Verifying ARM6 Multiplication

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 the multiply instructions. The processor's implementation is based on the *modified Booth's algorithm*. Correctness is defined using data and temporal abstraction maps. 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. This paper describes the approach used in the formal verification and presents some key lemmas.

1 Introduction

This paper discusses the ARM6 implementation of the multiply instructions, and the formal verification using the HOL proof system. For information about the ARM architecture the reader is referred to Furber and the ARM Architecture Reference Manual [10, 22]. The verification of the block data transfer instruction class and all of the remaining instructions is documented elsewhere [9, 8]. The approach used is based on an algebraic framework for correctness that was developed at Swansea [12] and later implemented in HOL at Cambridge [7]. Work has continued at Swansea using Maude [13].

The processor's micro-architecture and instruction set architecture (ISA) are given functional specifications (they are modelled with *state functions*); correctness is established with respect to data and temporal abstraction maps. An *immersion* gives the times at which the processor's state corresponds with the *programmer's model* specification.

The formal verification is based on *symbolic execution* – the instruction set, micro-architecture and abstractions (which are all modelled with functions) are evaluated using term-rewriting. This method is highly dependable (when using a fully expansive theorem prover, such as HOL), flexible and widely applicable, since it is not tailored to specific types of design. Also, proofs can be reasonably robust with respect to changes in the design. It was possible to verify the multiplies in isolation from the rest of the instruction set. Another advantage of producing functional specifications is that one can carry out simulations: our models were sanity checked by executing small ARM assembler programs – this is especially helpful with complex instructions, such as the block data transfers and multiplication. HOL-4 has proved to be well suited to this methodology: it provides good support for defining data types and functions; and the specification can be executed using call-by-value conversion [2]. The maturity of the

system means that one can draw upon a range of existing theories. It is also possible to develop new theories, construct lemmas and introduce abstractions as required. For example, a theory of n-bit words (based on modular arithmetic) was developed for this work. The correctness framework is naturally formalised in higher-order logic – one can define and reason about correctness in an abstract setting. This guarantees that the definition of correctness is identical from one case study to the next and helps ensure there is no ambiguity as to what has been proved. One can also prove and make use of general results, such as the *one-step* theorems [7].

The ARM6 implementation of multiplication is unusual in that the operation is not directly provided by the ALU. Instead, execution is multi-cycled with the processor's control logic making use of a barrel shifter and the ALU. This means that there is no clear split between the data and control aspects of the design. Therefore, any methodology that seeks to separately reason about control and data will encounter problems in this context. Although the ARM6 multiplication is not representative with respect to modern microprocessor implementations, one cannot rule out other circumstances in which a processor's control and data aspects become intertwined.

The nature of ARM6 multiplication has meant that it is not wholly straightforward to symbolically execute the model. In verifying this instruction class, an induction over time was used to establish the value of the destination register for each cycle of the pipeline's execute stage. This is a departure from previous work [8] in which such an induction is not necessary. The correctness of the modified Booth's algorithm is established with a couple of key lemmas; these abstract out many details found in the ARM6 model. Although the ARM6 model is complex, proof run-time has not been a problem – the entire processor verification takes just a few minutes.

An ISA specification of the multiply instructions is presented in Section 2. The ARM6 implementation is based on a form of Booth's algorithm, and the micro-architecture specification is discussed in Section 3. A formalisation of correctness, in terms of data and temporal abstraction maps, is given in Section 4. The formal verification in HOL is then discussed in Section 5.

1.1 Related Work

Early work on the mechanical verification of processors includes: TAMARACK [16], SECD [11], the partial verification of Viper [6], Hunt's FM8501 [14], and the generic interpreter approach of Windley [23]. Following this work, Miller and Srivas verified some of the instructions of a simple commercial processor called the AAMP5 [20]. Complex commercial designs have also been specified, simulated and verified using ACL2 [4, 17].

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 [19, 15, 21, 3]. The instruction sets used

for this work are often relatively simple, 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 [5]. It has been observed that the Burch and Dill model is not perfect and, rather confusingly, plenty of alternatives exist [18, 1]. We use a strict, well-founded correctness model that is not specific to pipelined designs.

