

**SOFTWARE VERIFICATION RESEARCH CENTRE  
DEPARTMENT OF COMPUTER SCIENCE  
THE UNIVERSITY OF QUEENSLAND**

**Queensland 4072  
Australia**

**TECHNICAL REPORT**

**No. 92-10**

**Real Time Behaviour of a RISC processor: Specification and Computer-Aided Verification.**

**Peter Kearney, Mark Utting and Keith Whitwell**

**April 1994**

**Phone: +61 7 365 1003**

**Fax: +61 7 365 1533**

# Real Time Behaviour of a RISC processor: Specification and Computer-Aided Verification

Peter Kearney, Mark Utting and Keith Whitwell  
Software Verification Research Centre  
The University of Queensland  
St Lucia, Queensland 4072, Australia

**Abstract.** This paper gives an overview of: two levels of formal specification of the real-time behaviour of a commercial RISC chip; an approach to verifying the higher-level specification relative to the lower-level one; and the proof tool and environment used for the proofs. The specifications are written in functional logic, which provides an adaptable modal facility. The proof tool and environment support both rewriting and forward and backwards proof, through a development of the sequent calculus called window inference, and provide for the flexible interaction of manual and automatic modes of proof.

## 1. Introduction

Interest in RISC processors for critical real-time applications is growing rapidly [12]. This paper describes an approach to the formal specification and computer-aided verification of the behavioural and real-time aspects of a MIPS R3000 RISC CPU [4]. The approach provides a basis for verifying the real-time behaviour of assembly-level software running on the processor, including interrupt-driven software.

In common with other RISC processors, instruction execution in MIPS processors is pipelined; that is, instruction execution is divided into a number of phases, and at a given time a number of instructions are active simultaneously, in different phases. On the other hand, the assembly language level of the architecture hides the detail of this pipeline structure, and presents a more conventional sequential instruction processing interface to the CPU.

To increase assurance that the specification correctly describes the real-time behaviour of the actual device, our primary specification defines the behaviour of the instruction pipeline directly rather than taking an instruction level view. We also postulate an *instruction-level*

specification as an abstraction of the pipeline specification. This instruction-level specification provides a basis for software verification.

The instruction-level specification needs to be verified against the pipeline-level specification; that is, the postulated instruction level properties should be derived from the pipeline specification. This process also increases confidence in the pipeline specification, since the CPU pipeline is designed to support a consistent assembly-level language view. Indeed, the process of developing an instruction-level specification and its verification has revealed defects in the original pipeline specification.

We are currently carrying out this verification using an interactive proof tool [8] which supports window inference [7], a technique permitting the user to focus on sub-terms and transform those sub-terms using context-dependent hypotheses. The construction of proofs is assisted by the use of an environment [11] which runs on top of the interactive prover, providing: a number of useful proof tactics; a way of configuring table-driven heuristic tactics; and facilities for recording proofs as structured tactics.

## 2. The pipeline level specification

### 2.1. Specification Scope

This specification, documented in [10], makes a number of assumptions about the CPU and its configuration. These assumptions can be realised in a suitable configuration of, for example, the IDT79R3500A CPU [3]. The principal assumptions are that the translation lookaside buffer is disabled, that the CPU is not stalled by other processors or DMA controllers, and that the instruction and data caches are used as the main memory of the system.

### 2.2. Specification formalism

The formalism on which the specification is based is essentially as in [6]. Reasoning is about complete runs of the system, and behaviours of the system are specified by constraining possible system runs.

To achieve more compact and higher-level notation, the specification uses the implicit parameterisation facility of functional logic [9], which supports classical reasoning about assertions that are evaluated relative to an implicit parameter. In effect, functional logic provides an adaptable modal reasoning facility. An assertion, or other term, that depends on an implicit parameter is interpreted as a function with domain a set of implicit parameter values. The logic includes forward function composition, denoted ‘;’. The expression  $\mathbf{x};\mathbf{y}$  is evaluated with respect to an implicit parameter  $i$  as follows:  $(\mathbf{x};\mathbf{y})(i) = \mathbf{y}(\mathbf{x}(i))$ . Intuitively,

