Specification and verification of the ARM6 microprocessor in HOL

Written report for the seminar
"State of the Art of Formal Hardware Verification"

Oleg Parshin

January 2004

Abstract

This report describes the formal verification of the ARM6 micro-architecture using the HOL theorem prover [5]. 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 behavior 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.

Contents

1 Introduction

This report describes formal verification of the ARM6 microprocessor. The approach used for the formal verification is based on work done at Swansea [2], which has been formalized in HOL at Cambridge [3]. This approach provides a general and structured framework for carrying out processor verifications. The verification project described in this report aimed to apply these methods to a commercial processor design and, in doing so, to assess the suitability of HOL for this task.

The HOL proof tool is briefly described in Section 2. Section 3 presents the algebraic approach to processor verification, which was developed at Swansea and used for the ARM6 verification. 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 architecture is given in Section 4. The HOL specification of this architecture is documented in [4]. The ARM6 micro-architecture is described in Section 5. The formal verification is then discussed in Section 6.

2 About HOL Proof Tool

The HOL System is an environment for interactive theorem proving in a *higher-order logic*. Its most outstanding feature is its high degree of programmability through the *meta-language* ML. The system has a wide variety of uses from formalizing pure mathematics to verification of industrial hardware. Academic and industrial sites world-wide are using HOL.

The HOL system is designed to support interactive theorem proving in higher order logic (hence the acronym "HOL"). To this end, the formal logic is interfaced to a general purpose programming language (ML, for meta-language) in which terms and theorems of the logic can be denoted, proof strategies expressed and applied, and logical theories developed. The version of higher order logic used in HOL is predicate calculus with terms from the typed lambda calculus (i.e. simple type theory). This was originally developed as a foundation for mathematics [1]. The primary application area of HOL was initially intended to be the specification and verification of hardware designs (the use of higher order logic for this purpose was first advocated by Keith Hanna [7]). However, the logic does not restrict applications to hardware; HOL has been applied to many other areas.

The approach to mechanizing formal proof used in HOL is due to Robin Milner [6], who also headed the team that designed and implemented the language ML. That work centered on a system called LCF (logic for com-

putable functions), which was intended for interactive automated reasoning about higher order recursively defined functions. The interface of the logic to the meta-language was made explicit, using the type structure of ML, with the intention that other logics eventually be tried in place of the original logic. The HOL system is a direct descendant of LCF; this is reflected in everything from its structure and outlook to its incorporation of ML, and even to parts of its implementation. Thus HOL satisfies the early plan to apply the LCF methodology to other logics.

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 [3].

3.1 Approach

This section formalizes, 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 level of data and temporal abstraction. Correctness require there to be a correspondence in behavior between these two models. The MA and ISA are modelled using *state functions*; these are maps of the form $f : time \rightarrow state \rightarrow state$. 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 behavior 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

[1]The initialization function $f(0)$ maps arbitrary, pre-initialized states to valid initial states.

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.

We will denote the state space of the implementation by A and state space of the specification by B. Therefore, the state function *Impl* (modelling the MA) has type

$$Impl : \mathbb{N} \to A \to A,$$

and state function *Spec* (modelling the ISA) has type

$$Spec : \mathbb{N} \to B \to B.$$

Initialization functions for the implementation and specification we denote by $Init'$ and *Init*, respectively:

$$Init = Spec(0) : B \to B,$$

$$Init' = Impl(0) : A \to A.$$

Initialization functions are used to map *arbitrary* states from the state-space to *valid* initial states. For example, the pipeline of a pipelined processor cannot be filled arbitrarily because each pipeline stage (such as, fetch, decode and execute) is expected to be working on consecutive instructions. A trivial initialization function for such a pipelined design would simply ensure that the pipeline was empty (flushed), ready to be filled with instructions. A more complex initialization function may, instead of always resetting the pipeline, preserve states for which the pipeline is correctly filled. Therefore, the initialization function is not simply the "reset" function, which returns starting state of specification.

The state spaces of the implementation and specification are related using a data abstraction *Abs* of type

$$Abs : A \to B.$$

The implementation must be complete with respect to the specification i.e. each initial state of the specification must be an abstraction of an initial implementation state. This condition is checked with the following predicate:

Definition 1 *ONTO_INIT predicate.*

$$ONTO\_INIT(Abs)(Init)(Init') \stackrel{\text{def}}{=} \forall b \in B\ \exists a \in A : Abs(Init'(a)) = Init(b).$$

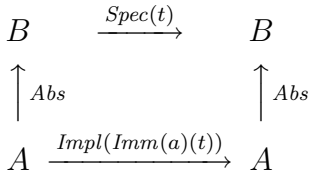

Figure 1: Commutative diagram for the correctness condition

The *clocks* of two systems are related using temporal abstraction (which is called *immersion*) *Imm* of type

$$Imm : A \to \mathbb{N} \to \mathbb{N}.$$

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-initialized state of the implementation.

