Verifying the ARM Block Data Transfer Instructions

Anthony Fox
Computer Laboratory, University of Cambridge

Abstract

The HOL-4 proof system has been used to formally verify the correctness of the ARM6 *micro-architecture*. This paper describes the specification and verification of one instructions class, *block data transfers*; these are a form of load-store instruction in which a set of up to sixteen registers can be transferred atomically. The ARM6 is a commercial RISC microprocessor that has been used extensively in embedded systems – it has a 3-stage pipeline with a multi-cycled execute stage. A list based *programmer's model* specification of the block data transfers is compared with the ARM6's implementation which uses a 16-bit mask. The models are far removed and reasonably complex, and this poses a verification challenge. This paper describes the approach and some key lemmas used in verifying correctness, which is defined using data and temporal abstraction maps.

1 Introduction

This paper presents a HOL specification of the ARM block data transfer instruction class [7, 18], together with a description of the ARM6 implementation and its formal verification using the HOL proof system. This work builds upon an ARM6 verification [6] which did not cover the block data transfer or multiply instruction classes.

The correctness model and underlying approach used for this work has been formalised in HOL [5]. This methodology was developed at Swansea and work has continued there using Maude [9, 10]. Using this approach, the correctness of the ARM6 implementation of the block data transfers has been formally verified. This is achieved by relating state machine models at the instruction set and micro-architecture levels of abstraction.

One source of difficultly in verifying this instruction class is the relatively complex nature of the implementation. The ARM6 has a 3-stage pipeline with fetch, decode and execute stages, and the execute stage can take a number of processor clock cycles to complete. With most instructions the number of cycles required is a small constant value. For example, an ordinary (single) load instruction takes three cycles, or five if the program counter is modified. The processor control logic makes use of a counter (the *instruction sequence*, `is`) to implement this behaviour. Typically this counter is incremented after each execute cycle and this provides a simple mechanism to symbolically execute an instruction to completion. However, with block data transfer instructions the counter takes and holds the value `tn` until a termination condition is met (this can take up to sixteen cycles). A 16-bit mask is used to keep track of which registers have been transfered and this forms the basis for the termination test. Consequently, symbolic execution for this instruction class is not straightforward.

From a correctness standpoint one must consider the case of writing to the memory at the address $pc + 8$ or $pc + 4$, where pc is the address of the instruction being executed. These addresses correspond with instructions that have been fetched and decoded respectively.[1] In order to provide a clean model, the ARM6 *should* detect when these instructions have been updated by a memory write and take steps to fetch and decode them again. However, the ARM6 does not waist costly control logic in dealing with this, instead it just carries on regardless. Before the block data transfers were verified two solutions to this problem were applied (for the verification of single word/byte data stores):

1. No-clobber method: a write to the addresses $pc + 8$ and $pc + 4$ is nullified at the programmer's model and micro-architecture levels.

[1] The architecture does not split the main memory into program and data parts. Memory is byte addressable and each 32-bit instruction occupies four bytes.

User	FIQ	IRQ	SVC	Abort	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12
r13	r13_fiq	r13_irq	r13_svc	r13_abt	r13_und
r14	r14_fiq	r14_irq	r14_svc	r14_abt	r14_und
r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_irq	SPSR_svc	SPSR_abt	SPSR_und

Figure 1: ARM's visible registers.

2. Data forwarding implementation: a write to these addresses is detected by the processor and the fetch and decode components of the state are updated as appropriate.

The second method has a clean programmer's model specification but it complicates, and does not accurately model, the micro-architecture. Both of these methods were manageable with `STR` instructions but they would have significantly added to the difficulty of verifying the block data transfers. Therefore, a third method has been adopted: the programmer's model is augmented with two registers forming a rudimentary pipeline. A further more abstract level of abstraction has been introduced to hide the pipeline i.e. the model accepts a stream of instructions, abstracting out details about instruction fetching.

1.1 Related Work

Early work on the mechanical verification of processors includes: TAMARACK [13], SECD [8], the partial verification of Viper [4], Hunt's FM8501 [11], and the generic interpreter approach of Windley [19]. Following this work, Miller and Srivas verified some of the instructions of a simple commercial processor called the AAMP5 [16]. Complex commercial designs have also been specified, simulated and verified using ACL2 [2, 14].

With the addition of complex multi-stage pipelines and out-of-order execution, contemporary commercial designs were considered too complex for *complete* formal verification. Recently progress has been made in verifying academic designs based around Tomasulo's algorithm [15, 12, 17, 1]. The instruction sets used for this work are often relatively simple (i.e. no block data transfers) with many based on the DLX architecture of Hennessy and Patterson. Most recent projects have used variants of the *flushing* correctness model of Burch and Dill [3]. We use a stronger notion of correctness.

2 The Instruction Set Architecture

For details of the ARM programmer's model the reader is referred to Furber [7] and the ARM Architecture Reference manual [18]. A limited précis is provided here.

The ARM architecture's visible state consists of a main memory and a set of 32-bit registers. The main memory is effectively an array of 2^{32} bytes. The registers form overlapping banks, as shown in Figure 1. Six *processor modes* provide support for exception handling and system-level programming. The general purpose registers are named `r0` to `r14`, the program counter is `r15`, and CPSR is the Current Program Status Register. When not in user mode the programmer also has access to a Saved Program Status Register (SPSR). The CPSR stores the current processor mode, together with four flags: `N` (negative), `Z` (zero), `C` (carry) and `V` (overflow). These flags are used to control program flow: all instructions are conditionally executed. For example, the instruction `STMHI` will be a no-op if `C` is clear or `Z` is set.

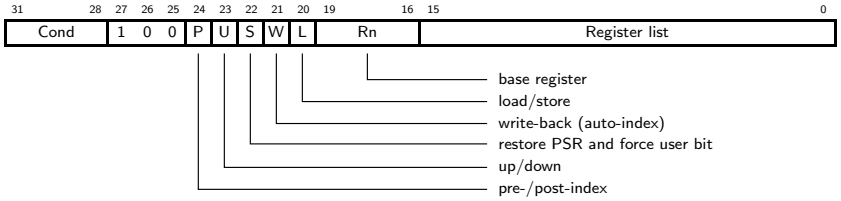

Figure 2: Encoding for the block data transfer instruction class.

2.1 Block Data Transfers

Block data transfer instructions load/store a set of general purpose register values from/to main memory; the instruction format is shown in Figure 2. These instructions are used for procedure entry and return (saving and restoring workspace registers), and in writing memory block copy routines.

