pcwa,pcchange,pipeachange,pipebchange,decchange,
           pipeall,pipec,pipecval,ARM6 mem (DP reg psr areg din alua alub)
             (CTRL pipea pipeaval pipeb pipebval ireg iregval apipea apipeb
                intseq newinst pipebll nxtic nxtis aregn nbw nrw sctrlreg psrfb oareg)) =
    let alu = ALU6 ic is ireg alua alub c
    in
    let aluout = ALUOUT alu
    and inc = areg + 0x4
    and pcbus = REG_READ6 reg usr 15
    and psrwa = PSRWA ic is ireg nbs
    in
    let psrdat = PSRDAT ic is ireg nbs aregn cpsr psrfb alu shcout
    and data = if nrw then ARB else MEMREAD mem areg
    and oareg' = BITSw 1 0 areg
    in
    let areg' = if decchange then apipea else AREG ic is ireg aregn inc pcbus aluout
    and nxtic' = NXTIC intseq newinst ic pipec
    and nxtis' = NXTIS ic is newinst
    and din' = if DINWRITE ic is then DIN ic is pipec data else din
    and aregn' = AREGN1 intseq
    and nbw'  = NBW ic is ireg
    and nrw'  = NRW ic is
    in
    let mem' = if nrw then MEMWRITE mem busb areg nbw else mem
    and reg' = if decchange then REG_WRITE reg nbs 15 apipea
               else
                 if pcwa then REG_WRITE reg nbs 15 inc else reg
    and psr' = if FST psrwa then
                  if SND psrwa then CPSR_WRITE psr psrdat else SPSR_WRITE psr nbs psrdat
               else psr
    in
    let reg'' = if FST rwa then REG_WRITE reg' nbs (SND rwa) aluout else reg'
    in
    let pipea' = if PIPEAWRITE pipeall then data else
                   if pipeachange then
                     if nbw then busb else SET_BYTE oareg' busb pipea
                   else pipea
    and apipea' = if PIPEAWRITE pipeall then areg else apipea
    in
    let pipeb' = if PIPEBWRITE pipebll decchange then pipea' else
                   if pipebchange then
                     if nbw then busb else SET_BYTE oareg' busb pipeb
                   else pipeb
    and apipeb' = if PIPEBWRITE pipebll decchange then apipea' else apipeb
    and pipeaval' = if PIPESTATAWRITE pipeall pcchange then PIPEAVAL pcchange else pipeaval
    in
    let pipebval' = if PIPESTATBWRITE pipebll pcchange then pipeaval' else pipebval
    and iregval' = if PIPESTATIREGWRITE newinst pcchange then IREGVAL pipecval pcchange decchange
                                                         else iregval
    in
  ARM6 mem' (DP reg'' psr' areg' din' alua alub)
    (CTRL pipea' pipeaval' pipeb' pipebval' pipec iregval' apipea'
       apipeb' intseq newinst pipebll nxtic' nxtis' aregn' nbw' nrw'
       sctrlreg psrfb oareg')
```

Notes:

- The memory write is restored in accordance with the actual ARM6 behaviour i.e. the pipe change condition is dropped.
- The pipeline behaviour is now more complicated: the `busb` value (to be stored in memory) is conditionally forwarded to `pipea` or `pipeb`. The function `SET_BYTE` (from the ISA specification) is needed to cope with a byte store i.e. the least significant byte of `busb` overwrites a single byte in `pipea` or `pipeb`.
- If `decchange` is set then the `ireg` register is flagged as being invalid because `pipeb` needs to be decoded.

D The Data and Temporal Abstractions

```
⊢def SUB8_PC reg = SUBST reg (r15,reg r15 - 0x8)
⊢def ABS_ARM6 (ARM6 mem (DP reg psr areg din alua alub) ctrl) =
        ARM mem (SUB8_PC reg) psr
⊢def DUR_ARM6 (ARM6 mem (DP reg psr areg din alua alub)
              (CTRL pipea pipeaval pipeb pipebval ireg iregval apipea apipeb
                ointstart onewinst opipebll nxtic nxtis aregn nbw nrw sctrlreg psrfb oareg)) =
      let (n,z,c,v,nbs) = DECODE_PSR (CPSR_READ psr) in
      let abortinst = ABORTINST iregval onewinst ointstart ireg n z c v in
      let ic = IC abortinst nxtic in
      let pcchange = PCCHANGE (RWA ic t3 ireg)
      in
      if ic = undef then
        4
      else
        if (ic = mrs_msr) ∨ (ic = data_proc) then
          if pcchange then 3 else 1
        else
          if ic = reg_shift then
             if (BITSw 15 12 ireg = 15) ∧ (¬BITw 24 ireg ∨ BITw 23 ireg) then 4 else 2
          else
             if ic = ldr then
                if (BITSw 15 12 ireg = 15) ∨
                   (BITSw 19 16 ireg = 15) ∧ (¬BITw 24 ireg ∨ BITw 21 ireg) then 5 else 3
             else
                if ic = str then
                   if (BITSw 19 16 ireg = 15) ∧ (¬BITw 24 ireg ∨ BITw 21 ireg) then 4 else 2
                else
                   if (ic = br) ∨ (ic = swi_ex) then
                      3
                    else
                       if ic = swp then
                         if BITSw 15 12 ireg = 15 then 6 else 4
                       else
                         1
⊢def IMM_ARM6 a 0 = 0 ∧
     IMM_ARM6 a (SUC t) = DUR_ARM6 (STATE_ARM6 (IMM_ARM6 a t) a) + IMM_ARM6 a t
```

With the data forwarding implementation, the function `DUR_ARM6` is refined for the case of store instructions:

```
...
 if ic = str then
    if (BITSw 19 16 ireg = 15) ∧ (¬BITw 24 ireg ∨ BITw 21 ireg) then 4 else
       if (let (Im,P,U,B,W,L,Rn,Rd,offset) = DECODE_LDR_STR (w2n ireg) in
           ALIGN_EQ (FST (ADDR_MODE2 (SUB8_PC reg) nbs c Im P U Rn offset)) apipeb) then 3 else 2
 else ...
```

Notes:

- See Section 7.1, page 27.
- The function `SUB8_PC` subtracts eight from the program counter and is used in the definition of `ABS_ARM6`.
- The duration map `DUR_ARM6` gives a value based on the instruction class (`ic`) and whether or not the program counter is a destination register.
- The map `DUR_ARM6` is only ever applied to initial cycles i.e. for states part way through the execution of an instruction, it does not determine how many cycles remain.
- The immersion `IMM_ARM6` is constructed so as to be uniform with respect to `DUR_ARM6` and `STATE_ARM6`.