2 ARM Multiply Instructions

The ARM multiply instruction syntax is:

```
MUL{<cond>}{S} Rd, Rm, Rs
MLA{<cond>}{S} Rd, Rm, Rs, Rn
```

These instructions carry out 32-bit[1] multiplication with registers `Rm` and `Rs`. The result is stored in the destination register is `Rd` and the accumulate version (`MLA`) increments the result with register `Rn`. As with all other ARM instructions, the execution is conditional, as designated by a condition code `<cond>`. The `S` suffix indicates that the `N`, `Z` and `C` flags are to be set in the Current Program Status Register (CPSR). The instruction encoding is shown in Figure 1.

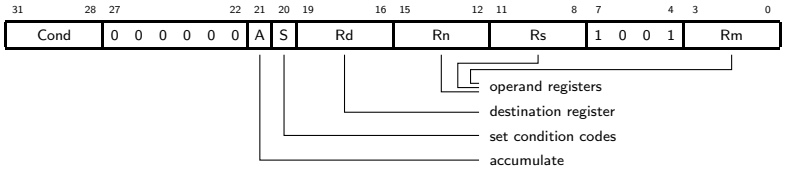

Fig. 1: Machine encoding for multiply instructions.

2.1 HOL Specification

A HOL specification for the multiply instructions is shown in Figure 2. The function `MUL_MLA` gives the programmer's state after executing the multiply instruction; this function takes the current programmer's model state (formed with the constructor `ARM`), the state of the carry flag, the processor mode and the instruction's op-code `n`. The op-code is decoded into a 6-tuple using the function `DECODE_MLA_MUL`. The register arguments `rm`, `rs` and `rn` are obtained by reading from the register map `reg`.

The multiplication result forms a 4-tuple and this is determined by the following function:

[1] Signed and unsigned long multiplication is only supported by later generations of ARM processors.

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

Fig. 2: The ARM multiply specification.

```
⊢def ALU_multiply A rm rs rn C =
       let res = if A then rm * rs + rn else rm * rs in
         (MSB res, res = word_0, MLA_MUL_CARRY rm rs C, res)
```

The first two elements are truth values and these indicate whether the result is negative or equal to zero. The third element is the carry, but this truth value will vary from one processor design to the next. The function `MLA_MUL_CARRY` gives the carry value associated with the ARM6 implementation – this is the last carry-out value from the barrel shifter. If the `S` flag is set then the CPSR is updated. The result `res` is stored in register `Rd`.

Multiply instructions are not valid when the destination register is the same as the multiplier register `Rm` or is the program counter (register fifteen). To make our specification (and verification) total, the programmer's model is made to conform with ARM6 behaviour: in these cases the instruction is effectively a no-op. This is actually a pragmatic simplification of the true ARM6 behaviour: if `Rd` is equal to `Rm` then an `MLA` instruction will give a nonsense result. Modelling and verifying this would have added a reasonable amount of complication, whereas it was quite simple to modify the ARM6 model to give the no-op behaviour. Code of this form should not exist because the architecture reference books say that such code is *unpredicatable*. Our model is a legitimate variant of the actual ARM6, however the absolute assurance provided by this formal verification would only be fully realised if this ARM6 model were taken to the point of fabrication.

3 ARM6 Multiplication

When the ARM6 was developed, in the early 1990s, ARM was a small company with limited resources, therefore it was important for them to keep the design complexity (and cost) of their new processor down. This would help reduce the time to market, and by having a low transistor count the processor would consume less power. To this end, the processor's ALU was limited to addition/subtraction; multiplication is implemented by the processor's control logic, which makes use of the barrel shifter and the ALU. The ARM6 data path is shown in Figure 3. The multiply instructions can take up to seventeen clock

cycles to complete, this is because the multi-cycled implementation is based on the modified Booth's algorithm. Although ARM6 multiplication is pretty slow, it is not a frequently used instruction class and in many cases assembly code can be optimised by replacing simple multiplications with a small number of other data operations (all data processing instructions can shift one of their arguments).