Function	Description
<code>is_time(V)</code>	<code>V</code> is a valid time (i.e., <code>V in times</code> ).
<code>time</code>	The current time.
<code>at(T)</code>	A function that changes the current time to be <code>T</code> .
<code>next_time(P)</code>	The first time strictly later than the current time at which the predicate <code>P</code> is true, or <code>?</code> if there is no such time.
<code>prev_time(P)</code>	The most recent time strictly earlier than the current time at which the predicate <code>P</code> is true, or <code>?</code> if there is no such time.
<code>next(P)</code>	Equals <code>at(next_time(P))</code>
<code>prev(P)</code>	Equals <code>at(prev_time(P))</code>
<code>during(T1,T2,P)</code>	<code>true</code> if <code>T1</code> and <code>T2</code> are times and the predicate <code>P</code> is true at all times from <code>T1</code> to <code>T2</code> , inclusive; <code>false</code> otherwise.
<code>until(T,P)</code>	Equals <code>during(time,T,P)</code>

Table 1: Specification Functions.

`y` is here evaluated with respect to an implicit parameter delivered by `x`.

The set of possible implicit parameters includes a value `?`, which represents undefinedness.

The implicit parameters used to model system runs are time-trace pairs, that is, members of  $Times \times Traces$  where  $Times$  is the set of non-negative integers and  $Traces$  is the set of possible system behaviours (over all times). That is,

$$\begin{aligned}
 Traces &= Times \rightarrow States \\
 States &= Locs \rightarrow Values
 \end{aligned}$$

where  $Locs$  is a set of locations and  $Values$  includes integers, truth values and locations.

The time component of a time-trace pair represents the current time.

The unit of time used for the pipeline level specification is the clock cycle time of the CPU (with each new time unit starting on the falling edge of the  $\overline{SysOut}$  pin).

Table 1 lists some of the notations used in the specification, with an informal description of their semantics. Formal definitions are in [5].

### 2.3. Specifying instruction execution

In the space available only an indication of the specification method can be given. Details are in [10].

Instruction execution is described by specifying the behaviour of the CPU pipeline, which consists of the following five stages:

- IF — Instruction fetch.
- RD — Read arguments from registers and decode instruction.
- ALU — Perform the required arithmetic operation or calculate the effective address for load and store instructions.
- MEM — Access memory (data cache) if required (for load and store instructions).
- WB — Write back ALU or memory results to registers.

The pipeline is modelled directly in the specification, including program counters and instruction registers for each pipeline stage. Each pipeline stage also has a number of flags associated with it, for example to indicate if the instruction in that phase is to be annulled (in interrupt processing), or is in a branch delay slot or is in a load delay slot.

A number of conditions can cause the pipeline to stall. A *run* cycle is a CPU cycle in which no stalls occur, so that the pipeline progresses. A location *run* is used to record the run status of the current cycle. That is,  $\text{run}^\wedge$  is true when no stalls are occurring. (The postfix ‘ $\wedge$ ’ is the location dereferencing operator).

Pipeline bypassing [2] is modelled explicitly. The bypassing logic is encapsulated in the defined functions `alu_get`, `alu_put` and `mem_put`. Asserting the predicate `alu_put(Reg, Val)` means that at the next run cycle, the bypass locations correctly reflect an ALU phase update to the register `Reg` with the value `Val`. The predicate `mem_put` plays a similar role for register updates at the MEM phase. The function `alu_get(Reg)` yields the value in the appropriate bypass location for the register `Reg`, if `Reg` is affected by bypassing, or the value in the register itself, if not.

Many of the axioms in the specification refer to a predicate called `hw`. Intuitively, `hw` is true of all implicit parameters that are time-trace pairs and whose trace component represents a feasible run of the modelled hardware.

The `running` predicate is stronger than `hw` since it also requires that the current cycle is a run cycle, that there is a future run cycle and that the `reset` flag is false until the next run cycle.

$$\text{running} = \text{hw} \text{ and } \text{run}^\wedge \text{ and } \text{until}(\text{next}(\text{run}^\wedge); \text{time} - 1, \text{not } \text{reset}^\wedge).$$

A typical axiom of the specification is the following for an unsigned add instruction. It says that if `running` is true, and the instruction currently in the ALU phase is an unsigned add instruction and that instruction is not annulled, then bypass locations are updated appropriately.

```

running
and not alu_annul^
and r_type(alu_ireg^, Rs, Rt, Rd, 0, fn_addu)
=> alu_put(Rd, (alu_get(Rs) + alu_get(Rt)) mod 2**32).