The set of registers to be transferred is encoded using a 16-bit value; the program counter may be included in the list (bit fifteen). The memory block is determined by the base register `Rn`, and the bits `P` and `U`. The `W` flag enables base register write back (auto-indexing). There are also special forms of the instruction for accessing the user mode registers (when not in user mode) and for restoring the CPSR when returning from an exception – these options are controlled by the `S` flag and bit fifteen.

The instruction syntax is illustrated below:

```
LDM|STM{<cond>}<add mode> Rn{!}, <registers>
LDM{<cond>}<add mode> Rn{!}, <registers + pc>^
LDM|STM{<cond>}<add mode> Rn, <registers - pc>^
```

Here `<cond>` is a condition code, `<addr mode>` is the address mode and `<registers>` is a list of registers. The block copying address modes are `IA`, `IB`, `DA` and `DB` – as indicated these increment/decrement the address register after/before each memory access.[2] An `!` is used for base register write-back, and the suffix `^` is used to set the `S` flag.

As an example, if the processor is in supervisor mode with the `Z` flag set, the instruction

```
LDMEQDB r0!, {r1,r2,pc}^
```

will perform the following assignments:

```
r0 ← r0-12; r1 ← mem[r0-12]; r2 ← mem[r0-8]; r15 ← mem[r0-4]; CPSR ← SPSR_svc .
```

All transfers are ordered: registers with lower indices are mapped to lower memory addresses.

The register list should not be empty i.e. the lowest sixteen bits of the op-code should not all be clear. This restriction will be enforced by any sensible compiler and/or assembler, but this does not guarantee that such instructions can never be executed (it is trivial to write an assembly program that generates and then executes such an instruction). The ARM6 has an unfortunate load multiple behaviour when the register list is empty – a load to the program counter occurs. Rather than specify this at the programmer's model level, the HOL model of the ARM6 has been modified to give a more sensible behaviour i.e. no load occurs.

With block stores, if the program counter is in the list then the value stored is implementation dependent. If the base register is in the list then write-back should not be specified because the result is *unpredictable*. The HOL programmer's model specification has been tailored to conform with ARM6 behaviour for these cases.

2.2 A HOL Specification

The HOL specification of the block data transfers is shown in Figure 3. The function `LDM_STM` takes the current programmer's model state, the processor mode and the instruction op-code, and it gives the next state. This function is only called when it is established that op-code `n`

[2]Stack based mnemonics are available as an alternative: `FA`, `FD`, `EA` and `ED` are used to implement full/empty, ascending/descending stacks.

```
⊢def LDM_STM (ARM mem reg psr) mode n =
      let (P,U,S,W,L,Rn,pc_in_list) = DECODE_LDM_STM n in
      let rn = REG_READ reg mode Rn in
      let (bl_list,rn') = ADDR_MODE4 P U rn n
      and mode' = if S ∧ (L ⇒ ¬pc_in_list) then usr else mode
      and pc_reg = INC_PC reg in
      let wb_reg =
           if W ∧ ¬(Rn = 15) then REG_WRITE pc_reg (if L then mode else mode') Rn rn' else pc_reg
      in
         if L then
           ARM mem (LDM_LIST mem wb_reg mode' bl_list)
             (if S ∧ pc_in_list then CPSR_WRITE psr (SPSR_READ psr mode) else psr)
         else
           ARM (STM_LIST mem (if FST (HD bl_list) = Rn then pc_reg else wb_reg) mode' bl_list) wb_reg psr
```

Figure 3: Programmer's model specification of block data transfers.

is a block data transfer instruction that passes the conditional execution test. The state space constructor `ARM` takes a triple: the memory `mem`, the general purpose registers `reg`, and the program status registers `psr`.

Although the definition of `LDM_STM` is not especially large, there are some subtle aspects to the semantics of block data transfers. Depending on the context, the processor mode is either `mode` or `mode'` (which might be set to user mode), therefore one must pay attention as to which mode is being used when accessing registers. The register bank after incrementing the program counter is denoted by `pc_reg` and after register write-back this becomes `wb_reg`. If the *first* register of a block store is the base register then the value `rn` is stored (i.e. `pc_reg` is used), otherwise write-back may have occurred and `rn'` is stored (`wb_reg` is used). Write-back occurs only if the base register is not the program counter.

There are four key sub-functions: `DECODE_LDM_STM`, `ADDR_MODE4`, `LDM_LIST` and `STM_LIST`; these are defined in Appendix B. The function `DECODE_LDM_STM` takes the op-code and splits it into seven fields. For example, the instruction `LDMEQDB r0!, {r1,r2,pc}^` is encoded with the natural number 158367750, and this decodes as follows:

```
⊢ DECODE_LDM_STM 158367750 = (T,F,T,T,T,0,T) .
```

The function `ADDR_MODE4` takes the address mode flags (`P` and `U`), the base address `rn` and the op-code `n`, and it gives a pair `(bl_list,rn')`. With our example:

```
⊢ ADDR_MODE4 T F rn 158367750 = ([(1,rn - 0xC); (2,rn - 0x8); (15,rn - 0x4)],rn - 0xC) .
```

If write-back is enabled then register `Rn` takes the value `rn'`; `bl_list` consists of pairs of the form `(rp,addr)` where `rp` is a register index and `addr` is the corresponding memory address. A function `REGISTER_LIST` gives the list of register indices and this is 'zipped' with the memory block addresses.

The function `LDM_LIST` folds the list `bl_list` with a memory-read, register-write operation to give the next state of the register bank. Likewise, the function `STM_LIST` folds the list with a register-read, memory-write operation to give the state of the main memory.

This list based specification is compact and hopefully clear. Consequently, one can be confident that the specification is consistency with respect to the reference [18] – it also provides a model that can be executed efficiently. However, the verification must bridge a large gap between this abstract semantics and the concrete processor implementation.

3 The Micro-architecture

A simplified view of the ARM6 data path is shown in Figure 4; the components of the data path (busses, latches, multiplexers and functional units) are used in executing all of the ARM instructions. When reading from memory the data is transfered to the data-in register `din`. When writing to memory the data is placed on the B bus. The address register `areg` may be updated using the program counter, the address incrementer, or output from the ALU.