For checking, if *Imm* is an immersion, we define the predicate *IMMERSION*.

Definition 2 *IMMERSION predicate.*

$$IMMERSION(Imm) \stackrel{\text{def}}{=} (\forall a \in A : Imm(a)(0) = 0) \wedge$$
$$\wedge (\forall a \in A\, \forall t_1, t_2 \in \mathbb{N} : t_1 < t_2 \Rightarrow Imm(a)(t_1) < Imm(a)(t_2))$$

Now we can formally define correctness of the implementation: A state function $Impl : \mathbb{N} \to A \to A$ is a *correct implementation* of a specification $Spec : \mathbb{N} \to B \to B$ with respect to an immersion $Imm : A \to \mathbb{N} \to \mathbb{N}$ and data abstraction $Abs : A \to B$ if, and only if, for all $t \in \mathbb{N}$ and $a \in A$

$$Spec(t)(Abs(a)) = Abs\left(Impl(Imm(a)(t))(a)\right).$$

This correctness condition is illustrated with the commutative diagram in Figure 1.

This condition ensures that after executing t instructions 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*. By the given definition of correctness, the data abstraction relationship (between the states of the implementation and specification) is allowed to break down at cycles between $Imm(a)(t+1)$ and $Imm(a)(t)$. This need not matter, provided one is aware that correctness only refers to implementation states occurring at cycles in the the range of *Imm*.

The correctness condition is checked with the *CORRECT* predicate.

Definition 3 *CORRECT predicate.*

$$CORRECT(Spec)(Impl)(Imm)(Abs) \stackrel{\text{def}}{=}$$
$$IMMERSION(Imm) \wedge ONTO\_INIT(Abs)(Spec(0))(Impl(0)) \wedge$$
$$\wedge (\forall a \in A\ \forall t \in \mathbb{N} : Spec(t)(Abs(a)) = Abs\,(Impl(Imm(a)(t))(a)))$$

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:

1. Immersion should be *uniform* with respect to implementation, i.e. each interval $Imm(a)(t+1) - Imm(a)(t)$ is a function of the implementation's state a time $Imm(a)(t)$. This is expressed with the following definition:

   Definition 4 *UNIFORM predicate.*

   $$UNIFORM(Imm)(Impl) \stackrel{\text{def}}{=} \exists (Dur : A \to \mathbb{N}) :$$
   $$(\forall a \in A : Dur(a) > 0) \wedge (\forall a \in A : Imm(a)(0) = 0)$$
   $$(\forall a \in A\ \forall t \in \mathbb{N} : Imm(a)(t+1) = Imm(a)(t)) +$$
   $$Dur(Impl(Imm(a)(t))(a))$$

2. State function for the specification should be *time-consistent.*

   Definition 5 *Time-consistency.*

   $$TCON(Spec) \stackrel{\text{def}}{=} \forall b \in B\ \forall t_1, t_2 \in \mathbb{N} :$$
   $$Spec(t_1 + t_2)(b) = Spec(t_2)(Spec(t_1)(b))$$

3. State function for the implementation should be time-consistent with respect to immersion. This is expressed with the following definition:

   Definition 6 *Time-consistency with respect to an immersion*

   $$TCON\_IMM(Impl)(Imm) \stackrel{\text{def}}{=} \forall a \in A\ \forall t_1, t_2 \in \mathbb{N} :$$
   $$Impl(t'_1 + t'_2)(a) = Impl(t'_2)(Impl(t'_1)(a)),$$
   where
   $$t'_1 = Imm(a)(t_1),\ t'_1 = Imm(Impl(t'_1)(a))(t_2).$$

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

Note, that in Definition 6 initialization is only required to be an identity map at times given by the immersion.