```

### 3. The instruction level

The instruction level defines an abstraction of the pipeline level. At this level, the internal state of the CPU contains the programmer visible registers, plus the program counter (`pc`), the program counter of the next instruction to be executed (`next_pc`) and the instruction registers associated with those two program counters. There are also several flags that reflect whether the current instruction is annulled (`annul`), in a branch delay slot (`in_bds`) or in a load delay slot (`in_lds`). The effects of each instruction are modelled as occurring during the MEM stage.

This abstraction is given by defining the instruction level concepts in terms of the pipeline level specification. For example,

```

pc = mem_pc.
ireg = mem_ireg.
next_pc = alu_pc.
annul = mem_annul.

```

Access to registers is based on the function `areg` (for accessing registers) and the predicate `areg_put` (for asserting the updating of registers). These effectively abstract from the bypassing logic and give an instruction level view of register access.

Many instruction level properties use the predicates `irun` and `irunning` defined as follows.

```

irun = hw and run^ and
      during(prev(run^);prev(run^);prev(run^);time, time, not reset^).

irunning = irun and until(next(run^);time - 1, not reset^).

```

An instruction level property for the unsigned add is as follows.

```

irunning
and not annul^
and not int_pending
and r_type(ireg^, fn_addu, Rs, Rt, Rd, 0)
and (in_lds^ => Rd /= load_reg^)
=> areg_put(Rd, (areg(Rs) + areg(Rt)) mod 2**32).

```

## 4. Interactively verifying the instruction level

This section describes the proof tools we are using and illustrates aspects of those tools in an example proof.

### 4.1. Proof tools

The proof theorem prover Demo3.2 [8] is the latest in a series of prototype provers which support a window inference technique [7]. This technique, a development of the sequent calculus, supports both forward and backward proof, as well as the ability to rewrite subterms using context dependent hypotheses.

The tactic language of the prover is the logic programming language Qu-Prolog [1]. The tactical environment [11] uses tactics to automate some low-level proof details, to raise the level of recorded proofs by recording them as structured tactics, and to provide heuristic assistance to the interactive user. Heuristic assistance is provided by table-driven heuristic tactics which can be used during interactive proof, either to suggest useful proof steps or to carry out a portion of proof automatically. Heuristically constructed proofs are recorded as tactics which are independent of the heuristics that construct them, thus allowing heuristics to evolve while already constructed proofs are recorded in a compact and stable form.

### 4.2. An Example Proof

To illustrate the prover and the tactical environment we discuss the proof of a lemma that is needed for several proofs about the instruction level specification. The proof begins by entering the name and statement of the lemma, as follows (user input follows the % prompt and is terminated by '.').

```
/fol/fun/set/arith/div/rt/prec/mips/ispec
% prove(prevrun_running , irun => prev(run^));running).
Window level: 1           Window number: 1
focus:  irun => prev(run^);running
Equivalence: <=>      Goal: true
Heuristics: arith_auto(X_0)  Auto Status: Off, Low interaction
Rule Set: Standard
```

The resulting display shows the outermost window (level 1) of the proof attempt. The current focus is the theorem to be proved and our goal is to transform the focus to true, using equivalence rules involving  $\Leftrightarrow$ . The bottom two lines of the display are explained later and are omitted until then.

The first step of the proof is to focus (or, open a window) on the conclusion of the lemma. This is done by specifying the path down the syntax tree to the desired focus. (An

X-windows version of the prover allows selection by clicking the mouse on the desired focus.) Opening a window on the second argument of an implication causes the antecedents to be added to the list of hypotheses.

```
% [2].
Window level: 2           Window number: 2
focus: prev(run^);running
      1: irun
Equivalence: <=>   Goal: true
```

The next proof step is to expand the definition of `irun`, thus changing hypothesis 1 into the following three hypotheses.

```
4: during(prev(run^);(prev(run^);(prev(run^);time)), time, not reset^)
3: run^
2: hw
```

This is achieved by typing: `use(def(irun)) on_focus (h irun)`. The `on_focus` command opens a window at the position given by its second argument (the hypothesis `irun` in this case), executes its first argument, then closes that subwindow. The `use(R)` command uses rule `R` as a rewrite rule (in either direction) to transform the current focus. Note that `def(irun)` is the name of the definition for `irun` given on Page 5.