Block data transfers are multi-cycled instructions; their execution can take from two to twenty cycles to complete. The execute stage is split into sub-stages and these are shown

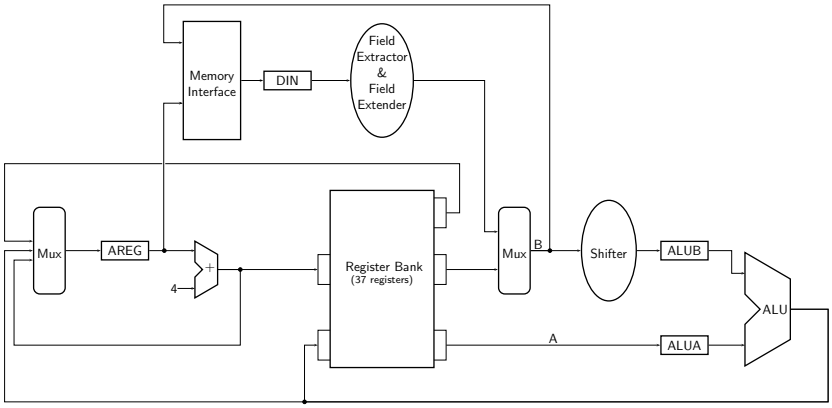

Figure 4: The ARM6 Data Path.

in Tables 1(a) and 1(b) for the block load and store instructions respectively. The first two cycles are needed for address computation and base register write-back, and then sub-stage t_n is repeated $\ell - 1$ times, where ℓ is the length of the register list. The leftmost columns in the tables show the value of the instruction sequence counter (`is`) – this component forms part of the processor's control logic. Note that the cycle t_2 is really the last execute cycle of the previous instruction and so in this case the component `is` does not actually take this value.

The first memory address (start of the memory block) is computed at cycle t_3 using an offset; this value is then stored in the address register. With `DB` addressing the offset is $4 * (\ell - 1) + 3$ and the first address is $\neg \mathit{offset} + rn$, where rn is the value of base register and $\neg x$ is the 32-bit one's complement of x. In our example (Section 2.1), the offset is 11 and the first address is $rn - 12$. On successive cycles the address register is incremented by four (only word access is supported). The last (write-back) address is computed at cycle t_4. With the `DB` address mode the write-back address is the same as the first address (i.e. $rn - 12$), but this is not the case for all of the other addressing modes.

A 16-bit mask is used to compute the register index used for each data transfer. The component `rp` stores the index for the next register to be processed – this is the *priority* register. The computation of `rp` (with two time shifted copies: `orp` and `oorp`) is shown below for our example instruction (op-code 158367750):

`is`	`mask`	`mask` $\wedge_{16}$ `ireg`	`rp`	`orp`	`oorp`
t_3	1111111111111111	1000000000000110	1		
t_4	1111111111111101	1000000000000100	2	1	
t_n	1111111111111001	1000000000000000	15	2	1
t_n	0111111111111001	0000000000000000	$\perp$	15	2
t_m	$\perp$	$\perp$	$\perp$	$\perp$	15

The operation $\wedge_{16}$ represents bitwise conjunction for 16-bit values. The t_n cycle lasts for two cycles – it is repeated until the mask conjunction (column three) is zero. The data transfers always occur in ascending register index order, irrespective of the addressing mode. Consequently, the priority register is always the lowest index position for a set bit in the conjunction. If the conjunction is zero then the value of `rp` is undefined and, consequently, so are the derived values; undefined values are represented by the symbol $\perp$.

In actuality, if `ireg[15:0]` is zero in a block load then an ARM6 will carry out a load with register fifteen as the destination register and this will instigate a branch. This is because of the way in which the index search is implemented in hardware (i.e. it gives the last possible value, which is fifteen) and the fact that the t_m cycle always occurs with block loads. It was decided not to model this counter intuitive behaviour in the HOL specification – instead, the value of `rp` is undefined and the control logic has been modified to prevent the load going ahead in this case – this gives a cleaner programmer's model specification. The block stores did not need modifying because the processor naturally avoids storing a value when the list is empty.

In our example there is a load to register fifteen and so a branch will occur. This means that additional cycles are required for instruction fetch and decode; the instruction will actually

Table 1: The sub-stages in the ARM6's block transfer execution. Cycle t_n is repeated until the bitwise conjunction of the mask and `ireg[15:0]` is zero.

(a) Block loads.

t_2	Set `mask` to `0xFFFF`
t_3	Fetch an instruction Increment the program counter Set `areg` to the first address (using `reg[Rn]` and `offset`) Set `rp` using `ireg` and `mask` Clear `rp` bit of `mask` Set `orp` to be `rp`
t_4	If write-back enabled then set reg[Rn] to the last address (using `offset`) Load `din` with `mem[areg]` Increment `areg` Set `rp` using `ireg` and `mask` Clear `rp` bit of `mask` Set `oorp` to be `orp` Set `orp` to be `rp`
t_n	Set `reg[oorp]` to be `din` Load `din` with `mem[areg]` Increment `areg` Set `rp` using `ireg` and `mask` Clear `rp` bit of `mask` Set `oorp` to be `orp` Set `orp` to be `rp`
t_m	Set `reg[oorp]` to be `din` Set `areg` to be program counter value If bit twenty-two and bit fifiteen of `ireg` set then set `cpsr` to be `spsr` Decode the next instruction

(b) Block stores.

t_2	Set `mask` to `0xFFFF`
t_3	Fetch an instruction Increment the program counter Set `areg` to the first address (using `reg[Rn]` and `offset`) Set `rp` using `ireg` and `mask` Clear `rp` bit of `mask` Set `orp` to be `rp`
t_4	If write-back enabled then set reg[Rn] to the last address (using `offset`) Store `reg[orp]` to `mem[areg]` Increment `areg` Set `rp` using `ireg` and `mask` Clear `rp` bit of `mask` Set `orp` to be `rp`
t_n	Store `reg[orp]` to `mem[areg]` If the last cycle then set `areg` to the program counter value and decode the next instruction otherwise increment `areg` Set `rp` using `ireg` and `mask` Clear `rp` bit of `mask` Set `orp` to be `rp`

fully complete two cycles after the t_m cycle.

3.1 A HOL Specification

A HOL specification of the ARM6 without block data transfers [6] was extended to cover this instruction class – most modifications to the previous specification are obvious and do not merit documenting here. Appendix C presents functions that were new to the specification.