If immersion *Imm* is uniform and state function *Spec* and *Impl* are time-consistent (i.e. predicates defined in Definitions 4, 5 and 6 hold), we can use for the verification following theorem:

Theorem 1 *One-step theorem for correctness*

$$
\begin{array}{l}
\forall Spec\, \forall Impl\, \forall Imm\, \forall Abs : \\
(UNIFORM(Imm)(Impl) \wedge TCON(Spec) \wedge TCON\_IMM(Impl)(Imm)) \Rightarrow \\
\quad CORRECT(Spec)(Impl)(Imm)(Abs) = \\
\qquad ONTO\_INIT(Abs)(Spec(0))(Impl(0))) \wedge \\
\qquad (\forall a \in A : Spec(0)(Abs(a)) = Abs(Impl(Imm(a)(0))(a))) \wedge \\
\qquad (\forall a \in A : Spec(1)(Abs(a)) = Abs(Impl(Imm(a)(1))(a)))
\end{array}
$$

This theorem reduces the verification of correctness to a goal in which t has been specialized 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 (*TCON*, *TCON_IMM* and *UNIFORM*) require them to be deterministic.

Proof of this theorem see in [8].

Theorem 1 would be of little use if were not possibly to verify the *TCON_IMM* property without resorting to an explicit induction over time. The following theorem is used:

Theorem 2 *One-step theorem for time-consistency*

$$
\begin{array}{l}
\forall Impl\, \forall Imm : (IMAP(Impl) \wedge UNIFORM(Imm)(Impl)) \Rightarrow \\
\quad TCON\_IMM(Impl)(Imm) = \\
\qquad (\forall a \in A : Impl(0)(Impl(Imm(a)(0))(a)) = Impl(Imm(a)(0))(a)) \wedge \\
\qquad (\forall a \in A : Impl(0)(Impl(Imm(a)(1))(a)) = Impl(Imm(a)(1))(a))
\end{array}
$$

Here *IMAP* is the predicate which shows that state function is an iterated map:

Definition 7 *Iterated map*

$$IMAP(Impl : \mathbb{N} \to A \to A) \stackrel{\text{def}}{=}$$
$$\exists(Init' : A \to A)\, \exists(Next' : A \to A) :$$
$$(\forall a \in A : Impl(0)(a) = Init'(a)) \wedge$$
$$(\forall a \in A\, \forall t \in \mathbb{N} : Impl(t+1)(a) = Next'(Impl(t)(a)))$$

Proof of Theorem 2 see in [8].
Therefore, in order to verify a processor using this approach we have to:

- Define
  - State function for specification (*Spec*),
  - State function for implementation (*Impl*),
  - Data abstraction (*Abs*) and
  - Temporal abstraction (immersion) (*Imm*). And then
- Prove
  - Uniformity of immersion,
  - Completeness of data abstraction,
  - Time-consistency of state functions, and then we can prove
  - Correctness of processor.

4 The ARM Instruction Set Architecture Specification

The HOL specification of the ARM instruction set is described in [4]. This specification was used as the basis for the ISA model used in the ARM6 verification, but a number of modifications have been made [5].

The ARM microprocessor has 37 32-bit general purpose registers (GPRs), six 32-bit program status registers (PSRs) and main memory. General purpose register *r15* is the *Program Counter* (*PC*).

4.1 State-space

The ARM state-space *state_ARM* is a triple

$$\langle mem, reg, psr \rangle,$$

which consists of the following components:

- $mem : word30 \rightarrow word32$ is the main memory. The main memory modelled as mapping from 30-bit addresses ($word30$) to 32-bit values ($word32$). Here $word30$, $word32$ are types created using the binary word theory, supplied with HOL.
- $reg : regs \rightarrow word32$ is the general purpose register file. Here $regs$ is the type containing names of GPRs.
- $psr : psrs \rightarrow word32$ is the program status registers bank. Here $psrs$ is the type containing names of PSRs.

4.2 State Function

State function $STATE\_ARM$ of the ARM ISA specification is defined with the following equations:

$$\begin{aligned} STATE\_ARM(0)(b) &= b \\ STATE\_ARM(t+1)(b) &= NEXT\_ARM(STATE\_ARM(t)(b)), \end{aligned}$$

where

- $b \in state\_ARM$ is the initial state of the specification.
- $t \in \mathbb{N}$ is the specification time ($instructions$).
- $NEXT\_ARM : state\_ARM \rightarrow state\_ARM$ is the next state function of the specification.

There is no initialization function for the specification (all configurations are valid initial configurations), i.e. initialization function can be assumed to be the identity map.

The next state function $NEXT\_ARM$ computes the next state of the specification. It takes as argument current state of the specification, and returns state after executing one instruction. This function fetches the instruction word from the main memory by the address stored in register $r15$ (PC). Then it computes new state of the specification according to the fetched instruction.

5 The ARM6 Implementation

The ARM6 design is split into control and data path components. The state-space of the data path is presented in Section 5.1.1 and the state-space of the control unit is introduced in Section 5.1.2.

