

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

**Queensland 4072
Australia**

TECHNICAL REPORT

No. 94-13

A Layered Real-Time Specification of a RISC Processor.

Peter Kearney and Mark Utting.

June 1994

Phone: +61 7 365 1003

Fax: +61 7 365 1533

A Layered Real-Time Specification of a RISC Processor

Peter Kearney and Mark Utting
Software Verification Research Centre,
University of Queensland, Brisbane, Australia.

Abstract

This paper gives an overview of the real-time specification of a commercial RISC processor. The specification is at two related levels, with an abstraction relation defined between them. The lower level specification models separate stages of execution of up to five overlapped instructions. The higher level specification abstracts from the lower level to recapture an atomic, instruction level view of code execution. The load word instruction is used as an example to illustrate the specification at both levels.

1 Introduction

This paper summarises a two-layered approach to specifying the real-time behaviour of a commercial RISC processor (the MIPS R3000¹). This provides a sound formal basis for reasoning about real-time programs running on the processor, with a degree of precision and rigour not made available by existing processor documentation.

The real-time performance of a RISC processor such as the MIPS R3000 depends on its use of instruction pipelining and data and instruction caches. In the ideal case, a MIPS R3000 executes one instruction per clock cycle. However, because of certain dependencies between instructions, and because data or instructions may not be in cache, this performance is not always achieved. Such conditions cause stalls in the pipeline, thus delaying the execution of instructions. Reasoning about real-time code running on the processor must take account of these and other conditions such as the effects of interrupts on pipelined execution.

We have specified the real-time processor behaviour in two layers. The first layer is a *pipeline-level specification* in which the progress of instructions through

¹MIPS is a registered trademark of MIPS Computer Systems.

the instruction pipeline is modelled explicitly: at each CPU cycle the state of up to five overlapped instructions in the pipeline is described. Specifying the processor at this detailed level increases confidence that processor behaviour is correctly modelled.

The second layer, *the instruction-level specification*, is defined as an abstraction of the pipeline level. The aim of this second layer is to recover as much as possible of the conventional machine model presented to the assembly-level programmer, in which instructions execute atomically and sequentially. This level is a more natural programming model and is more convenient for reasoning about code. It was only by first developing the pipeline-level model and subsequently abstracting to the instruction level that we obtained an instruction level specification which we could be confident faithfully represented processor behaviour.

Previous reports [5, 6] define the formal pipeline-level and instruction-level models in full detail. This paper gives an overview of and motivation for these specifications, concentrating on the way in which the instruction level is obtained as an abstraction of the pipeline level. Section 2 gives required background material on the real-time model used and the system specified. Section 3 introduces the specification formalism and highlights some aspects of the pipeline-level specification, by analysing the effects of the *load word* instruction at each pipeline stage. Section 4 explains how the instruction-level model is obtained as an abstraction of the pipeline level, and illustrates aspects of the instruction-level specification. For comparison with the pipeline level, an instruction-level specification of the load word instruction is discussed. In the space available not all aspects of the two models can be discussed. In particular, most details concerning interrupt processing are not discussed in this paper.

2 Background

2.1 A trace model with discrete time

We use a discrete time scale, where the unit of time is the CPU cycle time. The choice of this time scale is motivated by our intended use of the specification for reasoning about real-time software. The execution time of an instruction will be a multiple of the CPU cycle time. The details of event sequences at the sub-cycle level will be irrelevant to software behaviour and we wish to avoid formal modelling at too fine a level of granularity.

Our model of real-time behaviour is based on traces, where each trace represents a possible system behaviour (over all times). The state of the processor at each time is represented by a mapping from a set of locations to values:

$$\textit{Traces} = \textit{Times} \rightarrow \textit{States}$$

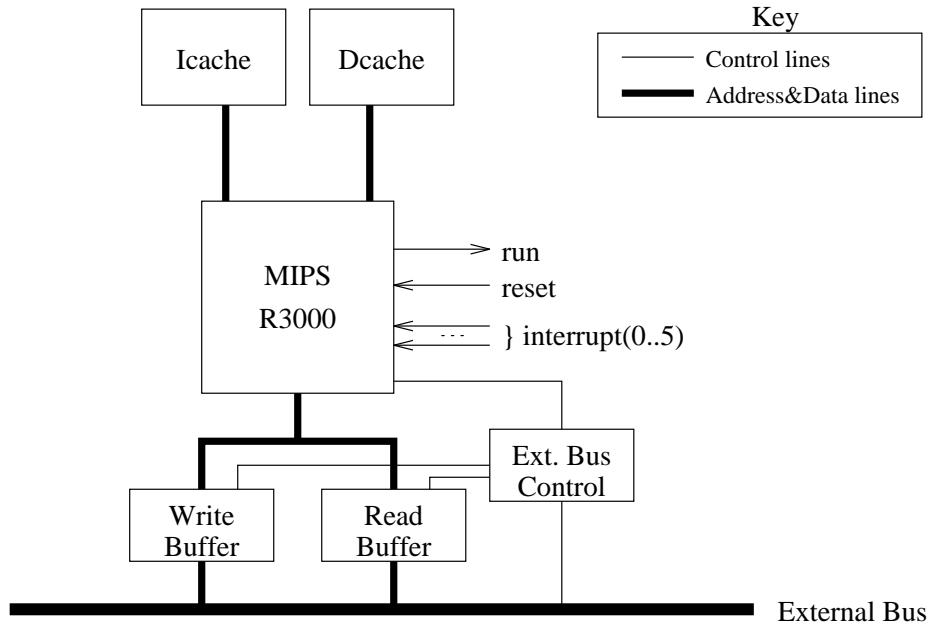


Figure 1: Block Diagram of the Specified System.

$$States = Locs \rightarrow Values$$

where $Times$ is the set of non-negative integers, $Locs$ is a set of locations and $Values$ includes integers, truth values and locations. Locations are used to model memory locations, processor registers, flags and components of internal processor state. The real-time behaviour of the CPU is specified by defining constraints on possible traces.

2.2 The System to be Specified

Figure 1 is a block diagram of the system to be specified. It shows the three main interfaces to the R3000 CPU: the instruction and data caches and the external bus. The CPU can access both caches within each clock cycle, but accessing the external bus is typically much slower, so it is decoupled from the CPU by read and write buffers.

In addition to the three main bus interfaces of the CPU, Figure 1 shows several other CPU control flags that are used in the specification. The run flag is output by the CPU to indicate whether or not the pipeline is advancing. Cycles in which run is not true are called *stall cycles* (the $\hat{\ }$ notation is explained in 3.1). The $reset$ and $interrupt(i)$ flags are controlled by external events and read by the CPU.

All MIPS R3000 CPUs have a five stage pipeline structure, composed of the following stages:

- IF — Initiate instruction fetch.
- RD — Complete instruction fetch, 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.

Figure 2 shows how the execution of instructions is overlapped in this five stage pipeline. Note the following aspects of pipelined execution.

- Instruction fetching is initiated half way through the IF cycle and is completed during the first half of the RD cycle.
- When an interrupt occurs, all instructions currently in the pipeline are *annulled* (they have no effect), save for the instruction currently in WB, whose WB stage is completed.
- To facilitate clean interrupt behaviour, registers are not updated until the write back stage. Since instructions in the pipeline which are annulled before write back leave processor registers unchanged, interrupted instructions have no effect on the register state.
- To avoid the delayed write-back causing a delay to later instructions which access register values, values computed in the ALU and MEM stages for register updating are *bypassed* to the next two instructions. For example, in figure 2, the results of the ALU stage of instruction i are made available through bypassing to the ALU and MEM stages of instruction $i + 1$ and the ALU stage of instruction $i + 2$. Further, results of the MEM stage of instruction i are bypassed to the the ALU stage of instruction $i + 2$.

In the ideal case, the pipeline results in five times as many instructions being executed per second as would be the case in an equivalent non-pipelined computer. In practice the pipeline may be *stalled* for one or more cycles, during which no instruction progresses through the pipeline. Three important types of stalls are:

Instruction cache miss stalls. If an instruction is not in instruction cache, it cannot be fetched in one cycle and the pipeline is stalled until the instruction is fetched from external memory. Our specification models this by the insertion of a number of stall cycles following the RD stage.

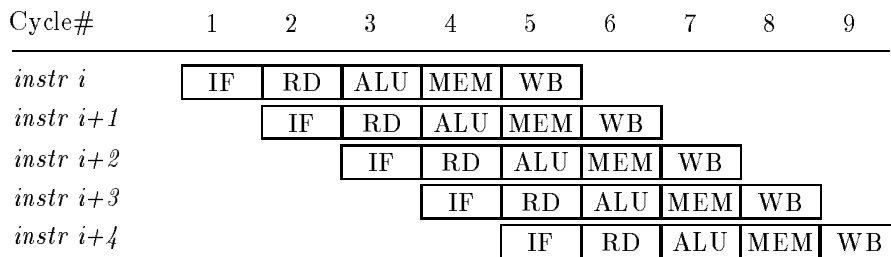


Figure 2: MIPS R3000 Instruction Pipeline

Data cache miss stalls. If a load or store instruction references a location whose value is not in cache the pipeline is stalled until the value is fetched from external memory.