The next obvious step in the proof is to focus on `running`, expand its definition and prove each of the resulting conjunctions. Moving across the `prev(run^)` function has the effect of changing the current time, and thus the implicit parameter, so the opening rule for `';` will retain only hypotheses that are obviously valid at the previous run cycle. Therefore, before focusing on `running`, we transform each hypothesis `H` to an equivalent one of the form `prev(run^);H1`.

We use the `during_prev` rule to transform hypothesis 4 by issuing the command:

```
% use(during_prev) on_focus (h during(.,.,.)).
```

The `during_prev` rule is

```
hw and A
=> (during(prev(A);T, T2, P)
    <=> prev(A);during(T, next(A);T2, P)).
```

The `use(during_prev)` command instantiates this rule so that one side of the `<=>` unifies with the focus and automatically discharges both the other side of the `<=>` and the two assumptions of the rule from the available hypotheses.

To transform `hw` to `prev(run^);hw`, we first do a forward proof step that deduces the definedness of `prev(run^)` from hypothesis 4, using the rule:

```
prev_defined: prev(A);P => prev(A) /= ?.
```

The `fwd(Rule,Target)` command uses `Rule` to perform a single forward inference step that produces `Target` as an additional hypothesis.

```
% fwd(prev_defined, not prev(run^)=?).
Window level: 2          Window number: 2
focus: prev(run^);running
      7: not prev(run^)=?
      5: prev(run^);during(prev(run^);(prev(run^);time),
                          next(run^);time, not reset^)
      3: run^
      2: hw
Equivalence: <=>      Goal: true
```

Two forward inference steps are sufficient to prove `prev(run^);hw` and `prev(run^);run^`. We can also invoke `fwd` without specifying a rule name. This causes it to search the entire database of rules for a rule that can reach the given target. This can take several minutes. However, the rule name is recorded in the proof record so that the proof can be re-run without searching and without ambiguity due to changes in the rule database.

```
% fwd(prev(run^);hw), fwd(prev(run^);run^).
Window level: 2          Window number: 2
focus: prev(run^);running
      11: prev(run^);run^
      9: prev(run^);hw
      (Hypotheses 2...7 are unchanged)
Equivalence: <=>      Goal: true
```

Now we focus on `running`. Note how hypotheses of the form `prev(run^);H` are transformed to `H` and other hypotheses are not carried into this window.

```
% [2].
Window level: 3          Window number: 5
focus: running
      11: run^
      9: hw
      5: during(prev(run^);(prev(run^);time), next(run^);time, not reset^)
Equivalence: <=>      Goal: unknown
```

The command `use(def(running))` expands the focus to:

```
focus: hw and (run^ and until(next(run^);time - 1, not reset^))
```

The first two conjunctions are easily discharged by two `use(h _) on_focus [1]` commands. After this, we expand the definition of `until` as follows:

```

% use(def(until)).
Window level: 3           Window number: 11
focus:  during(time, next(run^);time - 1, not reset^
        11: run^
        9: hw
        5: during(prev(run^);(prev(run^);time), next(run^);time, not reset^
Equivalence: <=>      Goal: unknown

```

Comparing the focus with hypothesis 5, we see that the set of times in the focus expression is a subset of the times in hypothesis 5. This suggests that the focus should follow from hypothesis 5, via the rule:

```

during_subset: is_time(X_9) and is_time(X_10)
               and X_11 =< X_9 and X_10 =< X_12
               and during(X_11, X_12, X_13)
               => during(X_9, X_10, X_13).

```

The command `back(during_subset)` performs a backward proof step. The focus matches with the conclusion of the `during_subset` rule and is replaced by the antecedents of that rule. To maintain the equivalence between the resulting window and the current one, the complement of the original focus is also added to the hypotheses. Since the `X_11` and `X_12` meta-variables do not occur in the conclusion of the rule, they appear uninstantiated in the new focus.

```

% back(during_subset, _).
Window level: 4           Window number: 14
focus:  is_time(time) and (is_time(next(run^);time - 1)
        and (X_11 =< time and (next(run^);time - 1 =< X_12
        and during(X_11, X_12, not reset^)))
        12: not during(time, next(run^);time - 1, not reset^
        11: run^
        9: hw
        5: during(prev(run^);(prev(run^);time), next(run^);time, not reset^
Equivalence: <=>      Goal: unknown