5.1 State-space

5.1.1 Data Path

The data path performs various data processing operations throughout the course of an instruction's execution. Figure 2 shows the (simplified) 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 to the shifter. The behavior of the data path is determined by the control unit.

The state-space of the ARM6 data path *dp* is a 6-tuple

$$\langle reg, psr, areg, din, alua, alub \rangle,$$

which consists of the following components:

- $reg : regs \to word32$ is the general purpose register file. Here $regs$ is the type containing names of GPRs.
- $psr : psrs \to word32$ is the program status registers bank. Here $psrs$ is the type containing names of PSRs.
- $areg : word32$ is the address register. This register contains the memory address for memory accesses.
- $din : word32$ is the data input register for memory reads.
- $alua,\ alub : word32$ are registers with operands for the ALU.

The register bank and program status register types are taken from the ISA model (see Chapter 4). 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 contains the address of the instruction being fetched.

The main memory is accessed using the address register *areg*, which is the source/destination address for memory reads/writes. The address register can take one of the following values: the value of register *r15*, the output of the ALU or the output of the address incrementer. When writing to memory the data source is always bus B. When reading from memory the data is fed to the pipeline (see Section 5.1.2) and to the register *din*. The register *din* can either remain unchanged, take the value of the current instruction code (after instruction fetch) or be updated with a word from memory.