Multiplier/Divider Busy stalls. Because multiplication and division take multiple cycles, these operations are performed in separate hardware in parallel with other instructions, with the results being stored in special HI and LO registers. However, if an instruction tries to access the HI or LO register while a multiply or divide operation is in progress, the pipeline is stalled until the operation is completed.

The MIPS documentation specifies that certain code sequences are illegal and their results undefined [2]. We briefly discuss two illegal code types.

Code in a load delay slot. An instruction is said to be in a *load delay slot* if it immediately follows a load instruction (which loads a memory value into a register). It is illegal for code in a load delay slot to access the target register of the preceding load because the MEM cycle cannot complete in time to make the resulting value available in the next instruction's ALU stage. For example, in Figure 2, the MEM cycle of instruction i only completes, making the result of a load available, after the ALU cycle of instruction $i + 1$ is executed.

Code in a branch delay slot. An instruction is said to be in a *branch delay slot* if it immediately follows a branch instruction. Suppose instruction i of figure 2 is a branch instruction. Since address calculation in a branch is completed during the ALU stage, the address for the target of the branch is not available in time to initiate fetching from the target for instruction $i + 1$. To avoid stalling the pipeline for a cycle at every branch instruction, the instruction in a branch delay slot is executed *regardless of the branch decision*. The branch decision of instruction i is completed during the ALU stage in time to initiate fetching the target instruction as instruction $i + 2$ (the fetch is initiated half way through the IF stage of $i + 2$).

The MIPS stores only a single program counter when an interrupt occurs. When an interrupt occurs that would normally require storing the address of an instruction in a branch delay slot, the address of the immediately preceding branch instruction is actually stored and it is thus re-executed after interrupt processing. If a branch instruction could be placed in a branch delay slot, no value of a stored program counter could assure a proper restart of instructions. Thus branch instructions in a branch delay slot are illegal.

2.3 The load word example

We illustrate both levels of specification by seeing how they define the behaviour of the load word instruction (opcode `lw`). This instruction fetches a word from a designated address and loads it into a target register. We highlight specification details relevant to the case when the instruction is in instruction cache and the loaded word is in data cache (and thus no stalls are generated). Sections 3.3 to 3.7 illustrate aspects of the pipeline-level description of `lw` behaviour and in section 4 the behaviour of the same instruction under the same circumstances is defined at the instruction level.

3 The pipeline specification

3.1 Specification formalism

The formal language of both levels of specification is functional set theory [1] which is based on functional logic [4]. Functional logic incorporates an implicit parameter mechanism into classical logic. This implicit parameterisation is used so that assertions can be relativised to some environment of ‘current’ information, without continual explicit reference to that environment. Other logical formalisms with similar objectives include modal and temporal logics and programming logics such as Hoare logic and dynamic logic.

The semantic basis of functional logic considers a set over which an implicit parameter ranges. This set is called the *index* set, and its members *indexes*.

In a model of a functional logic theory, all objects are functions which have the index set as their domain and co-domain. All types including the boolean type are treated uniformly, thus merging the concepts of formula and term into a single concept called an expression. Functional logic also supports the modelling of partial functions. A distinguished index, say $?$, is used as a function’s value whenever the function’s input is not in the domain of the modelled partial function. We call indexes other than $?$ *normal*. Functions f on the index set which satisfy the strictness condition $f(?) = ?$ are called *actions*.

Function composition is an operation on actions, denoted by the infix operator `;`, such that for all actions f and g and all normal indexes i :

$$(\mathbf{f};\mathbf{g})(i) = \mathbf{g}(\mathbf{f}(i)).$$

Note here that the action \mathbf{g} is effectively evaluated at the index yielded by the action \mathbf{f} at the current index. Intuitively, the `;` operator shifts the implicit parameter. We call an action which gives the same result at all normal indexes an *absolute* action. Note that a syntactic constant could denote either an absolute or a non-absolute action.

In functional set theory the index set is a model of classical set theory (and thus any set is available as a potential implicit parameter). The pipeline level specification is given as an axiomatic theory which is an extension of functional logic and set theory. The consistency of this extension could be checked relative to the consistency of functional set theory (and thus relative to classical set theory) by defining the primitives and proving the axioms in functional set theory. That is beyond the scope of the work described here.

The key implicit parameters used in the specification are members of $Times \times Traces$. Intuitively the time component of such a time-trace pair represents the ‘current’ time. Thus when the implicit parameter is such a pair, both the entire trace and the current time are implicitly available. The current time determines a ‘current state’, namely the state in the trace at the current time.