```

A variant of the `use(Rule)` command, `use_goal(Rule,Goal)`, can be used to prove the first of these conjunctions. The `Goal` argument is used to constrain the search to rules which transform the `Focus` to the `Goal`, possibly utilising hypotheses. The second conjunction follows from a lemma called `nextrun_sub1_is_time` once we have established that `next(run^)` is defined.

```

% use_goal(_, true) on_focus [1].
use_goal(time_is_time, true)
...
% fwd(is_time(next(run^);time)),
% fwd( not next(run^) = ? ),
% use(nextrun_sub1_is_time) on_focus [1].
Window level: 4          Window number: 14
focus: X_11 =< time and (next(run^);time - 1 =< X_12
        and during(X_11, X_12, not reset^))
        16: not next(run^) = ?
        14: is_time(next(run^);time)
        (Hypotheses 5...12 are unchanged)
Equivalence: <=>      Goal: unknown

```

Another variant of the `use` command, called `use_unify`, can be used to prove the last conjunction of the focus by unifying it with the `during(...)` hypothesis. The `use_unify` command permits instantiation of meta-variables in the proof: here `X_11` and `X_12` are instantiated.

```

% use_unify(h during(.,.,_)) on_focus [2,2].
Window level: 4          Window number: 14
focus: prev(run^);(prev(run^);time) =< time
        and next(run^);time - 1 =< next(run^);time
        (Hypotheses are unchanged)
Equivalence: <=>      Goal: unknown

```

Focussing on the second of these conjuncts (with the command [2]), we see that it is a simple arithmetic tautology, since times are represented as natural numbers. After a single forward proof step to make this explicit

```

% fwd(is_nat(next(run^);time)).

```

we can turn the proof over to an automatic set of heuristics that attempt to transform arithmetic terms to a canonical form. Executing the following two commands

```

% heuristics(reduce).
% auto_sg.

```

causes the bottom two lines of the window display to change to:

```

Heuristics: reduce(X_27)  Auto Status: On, Low interaction
Rule Set: Standard

```

The `auto_sg` command changes the tool into automatic mode (`Auto Status` equals `On`) for a *single goal* (the current focus), after which it will revert to manual mode. There are similar commands for attempting the remainder of the entire proof in automatic mode (`auto`) and for going into *high interaction* automatic mode (`iauto` and `iauto_sg`). In this mode,

the proof tool is driven by the current table of heuristics, but the user is required to accept or reject each proposed heuristic step. This is a simple way of controlling the heuristic proof process.

As shown above, the `heuristics(Table)` command is used to select the current table of heuristics. Heuristic tables are typically written by users to capture common proof fragments associated with particular theories. In addition to the `reduce` heuristics, we have a heuristic table for proving propositional tautologies and other tables that can assist with proofs about arithmetic and the verification of sequential code.

The `reduce` heuristics attack the current focus by moving all terms to the left hand side of the inequality, using the `le_sub_inv` rule. However, this rule has preconditions that require all the terms to be integers, so the `useR_goal` command is used to perform a recursive depth-bounded search for proofs of the required preconditions. The recursive search is limited to using rules from the current rule set (`Standard`). Such rule sets are designed to avoid obvious loops in the search and to order the rules from most specific to most general. The following steps are taken by the heuristics:

```

useR_goal(le_sub_inv, (next(run^);time - 1) - next(run^);time =< 0, X_28)
trying rule le_sub_inv
satR(is_int(next(run^);time))
  -- can be shown by trans_true(time_to_int, [14])
satR(is_int(next(run^);time - 1))
  -- can be shown by trans_true(nextrun_sub1_is_int, [9, 16])
Window level: 5           Window number: 23
focus: (next(run^);time - 1) - next(run^);time =< 0
        23: is_int(next(run^);time)
        21: is_int(next(run^);time - 1)
        19: is_nat(next(run^);time)
        17: prev(run^);(prev(run^);time) =< time
           (Hypotheses 5...16 are unchanged)
Equivalence: <=>      Goal: unknown
Heuristics: reduce(X_29)  Auto Status: On, Low interaction
Rule Set: Standard

```

The next steps taken by the `reduce` heuristics are to focus on the left hand side of the inequality and simplify it to `-1`.

```

focus([1], X_30)
Reduce focus to -1
  use_goal(def(-), X_32) ,
  focus([1], (use_goal(def(-), X_33),
              focus([2], use(eval_eq(-(1), -1))))),
  use_goal(add_assoc, X_34),
  focus([2], use_goal(add_comm, X_35)),
  use_goal(add_assoc, X_36),
  focus([1], useR_goal(add_inv_w, 0, X_37)),
  use(eval_eq(0 + (-1), -1))
Window level: 6           Window number: 32
focus: -1
...