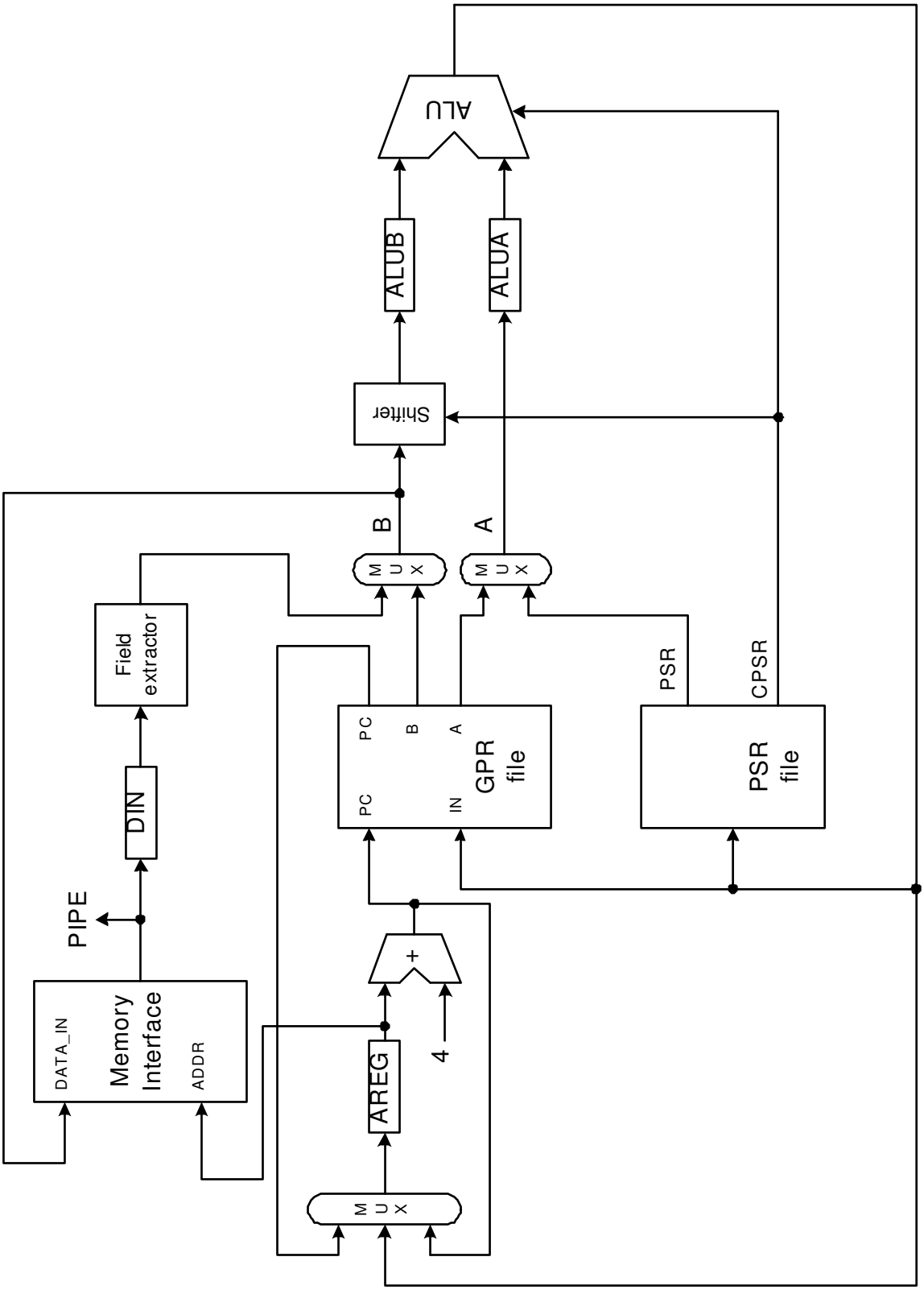

Figure 2: The ARM6 Data Path.

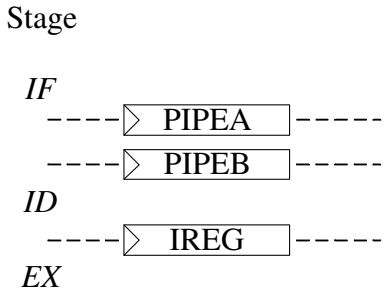

Figure 3: The ARM6 pipeline representation

5.1.2 The Control Unit

The control unit stores the state of the pipeline and controls the behavior of the data path.

The state-space of the ARM6 control unit *ctrl* consists of the 19 components, below are the most important components:

- $pipea : word32$ — stores the 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 pipeline representation with these components is shown on Figure 3.

The pipeline has three stages: fetch, decode and execute. A *single-stage* 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.

Instructions are fetched from the current program counter *PC*, which is register *r15*. Therefore, *PC* is *not* the address of the instruction being *executed*. In general, the instruction being executed is from the address $PC - 8$

and the instruction being decoded is from the address $PC - 4$. This relation holds only for single-cycle instructions. With multi-cycle instructions this relation holds only for the first cycle of the execution, since PC is incremented in the first execute cycle.

The *pipeb* component is needed for multi-cycle instructions, since pipeline stops while multi-cycle instruction is being executed (is in the execute stage). For single-cycle instructions the fetch and decode stage always performed simultaneously, and *pipea* and *pipeb* registers store the same instruction code.

Components *pipeaval*, *pipebval* and *iregval* are used to implement the re-filling of the pipeline in case of writing to the register *r15*.

Branch instruction takes three cycles in the execute stage in order to properly re-fill pipeline after jump (two cycles are idle).

The *apipea* and *apipeb* components are used for catching the overwriting already fetched and decoded instructions (self-modification of code).

5.1.3 Implementation State-space

The ARM6 implementation state-space $state\_ARM6$ is a triple

$$\langle mem, dp, ctrl \rangle,$$

which consists of the following components:

- $mem : word30 \rightarrow word32$ is the main memory. This component is taken from the ARM ISA specification (Section 4.1)
- dp is the data path state space (described in Section 5.1.1)
- $ctrl$ is the control logic state-space (described in Section 5.1.2).

5.2 State function

State function *STATE_ARM6* of the ARM6 micro-architecture is defined with the following equations:

$$\begin{aligned} STATE\_ARM6(0)(a) &= INIT\_ARM6(a) \\ STATE\_ARM6(t+1)(a) &= NEXT\_ARM6(STATE\_ARM6(t)(a)), \end{aligned}$$

where

- $a \in state\_ARM6$ is the initial state of the specification.
- $t \in \mathbb{N}$ is the implementation time (*cycles*).

- $INIT\_ARM6 : state\_ARM6 \rightarrow state\_ARM6$ is the initialization function of the implementation.
- $NEXT\_ARM6 : state\_ARM6 \rightarrow state\_ARM6$ is the next state function of the implementation.

The initialization function $INIT\_ARM6$ takes an arbitrary configuration with pre-initialized *mem*, *reg* and *psr* components, and returns valid initial configuration with correctly initialized pipeline:

$$\begin{aligned} areg &= r15 \\ din &= mem[r15 - 8] \\ pipea &= mem[r15 - 4] \\ pipeaval &= true \\ pipeb &= mem[r15 - 4] \\ pipebval &= true \\ ireg &= mem[r15 - 8] \\ iregval &= true \end{aligned}$$

Note, that initialization function does not change the three ISA components (*mem*, *reg* and *psr*). This ensures $INIT\_ARM6(a) = a$ whenever possible. For complex microprocessor designs the definition of the initialization function, which preserves valid initial states can be a non-trivial task.

The ARM6 pipeline after initialization is presented on the Figure 4:

- instruction I_i is decoded and is about to be executed,
- instruction I_{i+1} is fetched and is about to be decoded and
- instruction I_{i+2} is the next instruction to be fetched ($areg = r15$).