Table 1 introduces some notation used in this paper. Definitions of many of these notations appear in previous work [3]. To illustrate the notation, consider the expression `next(run^)` which appears often in the specification. Here `run` is a location, and thus `run^` is the contents of `run` at the current time. Now `next(run^)` is an action which shifts the current time to the first time in the future trace at which the contents of the location `run` is true. Intuitively, this is the next run cycle. Note that `next(run^);time` is the *time* at which the next run cycle occurs.

To improve readability, we sometimes use a local definition facility of the form

```
letabs Var Expr within Body.
```

Informally, this has the effect of evaluating `Expr` and declaring `Var` to be a variable that is local to `Body`, with a value that is absolute and is equal to the value of `Expr`.

All formal notation which appears in typewriter font in this paper can be input directly to the Ergo interactive theorem prover [7].

3.2 Specification components

To specify the behaviour of each pipeline stage, it is necessary to define some registers for holding intermediate results between the pipeline stages. For exam-

Function	Description
$\hat{}$	A postfix operator such that if L is a location in the current state, $L^{\hat{}}$ is the contents of that location in the current state.
<code>is_abs(V)</code>	V is an absolute action.
<code>locs</code>	The set of locations.
<code>time</code>	The time component of the implicit parameter, i.e. the current time.
<code>at(T)</code>	An action that changes the current time of the implicit parameter to be T .
<code>next_time(P)</code>	The first time strictly later than the current time at which the predicate P is true, or $?$ 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 P is true, or $?$ 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>nth(P, N)</code>	The identity action if N is zero; if N is positive, shifts the current time to the n -th time in the future at which P is true; if N is negative, shifts the current time to the n -th time in the past at which P is true.
<code>during(T1, T2, P)</code>	true if $T1$ and $T2$ are times and the predicate P is true at all times from $T1$ upto (but not including) $T2$; false otherwise.
<code>until(T, P)</code>	Equals <code>during(time, T, P)</code>

Table 1: Predefined Specification Functions and Predicates.

ple, the program counter of each instruction is carried along the pipeline as the instruction is executed, so most pipeline stages have a program counter register (`*_pc`, where `*` is the name of the pipe stage). The following declarations introduce these as absolute constants. Thus each of these constants denotes the same location at all times.

```
absolute_constant rd_pc      :locs.
absolute_constant alu_pc     :locs.
absolute_constant mem_pc     :locs.
absolute_constant wb_pc      :locs.
```

Similarly, after the instruction has been read in the RD stage, it is carried along the pipeline in an instruction register (`*_ireg`) as it is executed. The WB

stage does not require an instruction register, since all instructions have the same effect in the WB stage, so only the locations `alu_ireg` and `mem_ireg` are required.

Most of the pipeline stages also have `*_in_bds` and `*_in_lds` flags that reflect whether or not the instruction at that stage is in a branch delay slot or a load delay slot.

The CPU control logic of the specification maintains an `*_annul` flag for each stage, which determines whether the instruction should be annulled (due to an interrupt) or executed as normal. An interrupt at time t causes all stages of all instructions currently in the pipeline to be annulled, except that a WB stage at time t is completed.

The following locations correspond to the general purpose registers of the CPU, which are updated by the WB stage. The `gen_reg` function maps general purpose register numbers into their locations. Note that register zero is not defined as a location because of its special nature: it ignores writes and always returns 0 on reads.

```
absolute_function gen_reg(Regnum:1 upto 32):locs.
```

We separate as much as possible the functional and the real-time behaviour of the processor. Much of the functional behaviour is specified by defining the state of the pipeline at the next run cycle, using the action `next(run^)`, which changes the implicit time parameter to the time of the next run cycle. Other parts of the specification determine *when* the next run cycle will be, by asserting the presence or absence of various stall cycles.

Many of our axioms refer to a predicate called `hw` which is true of all implicit parameters that are time-trace pairs and whose trace component represents a feasible run of the modelled hardware. The pipeline specification is written so that it predicts behaviour only when `reset` has not been asserted. If `reset` is asserted the specification leaves subsequent behaviour of the processor undefined. The following `running` predicate is therefore frequently used as a precondition. It requires the current cycle to be a run cycle and the `reset` flag to be false until the next run cycle.

```
constant running
=== hw
   and run^
   and until(next(run^);time, not reset^).
```

3.3 The IF stage

The IF stage of each instruction is the same: appropriate locations are initialised to initiate a read from the instruction cache, from the location determined by

the instruction before last. For example, referring to figure 2 on page 5, the IF stage of instruction $i + 2$ sets up a read from instruction cache from the location determined by the ALU phase of instruction i .

In practice, the ALU stage does this in the first part of the cycle and the IF stage initiates the instruction cache read in the remainder of the cycle. To avoid specifying at a sub-cycle level, the ALU stage is specified so that it determines the required location by the *end* of the ALU stage, and in turn the IF stage sets up a read from the location at the end of the IF stage.