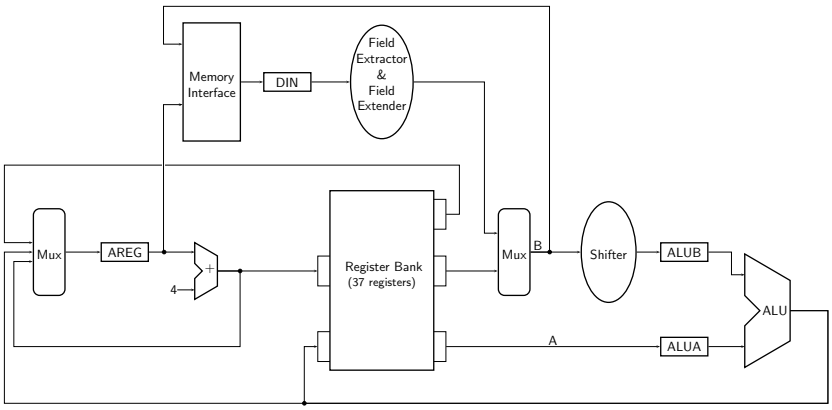

Fig. 3: The ARM6 data path.

A pseudo-code description of the modified Booth's algorithm is presented in Appendix A. When accumulating, the result *rd* is initially set to the value *rn*, otherwise it is set to zero. For each triple of bits[2] in the multiplicand *rs*, the multiplier *rm* is shifted left ($2n$ or $2n+1$ places) and then conditionally added or subtracted from *rd*. The bit triples account for the eight cases found in the body of the while loop – this loop is terminated when the remaining bits of *rs* are all zero.

Figure 4 shows how the multiply instructions have been implemented by the ARM6. Booth's algorithm is implemented using the resources of the processor's data path i.e. using the two busses, the barrel shifter and the ALU, with the arguments read from the bank of registers. The micro-architecture's state space is augmented with four components: `mul`, `mul2`, `borrow2` and `mshift`. The implementation also makes use of three latches: `mul1`, `borrow` and `count1`. These latches are set during the first clock phase and their values are then used in the second phase. Cycle t_3 sets up the computation and cycle t_n is repeated until a termination condition is met. Note that square brackets are used to denote bit ranges.

Let W_ℓ represent the set of ℓ-bit words and $\mathbb{B} = \{\top, \bot\}$. The next shift amount for the multiplier register `Rm` is stored in the component `mshift`; this

[2] For $n < 32$, bit $2n-1$ is called the borrow (which is false when $n = 0$), and bits $2n$ and $2n+1$ form the 2-bit value *mul*.

t_3	Fetch an instruction Increment the program counter Set `mul1` to `reg[Rs]` Set `borrow` to false Set `count1` to zero ..... Set `reg[Rd]` to `reg[Rn]` if accumulate, otherwise zero Set `mul` to `mul1[1:0]` Set `mul2` to `mul1[31:2]` Set `borrow2` to `borrow` Set `mshift` to `MSHIFT2(borrow,mul,count1)`
t_n	Set `alub` to `reg[Rm]` shifted left by `mshift` Set `alua` to `reg[Rd]` Set `mul1` to `mul2[29:0]` Set `borrow` to `mul[1]` Set `count1` to `mshift[4:1] + 1` ..... Set `reg[Rd]` to `ALU6`*`(borrow2,mul,alua,alub)` Set `mul` to `mul1[1:0]` Set `mul2` to `mul1[31:2]` Set `borrow2` to `borrow` Set `mshift` to `MSHIFT2(borrow,mul,count1)` Update `NZC` flags of `CPSR` (if `S` flag set) If the last iteration then decode the next instruction

Fig. 4: ARM6 implementation of the multiply instructions. Each cycle is split into two phases. The t_n cycle is repeated until `MULX(mul2,borrow,mshift)` is true. Register `Rd` is not updated when `Rd` is equal to `Rm` or fifteen.

