A Detailed Processor Model for Verification of Real-time Applications

Matthew Wilding
Computational Logic Inc.[1] and The University of Texas at Austin

Abstract. We describe a microprocessor model and its use for reasoning about real-time applications. The model is very detailed, and is expressed in the logic of a general-purpose theorem proving program that checks proofs. We verify mathematically that the bit vectors constituting an application cause a real-time system to have specified properties.

Key Words. microprocessor, real-time systems, reasoning, verification, mathematical models

1. Introduction

The correct operation of computer systems is difficult to assure because of their immense complexity. Building a reliable computer-based controller in a real-time environment is particularly difficult because events to which the computer must respond occur unpredictably. Many computer systems that operate in a real-time environment are safety-critical, so their correctness is crucial.

Formal proofs about computer programs are complex but not very deep, which makes them amenable to mechanical checking. Researchers have proved properties of programs using realistic computational models with the assistance of mechanical theorem provers. The use of mechanical tools to check correctness proofs enhances confidence that no errors occur in the proof, and therefore increases confidence in the implementation. The Nqthm theorem prover has been used to prove that several small compiled standard C library routines return correct values on the MC68020 processor (Boyer and Yu, 1992). The HOL theorem prover has been used to prove functional and timing properties of some small programs in a language that is closely related to a processor (Gordon, 1993). A machine code program has been proved using the Ergo theorem prover to implement a round-robin scheduler on a partial model of a pipelined processor (Fidge *et al.*, 1994).

There have been many formalizations of real-time systems with a view toward reasoning about aspects of their behavior. A common framework involves dividing the needed behavior into tasks and scheduling them so as to meet time deadlines. A wide variety of schedulers has been developed for various computer system designs to meet a variety of needs (Liu and Layland, 1973; Stankovic *et al.*, 1993; Wilding, 1993*b*). Another view of real-time systems models aspects of real-time system behavior with automata (Alur and Dill, 1990). The use of an abstract model of computation facilitates reasoning about system behavior.

We wish our specification of a real-time system to be abstract, and in particular not involve details about the computer application. But we also want our verified implementations to be close to the "reality" of an actual processor. Our approach is to use a formal language to express precisely real-time system specifications and use formal mathematical proof to demonstrate that an implementation achieves them. This paper addresses one challenge of this approach: How do we model computation in a way that allows useful and abstract reasoning about the computer application's effect on a real-time system? We model the behavior of a particular processor and relate it to a model of real-time system operation. We have chosen the FM9001 processor to model, and use the logic of the Nqthm theorem prover to describe it. Our conclusion is that a very detailed model of an actual processor is usable as a basis for reasoning about real-time systems with the aid of a mechanical theorem prover. The use of a model of an actual processor allows us to prove theorems about the behavior of a real-time system using the actual bit vectors that we download onto our processor.

The next section briefly describes Nqthm, the FM9001 processor, and some related application

[1] This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.

verification work. We introduce a model that describes the operation of the FM9001, and characterize formally that the real-time system acts in a manner consistent with the FM9001 model. We illustrate our model with a very simple program that reads an input device and displays the value on an output device, and describe a mechanically-checked proof that a real-time system containing an FM9001 running this program, a "bouncy" button, and an LED functions properly as a light switch.

2. Background

Nqthm is the name of both a logic and an associated theorem proving system that is sometimes called the Boyer-Moore theorem prover (Boyer and Moore, 1988). A large number of mathematical theorems from many disparate domains have been proved using Nqthm. The Nqthm logic is a quantifier-free, first-order logic resembling Pure Lisp. The user inputs definitions and conjectures to the Nqthm theorem prover which, when successful, outputs proofs. Proved conjectures are applied in later proofs as inference rules. In a shallow sense the theorem prover is fully automatic since once a conjecture is input to the theorem prover the proof is uninfluenced by the user. Usually, however, non-trivial theorems require the proof and subsequent use of many subsidiary lemmas, and so a carefully designed sequence of conjectures is needed to lead Nqthm to a proof.

The most important property of Nqthm-generated proofs is soundness. Extensive use of Nqthm over many years suggests that Nqthm-generated proofs are extremely dependable. The use of Nqthm often uncovers mistakes in hand-constructed proofs.

One interesting domain to which Nqthm has been successfully applied is computer systems verification. The requirement imposed by Nqthm to get every detail of a proof exactly correct is very important in computer systems verification as any mistake can be catastrophic. One reason Nqthm is well-suited for doing modeling of and proof about computer systems is the executable nature of the Lisp-like Nqthm logic. After one introduces a definition into the Nqthm system, one can "run" it as one might execute any other program. When building models of computer systems upon which to base theorems about computer system correctness, it is very useful to be able to execute the definitions and thereby use the formal model as a system simulator.