For example, again referring to figure 2 (page 5), the ALU stage of instruction i determines the value of rd_pc^{\wedge} by the end of cycle 3 (assuming no stalls). This value is used by the IF stage of instruction $i + 2$ to initiate an instruction cache read at the start of cycle 3.

The formal axiom for this behaviour is as follows, where `i_cache_read` (not defined here) has the effect of setting up an instruction cache read. Note that in this axiom `next(run^);rd_pc^` effectively denotes the value of rd_pc^{\wedge} at the end of the IF stage.

```
axiom if_stage
=== running
  and not if_annul^
  and is_word_addr(next(run^);rd_pc^)
=> next(run^);
  (rd_error^ = no_error
   and if(rd_annul^,
          icode_inactive,
          icode_read(physical_addr(rd_pc^),
                    cacheable(rd_pc^)))
  ).
```

Recall that in our running example, the instruction being fetched is assumed to be a ‘lw’ instruction.

3.4 The RD stage

As with the IF stage, the RD stage is the same for all instructions. The main three activities that occur during the RD stage are:

- completing the instruction cache read,
- decoding the resulting instruction and

- reading the source registers from the register file (this occurs during the second half of the RD stage, after the WB stage has updated the register file).

The first of these activities is specified by the axioms for the instruction cache (not detailed here), with the results being placed into locations `i_data`, `i_tag` and `i_valid`.

The specification does not explicitly describe the decoding of instructions, rather using a decoded logical representation for the instruction code. Similarly, the specification refers to dereferences of source register values only in the ALU stage, which is where their values are used, rather than in the second half of the RD stage, which is the earliest possible time operationally. This has the same logical effect, but avoids having to model the pipeline using half-cycle time steps.

The following axioms define how the program counter and the result of a successful instruction cache read are passed along the pipeline for execution. Similar axioms for other pipeline stages pass the program counter and the instruction register from one stage to the next. Note that `icache_readhit` (whose definition is not given here) is true if the address read is currently in instruction cache.

```
axiom next_alu_pc
=== running
=> next(run^);alu_pc^ = rd_pc^ .
```

```
axiom next_alu_ireg
=== running
and not rd_annul^
and icache_readhit
=> next(run^);alu_ireg^ = i_data^ .
```

In our running example, we are assuming that the ‘lw’ instruction is in cache.

To determine the time at which the next run cycle occurs, it is necessary to know whether and for how long stalls occur in the pipeline. The following axiom asserts that if the fetch results in an instruction cache hit (or if the RD cycle is annulled), there are no instruction miss stalls (`i_miss_stall` is true at the current time if there is an instruction miss stall, and false otherwise). Axioms for other pipeline stages also determine the presence or absence of stalls, and collectively this information determines *when* the next run cycle will be.

```
axiom i_miss_nostall
=== running
and (rd_annul^
or icache_readhit)
=> until( next(run^);time,
not i_miss_stall)
```

3.5 The ALU stage

The ALU stage is the main stage where the axioms are instruction-dependent. At the pipeline level, we explicitly model bypassing, as discussed in section 2.2. We will see in section 4 how the instruction-level succeeds in hiding this bypassing.

Both the ALU and MEM stages use pairs of *bypass* registers, that capture the destination register number and its new value so that updating of the main register file can be postponed until the WB stage. The ALU stage uses the `mem_dest_reg` to record the destination register of any result computed in the ALU phase and `mem_dest_val` to record the value. Similarly the MEM stage uses the pair `wb_dest_reg` and `wb_dest_val` to record the destination and value of results computed in the MEM stage.

For the `lw` instruction, the ALU phase sets the destination register to the register which is the target of the load, and also asserts that the next instruction is in a load delay slot. The following function is used for asserting this.

```
function alu_delayed_put(Reg)
=== (next(run^);mem_dest_reg^ = Reg
     and next(run^);alu_in_lds^).
```

Access to registers is via the `alu_get` function. This accesses the correct bypass register, if required, or accesses the general registers directly. Note that access to the target of a load in a load delay slot is undefined (`alu_get` yields `?`). Thus an instruction following a load which tries to access the target register will obtain the undefined value.

```
function alu_get(Reg)
=== if( alu_in_lds^ and Reg=mem_dest_reg^,      ?,
     if( Reg=0,                                0,
     if( Reg=mem_dest_reg^ and not mem_annul^,  mem_dest_val^,
     if( Reg=wb_dest_reg^ and not wb_annul^,   wb_dest_val^,
     /* otherwise */                           gen_reg(Reg)^
     )))).
```

The ALU stage determines whether or not a branch is taken. The `lw` does not cause a branch, and this will be asserted using the following function.

```
constant no_branch
=== (next(run^);rd_pc^ = twos_comp(rd_pc^ + 4)
     and not next(run^);alu_in_bds^).
```

A load instruction calculates an effective address, translates that address into a physical address and sets up a read to occur during the following MEM stage. The following predicate defines how this is done (`dcache_read` defines the effect of setting up a data cache read).

```

function setup_read(Addr)
=== (letabs A = Addr within
    next(run^);
      (if(mem_annul^,
          dcache_inactive,
          dcache_read(physical_addr(A), cacheable(A)))
      )
    ).