value is defined by the function MSHIFT2 : $\mathbb{B} \times W_2 \times W_4 \to W_5$,

$$\text{MSHIFT2}(\mathit{borrow}, \mathit{mul}, \mathit{count1}) = \\ \mathit{count1} * 2 + \begin{cases} 1, & \text{if } \mathit{borrow} \wedge (\mathit{mul} = 1) \vee \neg \mathit{borrow} \wedge (\mathit{mul} = 2) \\ 0, & \text{otherwise.} \end{cases}$$

Register `Rd` is updated with the output value from the ALU; on the t_n cycles this is given by the function ALU6* : $\mathbb{B} \times W_2 \times W_{32} \times W_{32} \to W_{32}$,

$$\text{ALU6}^*(\mathit{borrow2}, \mathit{mul}, \mathit{alua}, \mathit{alub}) = \\ \begin{cases} \mathit{alua}, & \text{if } \mathit{borrow2} \wedge (\mathit{mul} = 3) \vee \neg \mathit{borrow2} \wedge (\mathit{mul} = 0), \\ \mathit{alua} + \mathit{alub}, & \text{if } \mathit{borrow2} \wedge (\mathit{mul} = 0) \vee (\mathit{mul} = 1), \\ \mathit{alua} - \mathit{alub}, & \text{otherwise.} \end{cases}$$

The termination condition is given by the function MULX : $W_{32} \times \mathbb{B} \times W_5 \to \mathbb{B}$,

$$\text{MULX}(\mathit{mul2}, \mathit{borrow}, \mathit{mshift}) = \\ (\mathit{mul2}[29:0] = 0) \wedge \neg \mathit{borrow} \vee (\mathit{mshift}[4:1] = 15) \ .$$

When this condition holds, register `Rd` has the required value. On the last cycle the next instruction is decoded, ready for execution on the next cycle.

Observe that there is a correspondence between the pseudo-code description of Booth's algorithm (Appendix A) and the ARM6 implementation. The functions MSHIFT2 and ALU6* combine to cover each of the eight cases found in the while loop. The operation `rs div 4` and `rs mod 4` are effectively implemented by selecting bit ranges i.e. with `mul1[31:2]` and `mul1[1:0]` respectively. The component `count1` corresponds with the counter n. Beyond carrying out the multiplication, the processor must also set the NZC flags, update the program counter and maintain a consistent pipeline state.

3.1 HOL Specification

The HOL model of the ARM6 [8] was modified to incorporate the multiply instructions. This involved adding new cases when defining the behaviour of the ALU and barrel shifter. Appendix C presents functions that were entirely new to the specification, and Appendix B defines the primitive bit operations. These functions correspond with some of the operations found in Figure 4 and are used in defining the next state function `NEXT_ARM6`. The functions are made total with the use of `ARB`; the newly added components are of no significance when not executing a multiply instruction.

The initialisation function `INIT_ARM6` is an identity map with respect to the state components `mul`, `mul2`, `borrow2` and `mshift`. These components are set during the t_3 cycle of the multiplication, so they can take *any* value at the end of executing the previous instruction.

4 Correctness

The correctness model has been discussed in technical reports [7, 8]. Principally, one must prove that the following theorem holds:

Commutativity Theorem:

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

where `STATE_ARM_PIPE` is the ISA level state function, `STATE_ARM6` is the micro-architecture model, `IMM_ARM6` is a uniform immersion (temporal abstraction map) with duration function `DUR_ARM6`, and `ABS_ARM6` is a data abstraction.

The data abstraction is, as before, essentially a projection [8]. The duration map `DUR_ARM6` must now specify how long it takes to execute a multiply instruction. The duration is:

```
1 + MLA_MUL_DUR (REG_READ6 reg nbs (WORD_BITS 11 8 ireg))
```

where `nbs` is the current processor mode, and

```
⊢def MLA_MUL_DUR rs = LEAST n. MLA_MUL_DONE rs n
⊢def MLA_MUL_DONE rs n =
        ¬(n = 0) ∧ (WORD_BITS HB (2 * n) rs = 0) ∧ ¬BORROW2 rs n ∨
        ¬(2 * n < WL)
```

Constant `HB` is thirty-one and `WL` is thirty-two. The function `MLA_MUL_DUR` gives the number of t_n cycles; this is the least cycle (less than or equal to sixteen) such that the `MULX` condition becomes true. The total duration includes the single t_3 cycle, thus giving a value in the range two to seventeen. Note that the instruction timing is purely dependent upon the value of the multiplicand `rs`.