The FM9001 is a general-purpose 32-bit microprocessor with 15 arithmetic and logical operations, five addressing modes, extensive arithmetical flag support, 16 general-purpose registers, and other

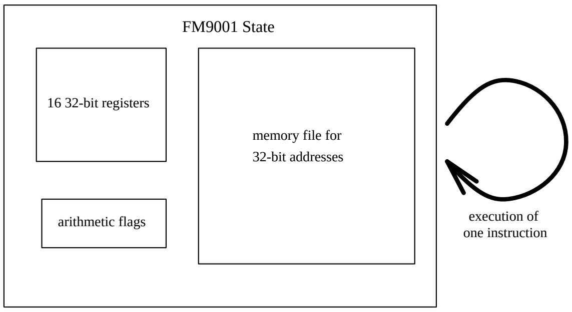

Fig. 1. Programmer Model of the FM9001

features that make it practical to use (Brock *et al.*, 1994). The FM9001 is verified in the sense that a programmer-level model of the processor operation is proved implemented in a hardware description language. The verification was accomplished using Nqthm, and the specifications are expressed in the Nqthm logic. The FM9001 programmer-level model describes the operation of the FM9001 at the level programmers are used to thinking about processors. The state of the processor is composed of the values of the general-purpose registers, the arithmetical flags, and the memory. It is a formal model of how a programmer might think of a processor, but it also has the virtue of being tied through machine-checked proof to a hardware design. Figure 1 presents pictorially the programmer-level model of the FM9001. The execution of an instruction is accomplished each "tick" of a computation, and the model gives a value for each bit of the processor's memory, registers, and flags for each instruction.

The FM9001 programmer-level model is useful as more than a specification for the correct hardware operation. It can be used as a basis for reasoning about applications. For instance, it is used as the target machine of a compiler for the language Piton (Moore, 1989). The compiler is proved correct using Nqthm, and since its target is the FM9001 programmer model the proofs can be "stacked" in the sense that we need only assume that the hardware is working properly for a Piton program compiled onto the FM9001 to work according the formal semantics of Piton. A 300-line Piton program that calculates moves efficiently in the puzzle-game Nim has been specified and proved correct (Wilding, 1993*a*). The FM9001 hardware description has been used to fabricate the processor and a single-board computer containing the processor has been constructed that allows FM9001 program execution (Wilding, 1992).

3. A Real-time Model of the FM9001

We present in this section a model of the FM9001 formalized in Nqthm that we use to reason about real-time programs running on an FM9001 processor. We discuss some of the issues faced in the development of the model, and to convey a notion of what the real-time model is. The complete real-time model consists of many Nqthm definitions, some of which are very complicated. We do not present the model formally in this paper, but refer the interested reader to its precise and complete description (Wilding and Young, 1995).

The programmer model developed as a part of the FM9001 hardware verification is inadequate for our purpose for two reasons. The FM9001 executes different instructions at different speeds depending on the instruction type, addressing modes, processor flag settings, and memory speed, so reasoning about timing behavior requires a model more detailed than the 1-instruction, 1-tick model. Also, to reason about interesting real-time programs we must model I/O.

Although the real-time model we describe in this section is quite different from previous models of the FM9001, we retain as much of the old model as possible. The proof that the FM9001 programmer model is implemented correctly by the FM9001 hardware design lends it considerable authority. The Nqthm function `FM9001-step` implements the single step of the programmer's model represented in Figure 1 and was developed for the FM9001 hardware verification (Brock *et al.*, 1994). We use this function in the real-time model as our definition of the functional behavior of the FM9001. Note that although we reuse much of the FM9001 programmer's model in our new real-time model, since the real-time FM9001 model is not identical to what has been proved implemented correctly in hardware we do not retain the extremely high confidence in model accuracy.

3.1. *Timing*

The real-time FM9001 model must reflect how much time an instruction consumes during its execution. We express time in terms of the hardware clock that is being used to drive the execution of the FM9001. We call the time between two successive ticks of this underlying clock a *microcycle*. This view of the FM9001 is implicit in its hardware design, since each instruction is defined in terms of a sequence of internal FM9001 states, one for each microcycle (Brock *et al.*, 1994).