```

The `lw`, in common with other load instructions, does not cause a branch, as asserted by the following:

```

axiom load_instr_alu
=== running
   and not alu_annul^
   and group(alu_ireg^) = load_instr
   => no_branch

```

Finally, the ALU stage of the `lw` instruction asserts `setup_read` which defines address translation and initiation of the data cache read. The read is completed in the MEM stage.

```

axiom lw_instr_alu
=== running
   and not alu_annul^
   and alu_ireg^ = lw_instr(Rt,Offset,Base)
   and Addr = twos_comp(alu_get(Base) + int_val_n(Offset,16))
   and is_word_addr(Addr)
   and (next(run^);user_mode => Addr < 2**31)
   => alu_delayed_put(Rt)
      and setup_read(Addr)
      and next(run^);mem_error^ = no_error.

```

The processor mode condition is required here because if the processor is not in user mode in the MEM stage (`next(run^)` relative to the current time) access to addresses above $2^{31} - 1$ is illegal.

3.6 The MEM stage

The `lw` instruction loads a value into its target register. The following predicate defines the appropriate bypass values to reflect that, and is used in the specification of `lw`.

```

function mem_put(Val)
=== (next(run^);wb_dest_reg^ = mem_dest_reg^
    and next(run^);wb_dest_val^ = Val).

```

As in the RD phase, causes of stalls must be accounted for. The following axiom asserts that there are no stalls due to data cache misses, under the stated preconditions (`d_miss_stall` is true if and only if there is a stall due to a data cache miss). The preconditions for this are that either the MEM phase is annulled, or the current instruction is not a load instruction, or the word read is in cache.

```

axiom d_miss_nostall
=== running
    and (mem_annul^
        or group(mem_ireg^) \= load_instr
        or dcache_readhit)
=> until(next(run^);time, not d_miss_stall).

```

Finally, the following instruction defines the behaviour of the `lw` when a data cache hit occurs.

```

axiom lw_instr_mem_hit
=== running
    and not mem_annul^
    and mem_ireg^ = lw_instr(Rt,Offset,Base)
    and dcache_readhit
=> mem_put(d_data^).

```

3.7 The WB stage

The WB stage of all instructions is the same—any result value of the instruction is written back into the general purpose register file. If the instruction entering the WB stage has not updated any registers, its ALU and MEM stages will have set its destination to be register zero, so the writeback will have no effect. Note that if the annul flag of the instruction is true, the writeback is cancelled, leaving all register values unchanged. Again, we will see in the next section how the pipelining details are hidden at the instruction level.

The following axiom defines values of the general registers at the next run cycle, as defined by the state of the bypass registers.

```

axiom next_reg
=== running
  and not wb_annul^
  => (all x
      is_abs(x)
      and x : 1 upto 32
      => next(run^);gen_reg(x)^
      = if(x=wb_dest_reg^,
          wb_dest_val^,
          gen_reg(x)^
      ).

```

The next axiom specifies that the general registers remain unchanged if the WB stage is annulled.

```

axiom next_reg_annul
=== running
  and wb_annul^
  => (all x
      is_abs(x)
      and x : 1 upto 32
      => next(run^);gen_reg(x)^ = gen_reg(x)^
  ).

```

4 The instruction-level specification

4.1 Relationship to the pipeline-level

Our approach to recovering an instruction-level view of CPU behaviour is to identify the ‘current instruction’ as the instruction currently in its MEM stage. The MEM stage is chosen for this role because it is during this stage that updates to external memory occur. Also, interrupts are processed at the beginning of the MEM stage: when an interrupt occurs, the instructions currently in the MEM, ALU and RD stages are annulled (the IF stage has not yet begun when interrupts are processed). The current WB stage is always completed. Thus, conceptually, the ‘current instruction’ is interrupted and the ‘previous’ instruction is completed. Figure 3 illustrates how the progress of the ‘current instruction’ is related to the pipeline level (compare with figure 2).

The instruction-level model presents a reduced number of processor registers. It also hides all details of pipeline bypassing, instead presenting a set of abstract registers which are updated atomically. Figure 4 illustrates components of the instruction-level model.