One of the advantages of using an immersion is that it can be used when simulating the processor; it gives the number of cycles needed to execute any given number of instructions. However, the duration cannot be readily evaluated using the HOL conversions `EVAL` and `CBV_CONV` because the `LEAST` operator will cause non-termination. Instead the definition is expanded as follows:

```
⊢ MLA_MUL_DUR (n2w rs) =
     if BITS 31 1 rs = 0 then 1 else
     if BITS 31 3 rs = 0 then 2 else
     if BITS 31 5 rs = 0 then 3 else
       ..
     if BITS 31 27 rs = 0 then 14 else
     if BITS 31 29 rs = 0 then 15 else 16
```

Here `n2w` maps natural numbers to 32-bit words (`w2n` maps in the other direction). If the value of register `Rs` is known then this theorem can be used to give the number of cycles needed to execute the multiply instruction.

5 Formal Verification

The formal verification of correctness is based on the use of one-step theorems [7]. These theorems are used to eliminate the universal quantification over time in the commutativity theorem (Section 4) i.e. it is sufficient to verify the following four theorems:

```
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)
```

Theorems 1 and 3 are trivial to verify and so the main proof effort lies in determining the state of the processor after executing one instruction i.e. at the cycle `IMM_ARM6(a)(1)`.

The verification of Theorems 2 and 4 proceeds with case splitting over the instruction class. The instruction classes that had already been covered were verified using the existing proof scripts – the approach is quite robust with respect to changes in the design. The case when the instruction class is `mla_mul` is tackled independently.

The state of the processor after executing an instruction is given by:

```
FUNPOW NEXT_ARM6 (DUR_ARM6 (INIT_ARM6 a)) (INIT_ARM6 a)
```

where `FUNPOW(f)(n)(a)` is function iteration i.e. $f^n(a)$. With most instruction classes the duration is a constant value and so the proof can proceed by expanding with the definition of `NEXT_ARM6`. The duration for multiply instructions is a function of the multiplicand, therefore a different approach must be used.

The processor state after the first cycle (t_3) can be determined; that is, one can rewrite using the definitions of `INIT_ARM6`, `DUR_ARM6`, `FUNPOW` and `NEXT_ARM6`. To proceed further one must construct a theorem of the form:

Multiply t_n: `⊢ ∀n. n ≤ MLA_MUL_DUR rs ⇒ (FUNPOW NEXT_ARM6 i x = X_n)`

where `x` represents the state after the t_3 cycle and $\mathtt{X}_\mathtt{n}$ is the state after the n^{th} following cycle.[3] When this theorem is specialised, with n taking the value `MLA_MUL_DUR(rs)`, it is relatively straightforward to complete the proofs for Theorems 2 and 4. The term $\mathtt{X}_\mathtt{n}$ was constructed manually and the theorem above is verified by induction on the variable `n`. Of most significance is the state of the

Table 1: Selected data and control component values for cycle `n`.

reg	if (Rd = 15) ∨ (Rd = Rm) then
	  REG_WRITE reg nbs 15 (pc + n2w 4)
	else
	  REG_WRITE (REG_WRITE reg nbs 15 (pc + n2w 4)) nbs Rd
	  (RD_INVARIANT (WORD_BIT 21 ireg) rm rs rn n)

mul	BITS 1 0 (WORD_BITS HB (2 * n) rs)
mul2	WORD_BITS HB (2 * (n + 1)) rs
borrow2	BORROW2 rs n
mshift	MSHIFT2 borrow2 mul (BITS 3 0 n)

four new control components and the state of register bank – in particular, the

[3] The actual theorem is too unwieldy to present here i.e. `x` and $\mathtt{X}_\mathtt{n}$ are large terms.