The next-state function $NEXT\_ARM6$ computes the next state of the implementation, it takes as argument current state of the implementation, and returns state after one cycle of execution. Note, that for executing one *instruction* on the ISA side we need several *cycles* on the MA side.

5.3 Data Abstraction

Data abstraction is the projection from the implementation state space to the specification state space:

$$ABS\_ARM6 : state\_ARM6 \rightarrow state\_ARM.$$

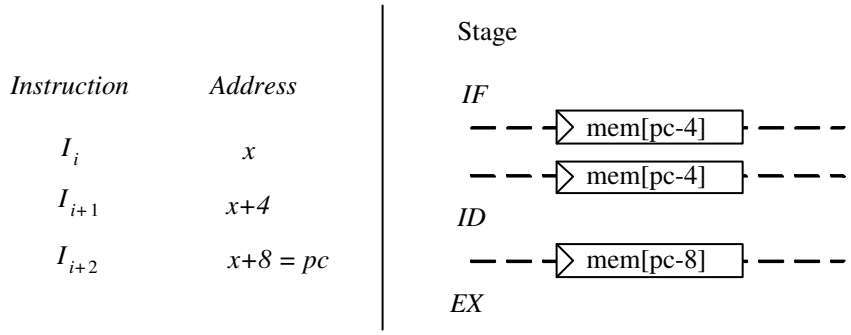

Figure 4: The ARM6 pipeline after initialization

For $a \in state\_ARM6$, $b \in state\_ARM$, $b = ABS\_ARM6(a)$ it is defined by the following equations:

$$\begin{aligned} b.mem &= a.mem \\ b.psr &= a.psr \\ b.reg.rx &= \begin{cases} a.dp.reg.rx - 8, & \text{if } rx = r15; \\ a.dp.reg.rx, & \text{otherwise.} \end{cases} \end{aligned}$$

The notation "$b.mem$" means "component mem of the b", and so on.

The case splitting in the third equation is needed because of different treating of the program counter ($r15$) on the different levels of abstraction (see Chapter 5.1.1). The illustration of this reader can find on the left part of Figure 4.

5.4 Temporal Abstraction

The temporal abstraction is needed to relate the number of cycles (on the MA side) with the number of instructions (on the ISA side).

At first, we define auxiliary *duration* function

$$DUR\_ARM6 : state\_ARM6 \to \mathbb{N}.$$

This function takes a state of the ARM6 implementation and returns the number of cycles required for the execution of the current instruction (i.e., the instruction which is pointed by the program counter).

Using the duration function we define immersion: a map from the specification's clock to the implementation's clock:

$$IMM\_ARM6 : state\_ARM6 \to \mathbb{N} \to \mathbb{N}.$$

The immersion is defined with the following equations:

$$\begin{aligned} IMM\_ARM6(a)(0) &= 0 \\ IMM\_ARM6(a)(t+1) &= DUR\_ARM6(STATE\_ARM6(IMM\_ARM6(a)(t))(a)) \\ &\quad + IMM\_ARM6(a)(t) \end{aligned}$$

where

- $a \in state\_ARM6$ is the initial state of the implementation.
- $t \in \mathbb{N}$ is the specification time (*instructions*).
- $STATE\_ARM6$ is the state-function of the implementation.

The immersion gives the number of the first cycle of the *execution* of the instruction number t.

6 The ARM6 Verification

After defining state functions for the specification and implementation, as well as the data and temporal abstraction, we are ready to proceed to the verification.

As it was described in Section 3, in order to verify the microprocessor design, we have to prove the following:

- Uniforminty of immersion (Section 6.1).
- Completeness of data abstraction (Section 6.2).
- Time-consistency of state functions (Section 6.3).
- Correctness (Section 6.4).

6.1 Uniformity

We have to prove:

$$UNIFORM(IMM\_ARM6)(STATE\_ARM6),$$

i.e.:

$$\begin{aligned} &\exists(Dur : state\_ARM6 \to \mathbb{N}) : \\ &\quad (\forall a \in A : Dur(a) > 0) \wedge \\ &\quad (\forall a \in state\_ARM6 : IMM\_ARM6(a)(0) = 0) \\ &\quad (\forall a \in state\_ARM6\ \forall t \in \mathbb{N} : IMM\_ARM6(a)(t+1) = \\ &\qquad IMM\_ARM6(a)(t) + \\ &\qquad Dur(STATE\_ARM6(IMM\_ARM6(a)(t))(a))) \end{aligned}$$

Proof. In order to prove existence, we must provide one example. The duration function DUR_ARM6 is used as the witness. It satisfies to the conditions for immersion by definition of immersion IMM_ARM6, and we only have to show that this function is always greater than zero. This is done automatically by the so-called *simplification* tactic in HOL. This tactic expands the definition of the DUR_ARM6 function and checks all cases. Since there is no case in which this function returns zero or negative values (each instruction takes at least one cycle to execute), we are done.