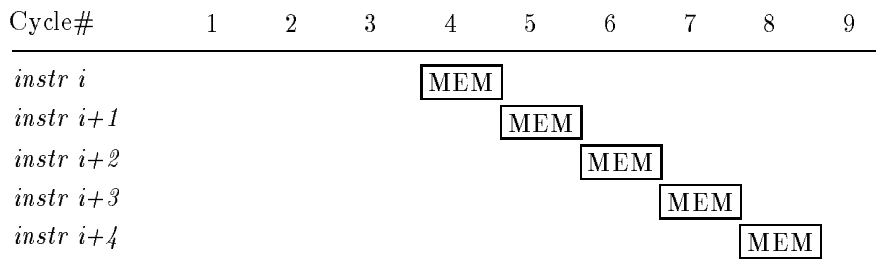


Figure 3: MIPS R3000 Instruction-Level View

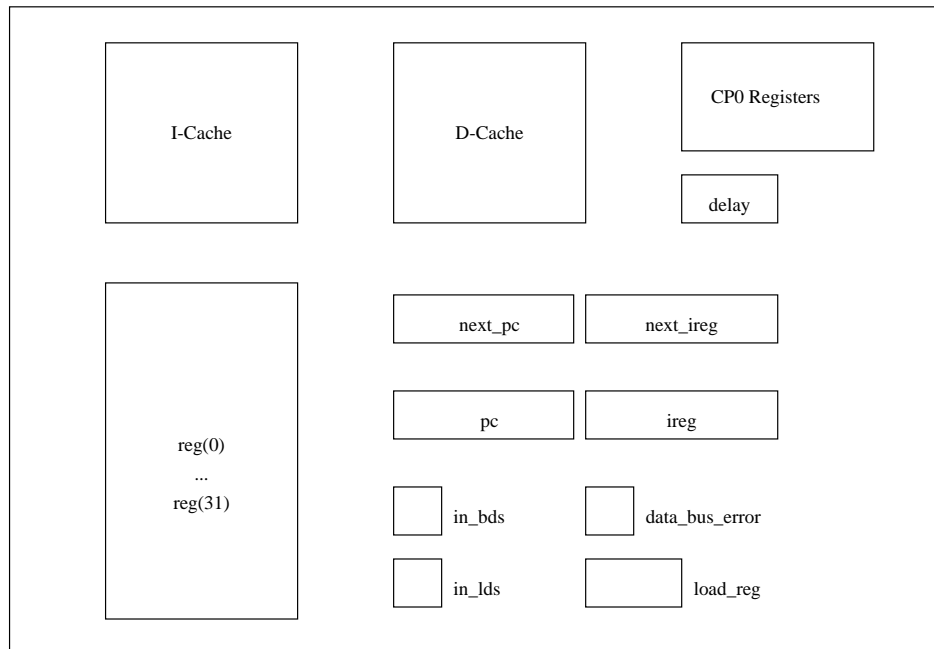


Figure 4: Instruction-Level Model of R3000 Processor

Conceptual components of the instruction-level model are defined in terms of pipeline-level components. For example, `pc`, conceptually the ‘current program counter’, is defined as the MEM stage program counter:

```
constant pc      === mem_pc.
```

A number of other instruction-level components are defined similarly:

```
constant ireg    === mem_ireg.
```

```
constant next_pc === alu_pc.
```

```

constant next_ireg === alu_ireg.

constant annul      === mem_annul.
constant in_bds    === mem_in_bds.
constant in_lds    === mem_in_lds.
constant load_reg  === mem_dest_reg.

```

The instruction-level abstracts from the details of register by-passing by defining the contents of an abstract set of CPU registers (`reg(R)` for `R` from 1 to 31 inclusive) as follows:

```

axiom def_reg
=== is_abs(R)
    and R : 0 upto 32
    => reg(R)^ = prev(run^);alu_get(R).

```

Note that the values of the abstract registers are defined to be as obtained from the bypass registers *at the ALU stage* (in the previous run cycle) of the current instruction. This ensures that the abstract register values at the current time are the initial values as seen by the current instruction. For example, referring to figure 2 on page 5, the abstract register values of the current instruction at cycle 5 (instruction $i + 1$), are obtained by accessing the bypass registers at cycle 4 (assuming no stalls). (Accessing the bypass registers at the start of the MEM stage would give the state as already modified by the ALU stage of the current instruction which is the initial state of the *next* instruction, i.e. instruction $i + 2$).

Axioms asserting processor behaviour at the instruction level (discussed below) are intended to be provable consequences of the pipeline-level specification, given the relationship between pipeline-level and instruction-level components illustrated above. While the bulk of the instruction-level specification remains at this time axiomatic, we have interactively verified a part of the instruction-level specification against the pipeline-level.