value of register Rd – see Table 1. After cycle t_3, the program counter has been incremented. If the destination register (Rd) is fifteen or Rm, then the register bank is not modified; there is no early termination for these cases. Otherwise, the value of register Rd is given by the function RD_INVARIANT, which is defined as follows:

```
⊢def RD_INVARIANT A rm rs rn n =
       (if BORROW2 rs n then
          rm * n2w (w2n rs MOD 2^(2 * n)) - rm << (2 * n)
        else
          rm * n2w (w2n rs MOD 2^(2 * n))) + if A then rn else word_0
```

If there is not a borrow then Rd stores the result of multiplying rm by the first $2n$ bits of the multiplicand rs, and if A is true then rn is also added. If there is a borrow then this partial product can be obtained by adding the value of rm shifted left by $2n$ places. There is not a borrow when n is zero. On the last cycle:

Rd Last:

```
⊢ ∀A rm rs rn.
    RD_INVARIANT A rm rs rn (MLA_MUL_DUR rs) = rm * rs + (if A then rn else word_0)
```

This means that when the MULX condition becomes true, the value of register Rd conforms with the ISA specification i.e. the multiplication is complete.

In verifying the theorem *Multiply* t_n by induction, it was relatively easy to establish that the ARM6 gives the right values for the four control components in Table 1. In order to verify that register Rd has the right value, the following lemma is used:

Rd Induction Step:

```
⊢ ∀n a rm rs rn.
    RD_INVARIANT A rm rs rn (n + 1) =
      (let borrow2 = BORROW2 rs n and
           mul = WORD_BITS (2 * n + 1) (2 * n) rs
       in
         ALU6_MUL borrow2 mul (RD_INVARIANT A rm rs rn n) (rm << MSHIFT2 borrow2 mul n))
```

Here ALU6_MUL corresponds with ALU6* from page 7. This theorem shows that the next value for register Rd is correctly given by our ALU definition. The ALU must be supplied with the appropriate values; for example, alua holds the current value of register Rd, and alub holds the value of register Rm shifter left according to the definition of MSHIFT2. There are sixteen cases to consider, covering all possible values for the variables A, borrow2 and mul. Each sub-goal is discharged using word arithmetic theorems; one case is illustrated with an informal proof in Appendix D.

In effect, *Rd Induction Step* and *Rd Last* provide a proof of the correctness for the algorithm presented in Appendix A. These lemmas hide (abstract out) many of the control and data path details from the ARM6 model. Theorem *Multiply* t_n shows that the ARM6 implements this algorithm, but it also gives the values for all the other control and data path components. In particular, it is necessary to establish the state of the pipeline and the state of the CPSR, which may have the condition code flags updated. Also, one must show that the function MULX becomes true at the time given by the duration function.

6 Summary

The ARM's multiply instructions have a fairly simple semantics (Figure 2), but the ARM6 implementation is quite unusual. With previous processor verifications data operations have been implemented by the ALU, and this provides a convenient abstraction mechanism. With the ARM6 implementation, the semantics of multiplication is inseparable from the processor's control logic. To be faithful to the ARM6 micro-architecture it was necessary to verify a modified form of Booth's algorithm. To execute a multiply instruction, the processor's next state function must be iterated up to seventeen times; this corresponds with the loop part of the algorithm in Appendix A.

By virtue of this unusual implementation, the duration function (used to specify the instruction timing) employs the `LEAST` operator, with the duration dependent upon the value of a 32-bit general purpose register, as opposed to being a simple function of the instruction op-code. This has not introduced any significant difficulty: the definition can be expanded to facilitate simulation, and HOL provides support for reasoning under the semantics of `LEAST`.