6.2 Completeness

To prove:

$$ONTO\_INIT(ABS\_ARM6)(INIT\_ARM)(INIT\_ARM6),$$

i.e.:

$$\begin{array}{l}\forall b \in state\_ARM\ \exists a \in state\_ARM6 : \\ \quad ABS\_ARM6(INIT\_ARM6(a)) = INIT\_ARM(b).\end{array}$$

Proof. For each state $b \in state\_ARM$ the state $a \in state\_ARM6$:

$$\begin{array}{rcl} a.mem & = & b.mem \\ a.dp.psr & = & b.psr \\ a.dp.reg.rx & = & \begin{cases} b.reg.rx + 8, & \text{if } rx = r15; \\ b.reg.rx, & \text{otherwise.} \end{cases} \end{array}$$

is a suitable witness.

6.3 Time-Consistency

6.3.1 Time-consistency of the state function for the specification

To prove:

$$TCON(STATE\_ARM),$$

i.e.,

$$\begin{array}{l}\forall b \in state\_ARM\ \forall t_1, t_2 \in \mathbb{N} : \\ \quad STATE\_ARM(t_1 + t_2)(b) = \\ \quad STATE\_ARM(t_2)(STATE\_ARM(t_1)(b))\end{array}$$

Time consistency of the $STATE\_ARM$ follows from Lemma 1 in [8]: *All non-initialized iterated maps are time-consistent.*

- $STATE\_ARM$ is an iterated map (by definition),
- $STATE\_ARM$ does not have initialization function.

Therefore, $STATE\_ARM$ is time consistent.

6.3.2 Time-consistency of the state function for the implementation

Proof of time consistency of the $STATE\_ARM6$ function is much more complicated.

According to the Theorem 2 (Section 3.2, page 7) we have to show that the initialization function is an invariant on states generated by the state function at steps 0 and 1.

Step 0:

$$\begin{array}{l}\forall a \in state\_ARM6 : \\ STATE\_ARM6(0)(STATE\_ARM6(IMM\_ARM6(a)(0))(a)) = \\ \qquad\qquad STATE\_ARM6(IMM\_ARM6(a)(0))(a)\end{array}$$

Step 1:

$$\begin{array}{l}\forall a \in state\_ARM6 : \\ STATE\_ARM6(0)(STATE\_ARM6(IMM\_ARM6(a)(1))(a)) = \\ \qquad\qquad STATE\_ARM6(IMM\_ARM6(a)(1))(a)\end{array}$$

Proof for step 0. The proof for time-consistency exists only in HOL and it is very large, so only the algorithm of the proof is shown below.

We consider only one component of the configuration: the address register *areg*. For other 25 components of the state-space reasoning is similar.

By expanding definitions we can rewrite the goal in the form:

$$INIT\_ARM6(INIT\_ARM6(a)) = INIT\_ARM6(a),$$

since $IMM\_ARM6(a)(0) = 0$ and $STATE\_ARM6(0)(a) = INIT\_ARM6(a)$ by definitions.

Let $INIT\_ARM6(a) = a'$. Then

$$a'.dp.areg = a'.dp.reg.r15,$$

by the definition of the $INIT\_ARM6$ (Section 5.2).

Let $INIT\_ARM6(a') = a''$. Then

$$\begin{array}{rcl} a''.dp.reg & = & a'.dp.reg \\ a''.dp.areg & = & a''.dp.reg.r15 = a'.dp.reg.r15 = a'.dp.areg \end{array}$$

For all other components we obtain equivalence in a similar way. Therefore,

$$a'' = a'.$$

Proof for step 1. By definition of the IMM_ARM6 function:

$$IMM\_ARM6(a)(1) = DUR\_ARM6(INIT\_ARM6(a)),$$

and we can rewrite the goal:

$$INIT\_ARM6(STATE\_ARM6(DUR\_ARM6(INIT\_ARM6(a)))(a)) = \\ STATE\_ARM6(DUR\_ARM6(INIT\_ARM6(a)))(a))$$

Then this goal is rearranged to be of the form:

$$\forall a \in state\_ARM6 : \\ a' = STATE\_ARM6(DUR\_ARM6(INIT\_ARM6(a)))(a) \Rightarrow \\ INIT\_ARM6(a') = a'$$

This rearranging prevents the state from being evaluated twice i.e. on both sides of the equality.