The function `NBS` specifies the mode change caused by the `S` flag option i.e. it determines when the user mode is activated. The mask behaviour (as described in the previous section) is implemented using the functions `MASK`, `RP` and `PENCZ`. The function `MASK` defines the state of the mask – if the next instruction class (`nxtic`) is a block transfer then the initial mask value is `0xFFFF` and on subsequent cycles a single mask bit is cleared using the function `CLEARBIT`, see Appendix A. The masking is modelled using natural numbers – this is done simply to avoid introducing additional operator overloading i.e. by simultaneously loading 16-bit and 32-bit words theories.

The priority register `rp` is given by the function `RP`; this computes the bitwise conjunction of the register list and the mask, then the lowest set bit is determined using the function

$$\vdash_{def} \texttt{LEASTBIT n = LEAST b. BIT b n .}$$

When the register list is exhausted `rp` takes the value `LEASTBIT 0`, which is undefined; in HOL this is an unspecified natural number value. The predicate `PENCZ` holds true only when this termination condition is met. The function `LEASTBIT` is not readily executable (i.e. one cannot evaluate ground terms by adding the definition to a HOL *compset*); this is because

the LEAST operation introduces non-termination problems. When simulating the ARM6, a nested-if expansion of LEASTBIT (for all $b < 16$) is used.

The function OFFSET is used to compute the first and write-back addresses on cycles t_3 and t_4. The number of registers in the register list is determined using SUM and BITV (Appendix A). Thus, the number of registers is $\sum_{i=0}^{15} \mathrm{bitv}(ireg)(i)$. Depending on the addressing mode (and the instruction sequence) the offset, its one's complement, or zero is added to the base register address; this is implemented by the processor's ALU.

4 Correctness

The correctness model used for the ARM6 verification has been formalised in HOL [5]. The following sections introduce the models and abstractions that are used in establishing the ARM6 processor's correctness.

Store instructions require special attention because they can invalidating the state of a processor's pipeline [6]. This problem is heightened by the inclusion of block data transfers. For example, consider the following fragment of ARM code:

```
       ADR   r0, label
       STMIA r0, {r1,r2}
label: MOV   r3, #10
       MOV   r4, #12
```

The first instructions sets register r0 to the label address. The STMIA instruction then stores registers r1 and r2 to this and the following address, thus overwriting the two MOV instructions. However, rather than execute the new instructions (i.e. r1 and r2), an ARM6 will actually execute both of the MOV instructions. These instructions are preserved because they have entered the *pipeline* and the processor only *flushes* the instruction pipeline after a branch (write to the program counter). Of course, this example has been construed so as expose the pipeline and hence be unsafe. In practice, it is not worth wasting valuable processor logic on handling such fundamentally flawed code, and this is the position that was adopted by the designers of the ARM6. However, from a correctness standpoint, this must be dealt with. The approach adopted here is to augment the programmer's model state space with two 32-bit registers and these hold the op-codes of the fetched and decoded instructions. By implementing a rudimentary pipeline at the ISA level the correspondence between our models is easier to establish. The ISA pipeline is really just a buffer: the model still occupies the same level of temporal abstraction i.e. each cycle always corresponds with the execution of a single instruction. One criticism that could be made of this approach is that the semantics of the abstract model is now too concrete when compared with the reference description [18]. On the other hand, our model is a verified abstraction of the ARM6 and so one can be wholly confident that it faithfully simulates the processor regardless of the code being executed. However, it is not a suitable target for other ARM processors because there will be differences with respect to the *unpredictable* parts of the programmer's model. In order to unite the ARM processor family, one must introduce another level of abstraction and construct a non-deterministic instruction set model.

4.1 State Functions

The ARM architecture and ARM6 processor are modelled in HOL with the following functions:

```
STATE_ARM_PIPE:num→state_arm_pipe→state_arm_pipe
STATE_ARM6:num→state_arm6→state_arm6
```

A constructor ARM_PIPE extends the programmer's model state space (state_arm) with the state of the pipeline. The two additional 32-bit words are named ireg and pipe – they form a simple buffer which is emptied and re-filled when a branch occurs. This enables the ISA model to simulate ARM6 behaviour when storing data to the addresses $pc + 4$ and $pc + 8$.

With the inclusion of the block data transfers, the processor's state space (state_arm6) now contains three additional components: mask, orp and oorp. For convenience these components are of type num, but they are more naturally 16-bit and 4-bit values.

```
⊢def DUR_ARM6 (ARM6 mem (DP reg psr areg din alua alub) (CTRL pipea pipeaval ..  mshift)) =
    let (nzcv,m) = DECODE_PSR (CPSR_READ psr) in
    let abortinst = ABORTINST iregval onewinst ointstart ireg nzcv in
    let ic = IC abortinst nxtic in
    let len = LENGTH (REGISTER_LIST (w2n ireg))
    in
       if ic = undef then
         4
       else if ..
         ..
       else if ic = ldm then
         2 + (len - 1) + 1 + (if WORD_BIT 15 ireg then 2 else 0)
       else if ic = stm then
         2 + (len - 1)
       else if ..
        ..
```

Figure 5: The duration map DUR_ARM6.

4.2 Data and Temporal Abstraction

A data abstraction ABS_ARM6:state_arm6→state_arm_pipe is defined as follows:

```
⊢def ABS_ARM6 (ARM6 mem (DP reg psr areg din alua alub)
       (CTRL pipea pipeaval pipeb pipebval ireg iregval ointstart
          onewinst opipebll nxtic nxtis aregn nbw nrw sctrlreg psrfb
            oareg mask orp oorp mul mul2 borrow2 mshift)) =
       ARM_PIPE (ARM mem (SUB8_PC reg) psr) pipeb ireg
```

The state components are grouped into vectors using five constructors: ARM6, DP (the data path), CTRL (the processor control), ARM_PIPE and ARM. The data abstraction projects out the pipeline state (pipeb and ireg) and the visible state components (mem, reg and psr). The function SUB8_PC is used to subtract eight from the ARM6's program counter value, which is eight bytes ahead of the address of the instruction being executed.

A *uniform immersion* [5] specifies the temporal relationship between the cycles of the ARM6 processor and single instruction execution. A function DUR_ARM6:state_arm6→num specifies the number of cycles required to complete the execution of an instruction. A fragment of this function, giving the timings for block data transfers, is shown in Figure 5. The instruction class (ic) and the length of the register list (len) are used to determine how long the block data transfer will take. The timings are presented as sums; this splits the execution into distinct phases. For example, with an LDM instruction the first two cycles are t3 and t4, then there are $\ell - 1$ cycles of tn, followed by one cycle of tm, and finally two extra cycles if the program counter is in the register list.