The instruction execution time calculation is in a sense outside the scope of the FM9001 microprocessor since the response time of the memory affects the execution time of instructions. The board we have built to run FM9001 programs provides constant-time access to all memory locations. We assume in our real-time FM9001 model

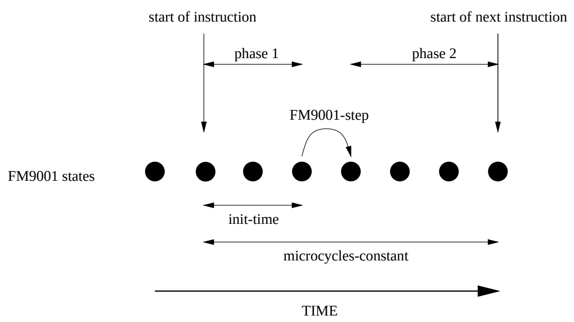

Fig. 2. The Real-time Instruction Execution Model

that memory access time is constant, and use it to parameterize a function that calculates instruction timing.

3.2. *I/O*

We perform input and output when using the FM9001 with memory-mapped I/O devices. That is, we associate I/O devices with locations in the FM9001 memory, and use FM9001 instructions that access memory to accomplish I/O. No distinction is made in the model between the value of memory locations that are associated with output devices and the value of memory locations that are "normal" memory. For example, a bell might be associated with a memory location 23 so that a 0 value turns it off and a non-0 value turns it on. We add inputs into our model in a very general manner, allowing any memory location to be changed at any microcycle during execution.

3.3. *The Two-phase model*

Since we wish our model of I/O to be as general as possible, we allow inputs and outputs to change at any microcycle. The challenge is to incorporate this into an FM9001 model simply but realistically. Our approach is to consider the execution of each instruction as having two phases. During each microcycle of the first phase the values of the FM9001 state remain unchanged except for the effect of inputs. The instruction's effect is "seen" at the first microcycle of the second phase. The values of the final FM9001 state of the first phase are used to calculate the first FM9001 state of the second phase. The FM9001 state remains unchanged for each microcycle of the second phase except for the effect of inputs. The two-phase model of instruction execution is represented in Figure 2.

Obviously, the internal state of the actual FM9001 processor need be consistent with our model after the execution of each instruction. When modeling the interaction of the processor with memory, particularly I/O devices, we require precision that reflects *to the microcycle* the effect of our processor on memory and the effect of inputs on the processor. We view each instruction as composed of two phases in order to use the inputs from the appropriate microcycle and to update the outputs at exactly the right microcycle during execution of an instruction. We take advantage of the relative simplicity of the FM9001 hardware design to calculate exactly the microcycle that memory is sensed or updated during instruction execution.

3.4. *Two-Phase Instruction Model Inaccuracies*

The simple two-phase instruction model is potentially inaccurate in two ways.

Like all conventional processors, the FM9001 fetches instructions from memory. Our simple two-phase model may be inaccurate if the value of the instruction being executed is changed by action of input devices. There are a number of ways we could complicate our real-time FM9001 model in order to reflect this possibility, but since it appears to have very little practical significance we accept this inaccuracy in the interest of greater model simplicity.

Some FM9001 instructions are *read-write* instructions that in addition to an instruction fetch both read from and write to memory. Any such FM9001 instruction can be written as several functionally equivalent (albeit slightly slower) instructions that are not read-write, so in the interest of model simplicity we have chosen to avoid them. We define an Nqthm function that takes an FM9001 state and returns whether the current instruction is read-write, and is used by the model to determine when to signal an error.

3.5. *Tying the model together*

We calculate the value of an FM9001 state for a particular microcycle during processor execution from the initial state, a list of input lists that represent the memory-mapped input device values for each microcycle leading up to the state in which we are interested, and a memory access time. If a read-write instruction is encountered an error-signaling 0 is returned. If the number of microcycles of input is not sufficient to complete more than the first phase of the current instruction, we return a new state that reflects only the inputs. If the number of microcycles of input is sufficient to complete the first phase of the current instruction but not sufficient to start execution of the next instruction, we return a new state that reflects the execution of the current instruction and the inputs. If there is enough input to execute more than just the current instruction, we recursively

```
program                   corresponding assembly code

#x00E0043F                (move t f r1 (pc+))
#x50000000                #x50000000
#x00E00011                (move t f r0 (r1))
#x01E04400                (inc t f (r1) r0)
#x00E03C3F                (move t f pc (pc+))
#x40000000                #x40000000
```

Fig. 3. Example FM9001 Program

```
*(fm9001-rt-history
   (fm9001-ex-state) 3
   (repeat 50 nil) '((#x50000000 out)))
'(((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5))
  ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5))
  ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5))
  ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5))
  ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5))
  ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5))
  ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5))
  ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5))
  ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5)) ((OUT 5))
  ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6)) ((OUT 6)))
```

Fig. 4. Example FM9001 real-time model execution

execute the model using a state that reflects the execution of the current instruction and the reduced number of microcycles of input remaining.