The formal verification hinges around verifying a theorem *Multiply* t_n, which gives the processor's state after each t_n cycle (Figure 4). This theorem was manually constructed and then checked by induction – making use of the lemma *Rd Induction Step*. This theorem is then specialised with the length of the t_n stage and lemma *Rd Last* shows that the final value of register `Rd` is as required. The two lemmas abstract out many of the details of the ARM6 implementation, focusing purely on the workings of the Booth's algorithm. The HOL theories of bits and n-bit words were used to reason about the word arithmetic. The proof run-times are independent of the word length and so the thirty-two bit word length was not a problem. This is one of the advantages of using a theorem proving approach when compared with model checking.

Although it is atypical for an ALU not to implement multiplication, modelling and verifying this instruction class was a worthwhile exercise. All of the ARM6 instructions have now been verified.

References

1. Mark D. Aagaard, Byron Cook, Nancy A. Day, and Robert B. Jones. A framework for microprocessor correctness statements. In *CHARME 2001*, volume 2144 of *LNCS*, pages 433–448. Springer, 2001.
2. Bruno Barras. Programming and computing in HOL. In Mark Aagaard and John Harrison, editors, *TPHOLs 2000*, volume 1869 of *LNCS*, pages 17–37. Springer, 2000.
3. 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.

4. 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.
5. 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.
6. Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, June 1989.
7. Anthony Fox. An algebraic framework for modelling and verifying microprocessors using HOL. Technical Report 512, University of Cambridge, Computer Laboratory, April 2001.
8. Anthony Fox. Formal verification of the ARM6 micro-architecture. Technical Report 548, University of Cambridge Computer Laboratory, November 2002.
9. Anthony Fox. Verifying the ARM block data transfer instructions. Technical report to be presented at DCC 2004, March 2004.
10. Steve Furber. *ARM: system-on-chip architecture*. Addison-Wesley, second edition, 2000.
11. Brian T. Graham. *The SECD Microprocessor, A Verification Case Study*. Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1992.
12. 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.
13. 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.
14. Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor*, volume 795 of *LNCS*. Springer-Verlag, 1994.
15. 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.
16. 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.
17. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.
18. Panagiotis Manolios. Correctness of pipelined machines. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design, FMCAD 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 161–178. Springer-Verlag, 2000.
19. 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.
20. 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.

21. 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.
22. David Seal, editor. *ARM Architectural Reference Manual*. Addison-Wesley, second edition, 2000.
23. 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 Modified Booth's Algorithm

```
Algorithm
  Modified Booth's Algorithm (with accumulator)

Constants
  wl : number  -- the word length

Input
  rm, rs, rn : wl-bit word
  accumulate : bool

Output
  rd : wl-bit word -- this could just as well have length 2 * wl
  rd = if accumulate then rm * rs + rn else rm * rs

Variables
  borrow : boolean
  n : number
  mul : 2-bit word

Pseudo code
  borrow := false
  n := 0
  rd := if accumulate then rn else 0
  while (not (rs = 0) or borrow) and n < wl do
    begin
      mul = rs mod 4
      if not borrow then
        case mul
        of 0 => rd := rd
        |  1 => rd := rd + rm << (2 * n)
        |  2 => rd := rd - rm << (2 * n + 1); borrow := true
        |  3 => rd := rd - rm << (2 * n);     borrow := true
      else
        case mul
        of 0 => rd := rd + rm << (2 * n);     borrow := false
        |  1 => rd := rd + rm << (2 * n + 1); borrow := false
        |  2 => rd := rd - rm << (2 * n)
        |  3 => rd := rd
      rs := rs div 4
      n := n + 1
    end{while}
```

Code kindly supplied by Daniel Schostak. (ARM Ltd.)