This duration function is only valid for processor states in which the pipeline is full i.e. the first execute cycle is about to commence – the component nxtis must have the value t3. An initialisation function for the ARM6 is provided in the following section. During verification one must show that the timings specified above are consistent with passing from one initial state to another.

4.3 Initialisation

An initialisation function INIT_ARM6:state_arm6→state_arm6 is used to ensure that the processor starts in a valid state. This function takes a state and converts it into an initial version – it is an identity mapping on valid initial states:

```
⊢def INIT_ARM6 (ARM6 mem (DP reg psr areg din alua alub) (CTRL pipea pipeaval .. mshift)) =
    let nxtic' = DECODE_INST (w2n ireg) in
      ARM6 mem (DP reg psr (REG_READ6 reg usr 15) ireg alua alub)
        (CTRL pipea T pipeb T ireg T F T T nxtic' t3 2 nbw F sctrlreg
           psrfb oareg (MASK nxtic' t3 mask ARB) orp oorp mul mul2 borrow2 mshift)
```

This function differs significantly from the earlier verification [6] – our ISA model now has a pipeline and so the pipeline components (pipea, pipeb and ireg) can be initialised with any values. If these values are not consistent with the instructions in memory (corresponding

with the current value of the program counter) then conceivably this could be because a store instruction has just invalidated the pipeline's state. However, this is not a problem because the pipeline state is visible at the ISA level, via the data abstraction `ABS_ARM6`. The components `orp` and `oorp` are not altered, but the mask is set using the function `MASK` – if the next instruction class is a block data transfer then this will set `mask` to the value `0xFFFF`.

With respect to initialisation, there are three classes of component:

- The visible state components: `mem`, `reg`, `psr`, `pipeb` and `ireg`. These components cannot be altered during initialisation because otherwise correctness would fail at time zero.
- State components whose initial values are of significance. For example, the next instruction class (`nxtic`) must be the decoding of the instruction register (`ireg`).
- State components whose initial values are of no significance. For example, the ALU registers (`alua` and `alub`) can take any values initially.

As a general rule an initialisation function should be as weak as possible i.e. it should only alter state components that are of the second type. In this context the initialisation represents an invariant for the design. However, the initialisation function is actually viewed as *part* of the design i.e. it is used to define the state function `STATE_ARM6` and is used when simulating the processor.

4.4 Correctness Definition

The ARM6 is considered correct if:

Commutativity Theorem

```
⊢ ∀t a. STATE_ARM_PIPE t (ABS_ARM6 a) = ABS_ARM6 (STATE_ARM6 (IMM_ARM6 a t) a)
```

where `IMM_ARM6` is the uniform immersion with duration function `DUR_ARM6`. To ensure that the implementation covers all of the specification's behaviour, one must also show that the data abstraction is a surjective mapping i.e. each initial specification state must have at least one initial implementation state that maps to it. This condition is relatively easy to verify for `ABS_ARM6` because the operation `SUB8_PC` has an obvious inverse. The main focus of the formal verification is, therefore, the commutativity theorem.

5 Formal Verification

The correctness condition presented in Section 4.4 is universally quantified over time (the natural numbers). Using the one-step theorems [5], it is sufficient to prove that the following four theorems hold:

```
1  ⊢ ∀a. (STATE_ARM6 (IMM_ARM6 a 0) a = a') ⇒ (INIT_ARM6 a' = a')
2  ⊢ ∀a. (STATE_ARM6 (IMM_ARM6 a 1) a = a') ⇒ (INIT_ARM6 a' = a')
3  ⊢ ∀a. STATE_ARM_PIPE 0 (ABS_ARM6 a) = ABS_ARM6 (STATE_ARM6 (IMM_ARM6 a 0) a)
4  ⊢ ∀a. STATE_ARM_PIPE 1 (ABS_ARM6 a) = ABS_ARM6 (STATE_ARM6 (IMM_ARM6 a 1) a)
```

The first and third theorems are trivial; the main verification effort lies in verifying the second and forth theorems. Here, the next state function `NEXT_ARM6` is iterated using `FUNPOW`, with the number of iterations given by the map `DUR_ARM6` ∘ `INIT_ARM6`. The proof proceeds with case splitting over the instruction class and this normally gives a small constant value for the number of iterations. However, with the block data transfer instruction class, the duration is a function of the length of the register list. Exhaustive proof over all of the 2^{16} possible register lists is not a viable option, especially considering that further case splitting is required for each combination of addressing mode and the options `S`, `W`, `L`, `Rn = 15` and `pc_in_list`.

5.1 Approach

In order to verify the block data transfers, invariants are constructed for the iterated t_n-phase of the execution. This phase occurs two cycles into the execution and accounts for $\ell - 1$ cycles, where ℓ is the length of the register list. Three cases must be considered: $\ell = 0$, $\ell = 1$ and

$1 < \ell$. In the first two cases the t_n-phase does not occur; with stores this means that the execution is complete after the t_3 and t_4 cycles, but with loads completion occurs after the t_m cycle, which will be followed by two extra cycles if the list is `{r15}`. Therefore, invariants are only needed when $1 < \ell$.

Let A be the processor's state space and $f : A \to A$ be the next state function. The state space has two disjoint subsets $X_c = \text{Image}(f^2, I_c)$, where I_c is the set of initial states for the classes $c \in \{\texttt{ldm}, \texttt{stm}\}$. Induction is used to verify that the functions $g_c : \mathbb{N} \times X_c \to A$ have the property: for all $a \in X_c$, $1 < \ell$ and $i < \ell - 1$

$$g_c(i, a) = f^i(a) \ .$$

The functions g_c are a form of invariant; they were constructed manually – an initial definition was made (guessed at) and this was refined until the final version was proved to be valid. Functions $h_c : X_c \to A$ are defined by

$$h_c(a) = f(g_c(\ell - 2, a)) = f^{\ell-1}(a) \ .$$

At cycle $\ell - 2$ the termination condition is about to be met and so each function h_c maps states in the set X_c to states corresponding with the end of the t_n-phase of execution.