The Nqthm function `fm9001-rt-history` generates values from successive FM9001 states in this way, and has a fourth argument that describes the values of the FM9001 state in which we are interested. It is very complex, just as the FM9001 is complex, so we demonstrate it on an example in this section and refer the interested reader to its precise definition (Wilding and Young, 1995). The Nqthm constant `fm9001-ex` represents an FM9001 state that has the bit vectors in Figure 3 loaded at location #x40000000, a bit vector representing the value 5 at location #x50000000, and a program counter with value #x40000000. The program increments repeatedly the value at location #x50000000. We take advantage of the executability of Nqthm functions in order to execute this example program symbolically. Figure 4 presents a run starting in this example state, assuming a memory response time of 3, for 50 microcycles during which no input devices change memory. The value of the fourth argument of `fm9001-rt-history` specifies that the model return the natural number representation of the value of location #x50000000 at each microcycle and that it be associated with the name `out`. Note that shortly after the program begins execution the value changes.

4. FM9001-based System Correctness Lemmas

In the previous section we described an Nqthm model FM9001 processor model. But we are not ultimately interested in proving theorems only about the execution of the FM9001. We wish

program	corresponding assembly code
#x00E0043F	(move t f r1 (pc+))
#xC0000000	#xC0000000
#x00E00011	(move t f r0 (r1))
#x00E0043F	(move t f r1 (pc+))
#xC0000001	#xC0000001
#x00E04400	(move t f (r1) r0)
#x00E03C3F	(move t f pc (pc+))
#x40000000	#x40000000

Fig. 5. Light switch program

to prove theorems about real-time systems that contain an operating FM9001. In this section we explain how we use the FM9001 model to state and prove correctness lemmas about the behavior of a real-time system.

4.1. *Histories and the FM9001*

We represent the state of a real-time system at an instant in time as an association of values with variables. We represent a system over time by a *history*, a sequence of associations in which consecutive entries represent instants separated by some constant amount of time. If for example the relevant aspects of a system we were modeling were the state of a buzzer and the value of particular location in memory, we might choose to represent the system with the following history.

```
(((buzz on) (loc 3)) ((buzz off) (loc 3))
 ((buzz off) (loc 4)))
```

The Nqthm function `extract-from-history` takes as arguments a list of new name/old name pairs and a history, and calculates a new history containing the values of the old names in the old history associated with the new names. The Nqthm function `extension-historyp` takes as arguments two histories, and returns whether each value of each association of the first history has identical value in the corresponding association of the second history. We use our Nqthm definitions to describe when a history representing a real-time system is consistent with our FM9001 model. In particular, some variables in a history may represent values that are input to or output from an operating FM9001. We express formally that each processor output in a history association has the value that the model calculates for it.

The "mirror" program is a simple application with which we demonstrate our formalism. It reads from an input device and copies the value to an output device. `(mirror-initial-state)` contains starting at memory location #x40000000 the bit vectors in Figure 5. A memory-mapped input device is associated with memory location #xC0000000 and an output device is associated with location #xC0000001.

If `h` is a history with variables `signal` and `output` that represent the input and output values of an FM9001 running the mirror program with the I/O devices connected as described, then the following Nqthm predicate holds on `h`:

```
(extension-historyp
  (fm9001-rt-history (light-switch-initial-state) 3
   (extract-from-history '((#xc0000000 signal)) h)
   '((#xc0000001 output)))
 h)
```

That is, history `h` has a value associated with the variable `output` that is identical to memory location #xC0000001 as calculated by the real-time FM9001 model, assuming the FM9001 starts in the state we have constructed, with constant memory access time of 3 microcycles, and with a single input device at location #xC0000000 that has the value of variable `signal` in `h`.

4.2. *A Simple Real-time System Specification*

We have physically constructed in our laboratory a very simple microprocessor-controlled real-time application: a light-switch. The system has two I/O devices — a button and a light. A button's signal is not guaranteed to reflect the status of the button until some known amount of time after a depress or release due to the literal bounce of the button contacts (Ursoleo, 1994). The theorem we prove is informally presented below.

```
for all histories:
  if
   history reflects FM9001 running light-switch
   and
   history has signal that reflects button actions
  then
   light quickly mirrors button in history
```

The first hypothesis of this theorem is formalized in the previous section. Our approach for formalizing other constraints on histories is described in (Wilding and Young, 1995), and involves the use of a kind of timed automaton to specify satisfying histories. We describe the behavior of a "bouncy" button, and the property that the light quickly reflects the most recent action on the button, using this formalism.