```

After this, the `reduce` heuristics stop. However, after closing window level 6 the focus becomes `-1 =< 0` and this is proved by a single `use_goal` command.

```

% use_goal(_, true).
use_goal(lt_imp_le, true)
Window level: 5           Window number: 23
focus: true
      23: ...
Equivalence: <=>      Goal: unknown
% quit.

```

The remaining conjunct from the level 4 window can be proved in half a dozen steps, which we do not show here, and closing several windows completes the proof.

## 5. Conclusions

This paper has given an overview of: two levels of formal specification of the real-time behaviour of a commercial RISC chip; our approach to verifying the higher-level specification relative to the lower-level one; and the proof tool and environment we are using for the proofs.

We believe that supporting a range of proof modes, such as single step manual interaction, automatic heuristic proof, heuristic proof guided by the user, and blind recursive search, is an important key to raising the productivity of formal verification with tool support.

The tactical environment illustrated in this paper extends an interactive proof tool to support such modes. It also supports both forward and backward proof steps in a natural fashion and allows the user to perform some proof steps without having to specify a rule name. We are continuing to develop the tactical environment, particularly in the area of support for reasoning about the MIPS R3000 specifications.

## Acknowledgement

This work has been supported by the Information Technology Division of the Australian Defence Science and Technology Organisation. Particular thanks to Dr. Brian Billard.

## References

- [1] A.S.K Cheng, Robinson P.J., and J. Staples. Higher level meta programming in Qu-prolog 3.0. In *Proceedings of the International Conference on Logic Programming-8*, pages 285–298. MIT Press, 1991.
- [2] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, San Mateo, California, 1990.
- [3] IDT. *1991 RISC Databook*. Integrated Device Technology, 3236 Scott Boulevard, Santa Clara, California 95054, 1991.
- [4] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.
- [5] P. Kearney, J. Staples, and A. Abbas. Functional verification of hard real-time programs. In J. van Leeuwen, editor, *Algorithms, Software, Architecture, Information Processing 92 Vol.1*, pages 113–119. Elsevier Science Publishers B.V. (North Holland), 1992.
- [6] P. Kearney, J. Staples, A. Abbas, and Chuichang Liu. Functional verification of real-time procedural code: a simplified RS232 software repeater problem. Technical Report 91-2, Software Verification Research Centre, Department of Computer Science, University of Queensland, August 1991. Revised May 1992.
- [7] Peter J. Robinson and John Staples. Formalising the hierarchical structure of practical mathematical reasoning. Technical Report 138, Key Centre for Software Technology, Department of Computer Science, University of Queensland, December 1989. Revised August 1990. To appear in *Logic and Computation*.
- [8] Peter J. Robinson and Tong Gao Tang. The demonstration interactive theorem prover demo2.1. Technical Report 91-4, Software Verification Research Centre, Department of Computer Science, University of Queensland, September 1991.
- [9] J. Staples, P. Robinson, and D. Hazel. A functional logic for higher level reasoning about computation. Technical Report 141, Key Centre for Software Technology, Department of Computer Science, University of Queensland, 1989. Revised September 1991; to appear in *Formal Aspects of Computing*.
- [10] Mark Utting and Peter Kearney. Pipeline specification of a MIPS R3000 CPU. Technical Report 92-6, Software Verification Research Centre, Department of Computer Science, University of Queensland, October 1992.
- [11] Keith Whitwell. A tactical environment for an interactive theorem prover. Technical Report 92-7, Software Verification Research Centre, Department of Computer Science, University of Queensland, December 1992.
- [12] Tom Williams. Performance pushes RISC chips into real-time roles. *Computer Design*, pages 79–86, September 1991 1991.