Using the functions h_c it is now possible to express the state of the processor after completing the execution of the block data transfers; for all $a \in I_c$ the final state is:

$$\begin{cases} f^2(a), & \text{if } c = \texttt{stm} \text{ and } \ell = 0, 1, \\ h_{\texttt{stm}}(f^2(a)), & \text{if } c = \texttt{stm} \text{ and } 1 < \ell, \\ f^3(a), & \text{if } c = \texttt{ldm} \text{ and } \ell = 0, \\ f^3(a), & \text{if } c = \texttt{ldm} \text{ and } \ell = 1 \text{ and } \texttt{r15} \text{ not in list}, \\ f^5(a), & \text{if } c = \texttt{ldm} \text{ and } \ell = 1 \text{ and } \texttt{r15} \text{ in list}, \\ f(h_{\texttt{ldm}}(f^2(a))), & \text{if } c = \texttt{ldm} \text{ and } 1 < \ell \text{ and } \texttt{r15} \text{ not in list}, \\ f^3(h_{\texttt{ldm}}(f^2(a))), & \text{if } c = \texttt{ldm} \text{ and } 1 < \ell \text{ and } \texttt{r15} \text{ in list}. \end{cases}$$

The initial state sets I_c are generated using the initialisation function. Having determined the state of the processor at the times given by the duration function, it is then necessary to relate these states to those of the specification. The following sections indicate how the h_c functions were constructed and show how the masking used in the processor model is related to the list model used in the ISA specification.

5.2 Lemmas about Priority Register Masking

The following functions are defined in HOL:

```
⊢def GEN_RP wl ireg mask = LEASTBIT (BITWISE wl (∧) ireg mask)
⊢def MASK_BIT wl ireg mask = CLEARBIT wl (GEN_RP wl ireg mask) mask
⊢def MASKN wl n ireg = FUNPOW (MASK_BIT wl ireg) n (ONECOMP wl 0)
```

These function generalise those of the ARM6 specification to an arbitrary mask length `wl`; this enables results to be proved by induction over the word length. The function `MASKN` gives the n^{th} value of the mask; with our block load example:

```
⊢ RP ldm (BITS 15 0 158367750) (MASKN 16 0 158367750) = 1
⊢ RP ldm (BITS 15 0 158367750) (MASKN 16 1 158367750) = 2
⊢ RP ldm (BITS 15 0 158367750) (MASKN 16 2 158367750) = 15
⊢ PENCZ (BITS 15 0 158367750) (MASKN 16 3 158367750)
```

These values correspond with those in the table on page 5.

The ISA level function `REGISTER_LIST` is also generalised to an arbitrary length:

```
⊢def GEN_REG_LIST wl a = (MAP SND o FILTER FST) (GENLIST (λb. (BIT b a,b)) wl)
```

Two key lemmas relate this function with the ARM6 model:

```
⊢ ∀wl ireg. LENGTH (GEN_REG_LIST wl ireg) = SUM wl (BITV ireg)

⊢ ∀wl ireg n. n < LENGTH (GEN_REG_LIST wl ireg) ⇒
              (EL n (GEN_REG_LIST wl ireg) = GEN_RP wl ireg (MASKN wl n ireg))
```

The first theorem shows that the length of the list is equal to the sum of the constituent bits. The second theorem shows that n^{th} element of the register list corresponds with the priority register obtained using the n^{th} mask value. Specialising `wl` to be sixteen then provides a connection between the function `REGISTER_LIST`, and the functions `RP` and `MASK`. The second lemma uses the first and it requires some work to prove: `GEN_REG_LIST` uses the primitives `MAP`, `FILTER`, `GENLIST` and `BIT`; and `MASKN` uses `FUNPOW`, `LEAST`, `BITWISE`, `ONECOMP` and `BIT`.

Another key lemma concerns the termination condition:

```
⊢ ∀ic. (ic = ldm) ∨ (ic = stm) ⇒
        (∀a n. n < LENGTH (REGISTER_LIST a) ⇒ ¬PENCZ ic a (MASKN 16 n a)) ∧
         ∀a. PENCZ ic a (MASKN 16 (LENGTH (REGISTER_LIST a)) a)
```

This lemma shows that the termination predicate `PENCZ` is false up until the ℓ^{th} mask value.

5.3 Block Data Transfers

In the previous section the function `REGISTER_LIST` was related to the ARM6's implementation, which uses a 16-bit mask. This section covers the functions `LDM_LIST` and `STM_LIST`. The following functions are defined in HOL:

```
⊢def REG_WRITE_RP n reg mode mem ireg first =
        REG_WRITE reg mode (RP ldm ireg (MASKN 16 n ireg)) (MEMREAD mem (first + w32 n * w32 4))

⊢def MEM_WRITE_RP n reg mode mem ireg first =
        MEM_WRITE_WORD mem (first + w32 n * w32 4) (REG_READ6 reg mode (RP stm ireg (MASKN 16 n ireg)))
```

The functions `REG_WRITE_RP`/`MEM_WRITE_RP` represent the micro-architecture level load/store operation for the n^{th} word transfered. These are used in the following definitions:

```
⊢def (REG_WRITEN 0 reg mode mem ireg first = reg) ∧
     REG_WRITEN (SUC n) reg mode mem ireg first =
     REG_WRITE_RP n (REG_WRITEN n reg mode mem ireg first) mode mem ireg first

⊢def (MEM_WRITEN 0 reg mode mem ireg first = mem) ∧
     MEM_WRITEN (SUC n) reg mode mem ireg first =
     MEM_WRITE_RP n reg mode (MEM_WRITEN n reg mode mem ireg first) ireg first
```

The functions `REG_WRITEN`/`MEM_WRITEN` give the n^{th} state of the register-bank/memory while performing a block load/store; they are used in constructing and validating the g_c functions in Section 5.1. The final state of the register-bank or memory – as given by the functions h_c – is obtained when the first argument is ℓ. The following lemmas relate these definitions with `LDM_LIST` and `STM_LIST`:

```
⊢ ∀P U base mem reg mode.
        LDM_LIST mem reg mode (FST (ADDR_MODE4 P U base ireg)) =
        REG_WRITEN (LENGTH (REGISTER_LIST ireg)) reg mode mem ireg
          (FIRST_ADDRESS P U base (WB_ADDRESS U base (LENGTH (REGISTER_LIST ireg))))

⊢ ∀P U base mem reg mode.
        STM_LIST mem (SUB8_PC reg) mode (FST (ADDR_MODE4 P U base ireg)) =
        MEM_WRITEN (LENGTH (REGISTER_LIST ireg)) reg mode mem ireg
          (FIRST_ADDRESS P U base (WB_ADDRESS U base (LENGTH (REGISTER_LIST ireg))))
```

The first element of `ADDR_MODE4` is a list of register indices paired with memory addresses (see Section 2.2). The lemmas show that applying the list folding operations `LDM_LIST`/`STM_LIST` to this list is equivalent to applying `REG_WRITEN`/`MEM_WRITEN` with appropriate arguments.