The Nqthm-checked proof of the Nqthm formalization of this correctness theorem is rather complicated. As has also been observed in other projects (Fidge *et al.*, 1994; Wilding, 1993*a*), program correctness proofs using formal mathematics is time-consuming. Note that each program instruction in this controller has associated with it dozens of invariants that seem obvious but require proof. As an example, the sixth line of the program in Figure 5 appears to move the contents of the value of the button signal stored in register 0 into memory location #c0000001 whose address is contained in register 1. But, of course, in order to prove that it works this way we must prove many things: the instruction has not been overwritten by the execution of the program, the value #c0000001 is in register 1 when we execute this instruction, the value of the button signal has not been changed in register 0 since it was read, etc. Although the proof is complex, we have great confidence that our correctness lemma is a theorem because the proof is certified by Nqthm.

The file that Nqthm checks is over 100K and contains mostly lemmas for Nqthm to prove and apply in later proofs. This does not include the definition of the real-time FM9001 model, some standard libraries of Nqthm strategy rules (Wilding, 1991), or some general rules about the real-time model. The proof of this small application required about 6 man-weeks of effort. The proof of several lemmas about our real-time model and other lemmas needed by the proof but not involving the light switch example required another 6 man-weeks of effort. We expect future proofs will be far easier because of greater experience and our reuse of previously-proved lemmas.

The verified light switch example runs on the FM9001 in our laboratory.

5. Conclusions

With mechanical theorem proving support, we have been able to prove using formal mathematics the correctness of a very simple real-time system using the exact bit vectors we download onto our controller. We reason about the operation of the program using the semantics implicit in a very detailed model of the processor. This approach has several advantages.

- It is very concrete as we are verifying the actual bit vectors of the program.
- The properties we are proving are appropriately abstract in the sense that they are formalizations of the properties we need from our system, not just properties of the program's execution.
- The use of machine-checked proof gives us very high confidence that we have made no reasoning errors.

We are currently developing our proof capabilities in order to verify more complex examples.

6. REFERENCES

Alur, Rajeev and David Dill (1990). Automata for modeling real-time systems. In: *Automata, Languages and Programming – 17th International Colloquium* (M.S. Paterson, Ed.). Vol. 443 of *Lecture Notes in Computer Science*. Springer-Verlag. pp. 322–335.

Boyer, R. S. and J S. Moore (1988). *A Computational Logic Handbook*. Academic Press.

Boyer, Robert S. and Yuan Yu (1992). Automated correctness proofs of machine code programs for a commercial microprocessor. In: *Automated Deduction – CADE-11* (D. Kapur, Ed.). Vol. 607 of *Lecture Notes in Computer Science*. Springer-Verlag. pp. 416–430.

Brock, B., W. A. Hunt Jr. and M.J. Kaufmann (1994). The FM9001 microprocessor proof. CLI Technical Report 86.

Fidge, Colin, Peter Kearney and Mark Utting (1994). Formal specification and interactive proof of a simple real-time scheduler. Technical report. Software Verification Research Centre, The University of Queensland.

Gordon, Mike (1993). A mechanized Hoare logic of state transitions. Technical report. University of Cambridge Computer Laboratory.

Liu, C.L. and James W. Layland (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery* **20**(1), 46–61.

Moore, J Strother (1989). A mechanically verified language implementation. *Journal of Automated Reasoning* **5**(4), 493–518. Also published as CLI Technical Report 30.

Stankovic, John A., Marco Spuri and Marco Di Natale (1993). Implications of classical scheduling results for real-time systems. Technical report. University of Massachusetts.

Ursoleo, John (1994). Software performs debouncing for large array of switches. *Electronic Design* p. 40. October 3.

Wilding, Matthew (1991). Proving Matijasevich's lemma with a default arithmetic strategy. *Journal of Automated Reasoning* **7**(3), 439–446.

Wilding, Matthew (1992). Using the FM9001 single-board computer. CLI Note 260.

Wilding, Matthew (1993*a*). A mechanically verified application for a mechanically verified environment. In: *Computer-aided Verification – CAV '93* (Costas Courcoubetis, Ed.). Vol. 697 of *Lecture Notes in Computer Science*. Springer-Verlag.

Wilding, Matthew (1993*b*). An optimal real-time scheduler based on a simple model of computation. CLI Note 276.

Wilding, Matthew and William Young (1995). Specification and verification of real-time executives. CLI Technical Report 110.