B 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)
```

C Addendum to the ARM6 Specification

```
⊢def MSHIFT2 borrow mul n =
        n * 2 + if borrow ∧ (mul = 1) ∨ ¬borrow ∧ (mul = 2) then 1 else 0

⊢def MSHIFT ic borrow mul count1 =
        if ic = mla_mul then MSHIFT2 borrow mul count1 else ARB

⊢def BORROW2 rs n = ¬(n = 0) ∧ WORD_BIT (2 * n - 1) rs

⊢def BORROW ic is mul =
        if ic = mla_mul then if is = t3 then F else BIT 1 mul else ARB

⊢def COUNT1 ic is mshift =
        if ic = mla_mul then
          if is = t3 then 0 else BITS 3 0 (mshift DIV 2 + 1)
        else
          ARB

⊢def MUL1 ic is ra mul2 =
        if ic = mla_mul then
          if is = t3 then w2n ra else BITS (HB - 2) 0 mul2
        else
          ARB

⊢def MULZ ic is mul2 =
        if (is = tn) ∧ (ic = mla_mul) then
          BITS (HB - 2) 0 mul2 = 0
        else
          ARB

⊢def MULX ic is mulz borrow mshift =
        if (is = tn) ∧ (ic = mla_mul) then
          mulz ∧ ¬borrow ∨ (mshift DIV 2 = 15)
        else
          ARB
```

D Informal proof: one case of *Rd Induction Step*

Goal:

$$\begin{aligned}\vdash \forall A\ rm\ rs\ rn\ n.\ &\mathrm{RD\_INVARIANT}(A, rm, rs, rn, n+1) = \\ &\mathrm{ALU6}^*(\mathit{borrow2}, \mathit{mul}, \mathrm{RD\_INVARIANT}(A, rm, rs, rn, n), \\ &\qquad rm \ll \mathrm{MSHIFT2}(\mathit{borrow2}, \mathit{mul}, n))\end{aligned}$$

where $\mathit{borrow2} = \mathrm{BORROW2}(rs, n)$ and $\mathit{mul} = rs[2n+1 : 2n]$. Note that, for the purposes of this proof, n and the shift amount are assumed to be natural numbers.

Case: $\neg A$, BORROW2(rs, n) *and* $rs[2n+1:2n] = 1$

Expanding the RHS gives:

$$\begin{aligned}
&\text{RD_INVARIANT}(\bot, rm, rs, rn, n) + rm \ll \text{MSHIFT2}(\top, 1, n) \\
&\quad = (rm * \text{n2w}(\text{w2n } rs \bmod 2^{2n}) - rm \ll 2n + \text{word_0}) + rm \ll (2n+1) \\
&\quad = rm * \text{n2w}(\text{w2n } rs \bmod 2^{2n}) - rm \ll 2n + rm \ll 2n * \text{n2w } 2 \\
&\quad = rm * \text{n2w}(\text{w2n } rs \bmod 2^{2n}) + rm \ll 2n \ .
\end{aligned}$$

Expanding the LHS gives:

$$\begin{aligned}
&\text{RD_INVARIANT}(\bot, rm, rs, rn, n+1) \\
&\quad = rm * \text{n2w}(\text{w2n } rs \bmod 2^{2(n+1)}) + \text{word_0} \\
&\quad = rm * \text{n2w}(\text{w2n } rs \bmod 2^{2(n+1)}) \ .
\end{aligned}$$

N.B. $(rs[2n+1:2n] = 1) \Rightarrow \neg\text{BORROW2}(rs, n+1)$.

We now make use of the following theorem:

$$\begin{aligned}
\vdash \forall n\, a\, b.\ & a * \text{n2w}(\text{w2n } b \bmod 2^{2(n+1)}) = \\
& a * \text{n2w}(\text{w2n } b \bmod 2^{2n}) + (a \ll 2n) * \text{n2w}(b[2n+1:2n]) \ .
\end{aligned}$$

LHS:

$$\begin{aligned}
&rm * \text{n2w}(\text{w2n } rs \bmod 2^{2(n+1)}) \\
&\quad = rm * \text{n2w}(\text{w2n } rs \bmod 2^{2n}) + (rm \ll 2n) * \text{n2w}(rs[2n+1:2n]) \\
&\quad = rm * \text{n2w}(\text{w2n } rs \bmod 2^{2n}) + (rm \ll 2n) * \text{n2w } 1 \\
&\quad = rm * \text{n2w}(\text{w2n } rs \bmod 2^{2n}) + rm \ll 2n \ .
\end{aligned}$$

□