The second lemma accounts for the possibility of storing the program counter. The function `STM_LIST` uses `REG_READ` to access the registers, whereas `MEM_WRITEN` uses `REG_READ6`; the former adds eight to the program counter value, but this is countered by the data abstraction which applies `SUB8_PC`. With load instructions, a series of additional lemmas are required to manipulate (normalise) various combinations of register updates (generated by the *pc*-increment,

write-back, block load and data abstraction operations) and these must take account of whether or not the fifteenth bit of the instruction register is set.

The first address is expressed using the function FIRST_ADDRESS; the ARM6 uses the ALU and an offset to compute this value. The following lemma shows that this computation is correct:

```
⊢ ∀ireg ic base c borrow2 mul.
        1 ≤ LENGTH (REGISTER_LIST (w2n ireg)) ∧
        ((ic = ldm) ∨ (ic = stm)) ⇒
        (FIRST_ADDRESS (WORD_BIT 24 ireg) (WORD_BIT 23 ireg) base
           (WB_ADDRESS (WORD_BIT 23 ireg) base (LENGTH (REGISTER_LIST (w2n ireg)))) =
         SND (ALU6 ic t3 ireg borrow2 mul (OFFSET ic t3 ireg (WORD_BITS 15 0 ireg)) base c))
```

There is a similar lemma to show that the computation of the write-back address, at cycle t_4, is also correct.

5.4 Summary

The formal verification makes use of one-step theorems [5]. The two main verification conditions (theorems two and four on page 9) are tackled using case splitting and the simplifier (term-rewriting). The first level of case splitting is on the instruction class; this means that pre-existing parts of the proof script [6] are used without major alteration.[3] The ARM6 implementation of the block data transfers is symbolically executed using functions h_c; temporal induction is used to prove that these functions evaluate the t_n-phase of execution (Section 5.1). Seven sub-cases are listed in Section 5.1, however, further case splitting is used to reason about particular instruction variants (e.g. with write-back, user mode access or when restoring the CPSR; and also whether the first register of a block store is the base register). The resultant processor states are expressed using functions REG_WRITEN and MEM_WRITEN; these are shown to be related to the function LDM_LIST and STM_LIST using lemmas about priority register masking (Sections 5.2 and 5.3). These and other lemmas are used to relate processor states with those given by the ISA specification.

By including the pipeline state at the ISA level, there was no need to explicitly consider the special cases of writing to the memory addresses $pc + 4$ and $pc + 8$. Using the no-clobber or data forwarding methods [6] would have added to the verification effort.

With the size and complexity of the ARM6 model, it is quite easy for the proof run-times and terms (representing the state of the processor) to become very large. Generating lots of sub-goals, possibly containing large terms, inevitably burdens the individual carrying out an interactive proof. This is mitigated by structuring the proof with the use of lemmas and by being careful in choosing when and how to case split. The method of state evaluation is also of significance; one must decide when to – or, more importantly, not to – expand with a given function definition. Call-by-value conversion is used when evaluating the next state function but this is then combined with the simplifier, which provides contextual re-writing. Although the complexity of the design must be managed it is not an overwhelming problem. The script files for the block data transfer lemmas and the main proof (covering all of the instruction classes) are both approximately a thousand lines long. The overall proof run-time is in the order of a few minutes.

6 Conclusion

This paper has described the work that was involved in extending a partial ARM6 verification [6] to include the block data transfer instruction class. The HOL proof system has been used to construct a concise programmer's model formalisation for this class; this is based on using standard list operations, which are provided in the standard HOL distribution. Daniel Schostak's specification was used as the basis for the HOL model of the ARM6 micro-architecture. The implementation's execute stage is multi-cycled, with a block load taking up to twenty cycles to complete. The instruction timing is determined by the number of registers to be transferred and this is specified by a duration map. In previous verification work with

[3] Some changes were made with the inclusion of the pipeline state at the ISA level. In particular, the single data store proof became simpler.

micro-programmed and pipelined designs [5, 6] the processor control logic has been sufficiently simple that the duration for each instruction is a known constant value. Therefore, additional verification techniques have had to be employed in order to reason about the correctness of the block data transfers. In particular, it was necessary to use induction over time to establish the behaviour of the processor during a sub-stage of the execution. This sub-stage is preceded by two cycles (forming an initial state precondition) and the instruction may complete up to three cycles afterwards. In order to relate the list model with the masking method, a number of auxiliary functions were defined; these enabled inductions to be carried out on the register list length. Functions were defined so as to directly specify the state of the processor part way through the iterated t_n-phase of execution. This paper has presented a number of key lemmas that were used in relating the two different models. The `LEAST` operator was used in specifying the next register to be transfered by the processor; HOL provides a few handy theorems for reasoning about this operator.

This work has demonstrated that the verification strategy (based on symbolic execution) is well suited to adding further instructions to a verified processor design. It was a relatively straightforward task to modify the processor and instruction set models, and much of the pre-existing proof scripts needed little or no modification. This point has been further demonstrated with the verification of the multiply instruction class. The proof run-time for the verification of the block data transfers is longer than for most other instruction classes, but the overall run-time has only increased proportionately i.e. the proof run-time is essentially linear with respect to the number of instruction classes.

To completely verify a commercial processor design one will inevitably have to tackle complex instruction classes such as the block data transfers. This may entail verifying that invariants hold during given phases of instruction execution. This has been shown to be feasible with the HOL model of the ARM6. All core instruction classes have now been verified.

References

[1] Sven Beyer, Chris Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang Paul. Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In Daniel Geist and Tronci Enrico, editors, *Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 51–65. Springer-Verlag, 2003.

[2] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In Mandayam K. Srivas and Albert Camilleri, editors, *FMCAD '96*, volume 1166 of *LNCS*, pages 275–293. Springer-Verlag, 1996.

[3] 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.

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

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

[6] Anthony Fox. Formal verification of the ARM6 micro-architecture. Technical Report 548, University of Cambridge Computer Laboratory, November 2002.

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

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

[9] Neal Harman and John Tucker. Algebraic models and the correctness of microprocessors. In George Milne and Laurence Pierre, editors, *Correct Hardware Design and Verification*

Methods, volume 683 of *Lecture Notes in Computer Science*, pages 92–108. Springer-Verlag, 1993.