Then the main cases for the state function (case splitting on instruction type) are introduced manually, and the *simplifier* tactic is applied, which evaluates the state function $STATE\_ARM6$ required number of times (one to six, depending on the current instruction). This step generates a set of *leaf* goals.

Leaf goals have the form

$$INIT\_ARM6(a') = a',$$

where a' is the state of the implementation after executing one instruction. In each leaf goal the initialization function is applied, giving

$$a'' = INIT\_ARM6(a'),$$

and we have to show, that

$$a'' = a'.$$

Again, we consider only one component of the state, namely *areg* register. The contents of this register in state a' is some function of state the contents of this register in the initial state:

$$a'.dp.areg = f(INIT\_ARM6(a).dp.areg),$$

where f is some large term, which comes from the evaluation of state function. The term f consists of several sequential assignments, for example, in the first assignment *areg* is loaded with the effective address of memory access, then the *areg* register is loaded with the contents of register *r15*.

To this term the rewrite tactic is applied. The rewrite tactic is supplied with the set of simple lemmas like:

After sequential loading of the address register with some address *addr* ($areg = addr$) and then with the contents of register *r15* ($areg = r15$) the value of the register *areg* is equivalent to the value of the register *r15*.

The rewrite tactic performs case-splitting, if necessary, and produces the following result:

$$a'.dp.areg = a'.dp.reg.r15$$

If the state function $STATE\_ARM6$ is correctly defined, this equality should hold because fetch of instruction I_{i+2} is performed in the very same cycle, in which instruction I_i enters the execute stage, so the address register should contain the address of instruction I_{i+2} (cf. left part of Figure 4), otherwise the state function $STATE\_ARM6$ is wrong.

Using this result and definition of $INIT\_ARM6$ we obtain:

$$\begin{aligned} a''.dp.reg &= a'.dp.reg \\ a''.dp.areg &= a''.dp.reg.r15 = a'.dp.reg.r15 = a'.dp.areg \end{aligned}$$

For all other components rewrite tactic finds equivalences in the similar way.

Therefore,

$$a'' = a'.$$

Q.E.D.

6.4 Correctness

Now we have proven all intermediate properties, and can prove the correctness of the microprocessor implementation using Theorem 1.

We have to show that correctness condition (Definition 3 on page 6) holds on steps 0 and 1.

Step 0:

$$\begin{aligned} &\forall a \in state\_ARM6 : \\ &STATE\_ARM6(0)(ABS\_ARM6(a)) = \\ &ABS\_ARM6(STATE\_ARM6(IMM\_ARM6(a)(0))(a)) \end{aligned}$$

Step 1:

$$\begin{aligned} &\forall a \in state\_ARM6 : \\ &STATE\_ARM6(1)(ABS\_ARM6(a)) = \\ &ABS\_ARM6(STATE\_ARM6(IMM\_ARM6(a)(1))(a)) \end{aligned}$$

Proof for step 0. By expanding definitions, we obtain the following equality:

$$ABS\_ARM6(a) = ABS\_ARM6(INIT\_ARM6(a)).$$

This is true, because $INIT\_ARM6$ does not change three ISA components (*mem*, *reg* and *psr*).

Proof for step 1. This proof is very similar to the proof of step 1 for the time consistency of implementation state function. The only difference is the form of leaf goals after case splitting and simplification. Leaf goals are ISA state equivalences, with the primitive operations on the three ISA components: *mem*, *reg* and *psr*. These goals are also automatically discharged by the rewrite tactic.
Q.E.D.

7 Conclusion

The work [3] has established that the HOL proof tool can be used to support algebraic framework from [2] for modelling and verifying processors.

Using this framework the ARM6 micro-architecture has been modelled and formally verified in HOL [5]. The processor's organization 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.

The ARM6 specification and verification took about six month (one person), with the time being split almost equally between specification and verification.

References

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

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

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

[4] Anthony C. J. Fox. A HOL specification of the arm instruction set architecture. Technical Report 545, University of Cambridge, Computer Laboratory, June 2001.

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

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

[7] F. K. Hanna and N. Daeche. Specification and verification using higher-order logic: A case study. In Milne and Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 179–213. North Holland, 1986.

[8] Gennady Shmonin. Algebraic models of correctness for microprocessors. Seminar report, Universität des Saarlandes, 2004.