[10] Neal A. Harman. Verifying a simple pipelined microprocessor using Maude. In M Cerioli and G Reggio, editors, *Recent Trends in Algebraic Development Techniques: 15th Int. Workshop, WADT 2001*, volume 2267 of *Lecture Notes in Computer Science*, pages 128–151. Springer-Verlag, 2001.

[11] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor*, volume 795 of *LNCS*. Springer-Verlag, 1994.

[12] Robert B. Jones, Jens U. Skakkebæk, and David L. Dill. Formal verification of out-of-order execution with incremental flushing. *Formal Methods in System Design*, 20(2):139–158, March 2002.

[13] Jeffrey J. Joyce. Formal verification and implementation of a microprocessor. In Graham Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 129–157. Kluwer Academic Publishers, 1988.

[14] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.

[15] K. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In A. J. Hu and M. Y. Vardi, editors, *CAV '98*, volume 1427 of *LNCS*. Springer-Verlag, 1998.

[16] 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.

[17] Jun Sawada and Warren A. Hunt, Jr. Verification of FM9801: An out-of-order model with speculative execution, exceptions, and program-modifying capability. *Formal Methods in System Design*, 20(2):187–222, March 2002.

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

[19] Phillip. J. Windley and Michael L. Coe. A correctness model for pipelined microprocessors. In Ramayya Kumar and Thomas Kropf, editors, *TPCD '94*, volume 901 of *LNCS*, pages 33–51. Springer-Verlag, 1995.

Appendix

A Primitive Operations

```
⊢def BITS h l n = n MOD 2^SUC h DIV 2^l
⊢def BIT b n = (BITS b b n = 1)
⊢def WORD_BITS h l n = BITS h l (w2n n)
⊢def WORD_BIT b n = BIT b (w2n n)
⊢def BITV n b = BITS b b n
⊢def SBIT b n = (if b then 2^n else 0)
⊢def (BITWISE 0 op x y = 0) ∧
     BITWISE (SUC n) op x y = BITWISE n op x y + SBIT (op (BIT n x) (BIT n y)) n

⊢def ONECOMP wl n = 2^wl - 1 - n MOD 2^wl
⊢def CLEARBIT wl b a = BITWISE wl (∧) a (ONECOMP wl 2^b)
⊢def (SUM 0 f = 0) ∧ SUM (SUC m) f = SUM m f + f m

⊢def (∀f e. FOLDL f e [] = e) ∧ ∀f e x l. FOLDL f e (x::l) = FOLDL f (f e x) l
⊢def (∀f. MAP f [] = []) ∧ ∀f x l. MAP f (x::l) = f x::MAP f l
⊢def (∀P. FILTER P [] = []) ∧ ∀P h t. FILTER P (h::t) = (if P h then h::FILTER P t else FILTER P t)
⊢def (∀l. EL 0 l = HD l) ∧ ∀n l. EL (SUC n) l = EL n (TL l)
⊢def (ZIP ([],[]) = []) ∧ ∀x1 l1 x2 l2. ZIP (x1::l1,x2::l2) = (x1,x2)::ZIP (l1,l2)
⊢def (∀x. SNOC x [] = [x]) ∧ ∀x x' l. SNOC x (x'::l) = x'::SNOC x l
⊢def (∀f. GENLIST f 0 = []) ∧ ∀f n. GENLIST f (SUC n) = SNOC (f n) (GENLIST f n)
```

B ISA Specification

```
⊢def REGISTER_LIST n = (MAP SND o FILTER FST) (GENLIST (λb. (BIT b n,b)) 16)

⊢def ADDRESS_LIST start n = GENLIST (λa. start + w32 (4 * a)) n

⊢def WB_ADDRESS U base len = (if U then $+ else $-) base (w32 (4 * len))

⊢def FIRST_ADDRESS P U base wb =
        if U then if P then base + w32 4 else base else if P then wb else wb + w32 4

⊢def ADDR_MODE4 P U base n =
         let reg_list = REGISTER_LIST n in
         let len = LENGTH reg_list in
         let wb = WB_ADDRESS U base len in
         let addr_list = ADDRESS_LIST (FIRST_ADDRESS P U base wb) len
         in (ZIP (reg_list,addr_list),wb)
```

```
⊢def LDM_LIST mem reg mode bl_list =
        FOLDL (λreg' (rp,addr). REG_WRITE reg' mode rp (MEMREAD mem addr)) reg bl_list

⊢def STM_LIST mem reg mode bl_list =
        FOLDL (λmem' (rp,addr). MEM_WRITE_WORD mem' addr (REG_READ reg mode rp)) mem bl_list
```

```
⊢def DECODE_LDM_STM n = (BIT 24 n,BIT 23 n,BIT 22 n,BIT 21 n,BIT 20 n,BITS 19 16 n,BIT 15 n)
```

C Addendum to the ARM6 Specification

```
⊢def NBS ic is ireg m =
        if WORD_BIT 22 ireg ∧
          (((is = tn) ∨ (is = tm)) ∧ (ic = ldm) ∧ ¬WORD_BIT 15 ireg ∨
           ((is = t4) ∨ (is = tn)) ∧ (ic = stm))
        then
          usr
        else
          DECODE_MODE m

⊢def MASK nxtic nxtis mask rp =
         if (nxtic = ldm) ∨ (nxtic = stm) then
            if nxtis = t3 then ONECOMP 16 0 else CLEARBIT 16 rp mask
         else
           ARB

⊢def RP ic list mask =
         if (ic = ldm) ∨ (ic = stm) then
           LEASTBIT (BITWISE 16 (∧) list mask)
         else
           ARB

⊢def PENCZ ic list mask =
         if (ic = ldm) ∨ (ic = stm) then
           BITWISE 16 (∧) list mask = 0
         else
           ARB

⊢def OFFSET ic is ireg list =
        if (is = t3) ∧ ((ic = ldm) ∨ (ic = stm)) then
          if WORD_BIT 23 ireg then
            w32 3
          else if WORD_BIT 24 ireg then
            w32 (4 * (SUM 16 (BITV list) - 1) + 3)
          else
            w32 (4 * (SUM 16 (BITV list) - 1))
        else if (is = t4) ∧ ((ic = ldm) ∨ (ic = stm)) then
          w32 (4 * (SUM 16 (BITV list) - 1) + 3)
        else if (is = t5) ∧ ((ic = br) ∨ (ic = swi_ex)) then
          w32 3
        else
          ARB
```