4.2 Functional instruction behaviour

As in the pipeline-level specification, functional and timing aspects are separated as much as possible. Functional axioms assert the processor state from run cycle to run cycle, while timing axioms determine how long it is to the next run cycle. This section illustrates aspects of the functional specification.

Because the current instruction is the one entering its MEM stage, the presence of a reset at any time since its fetch was initiated, three run cycles in the past, until its WB is completed, one run cycle in the future, invalidates conclusions about execution of the current instruction. Hence the following `irun` predicate is frequently used as a precondition for the execution of instructions.

```

constant irun
=== hw
  and run^
  and during(nth(run^, - 3);time,
             nth(run^, 1);time,
             not reset^
             ).

```

Instruction specific axioms assert the updating of the abstract processor registers using the following predicate.

```

function reg_put(Reg,Val)
=== (all x is_abs(x) and x : 1 upto 32
     => next(run^);reg(x)^ = if(x=Reg, Val, reg(x)^)).

```

Similarly, axioms assert that abstract processor registers remain unchanged by using the following predicates.

```

function reg_invar(T1,T2)
=== (all x is_abs(x) and x : 0 upto 32
     => at(T1);reg(x)^ = at(T2);reg(x)^).

```

```

abbreviation reg_invar === reg_invar(time, next(run^);time).

```

We consider now the following functional specification of the `lw` instruction in the case when the loaded word is in data cache.

```

axiom lw_instr_hit
=== ireg^ = lw_instr(Rt,Offset,Base)
  and irun
  and not annul
  and Addr = twos_comp(reg(Base)^ + int_val_n(Offset,16))
  and in_dcache(Addr)
  => reg_put(Rt, dval(Addr))
  and incr_next_pc
  and next(run^);(not in_bds^
  and next(run^);in_lds^
  and next(run^);load_reg^ = Rt
  and dcache_invar

```

Intuitively, `annul` is true if the current instruction is annulled as the result of an interrupt (and false otherwise). The predicate `incr_next_pc` specifies the updating of the program counter as follows:

```

constant incr_next_pc
=== next(run^);next_pc^ = twos_comp(next_pc^ + 4) .

```

Note that, as discussed in 2.2, at the time the `lw` instruction is decoded, fetching of the instruction at `next_pc^` has already been initiated. It is the program counter of the instruction following the next one (that is `next(run^);next_pc^`) that is determined by the current instruction.

Intuitively, `dcache_invar` asserts that the data cache is invariant until the next run cycle.

The following definition of the precondition of the load word is used by interrupt processing logic (not detailed here) to determine whether the instruction is to be an anulled due to an error caused by the instruction itself.

```

axiom lw_instr_pre
=== precond(lw_instr(Rt,Offset,Base))
   = (letabs x % Effective load address
      twos_comp(reg(Base)^ + int_val_n(Offset,16))
      within
        if(is_word_addr(Addr)
           and (user_mode => Addr < 2**31),
           no_error,
           error_addr_load)
        subject_to check_reg(Base)
      ).

```

The check `check_reg(Base)` simply checks that in a load delay slot the load target register is not being accessed:

```

function check_reg(A)
=== in_lds^
   => A \= load_reg^ .

```

The axioms `lw_instr_hit` and `lw_instr_pre` together encapsulate the functional behaviour of the `lw` under the stated conditions.

The following axiom for the branch instruction `beq` illustrates the functional specification of branch instructions.

```

axiom beq_instr
=== ireg^ = beq_instr(Rs,Rt,Offset)
    and irun
    and not annul
=> (letabs x    % Destination Address
      twos_comp(next_pc^ + int_val_n(Offset,16) * 4)
    within
      reg_invar
      and if(reg(Rs)^ = reg(Rt)^,
              next(run^);next_pc^ = x,
              incr_next_pc)
      and next(run^);in_bds^
      and next(run^);(not in_lds^)
      and dcache_invar
    ).

```

The above axioms define the behaviour of non-annulled instructions. Intuitively, instructions which *are* annulled leave the machine state unchanged:

```

axiom annulled_instr
=== irun
    and annul
=> reg_invar
    and next(run^);(not in_bds^)
    and next(run^);(not in_lds^)
    and dcache_invar

```

The following general axioms determine the updating of the program counter and the instruction register:

```

axiom next_pc
=== irun
=> next(run^);pc^ = next_pc^ .

```

```

axiom next_ireg
=== irun
=> next(run^);ireg^ = next_ireg^ .

```

4.3 Timing behaviour of instructions

Consider now the following timing specification of the `lw` instruction:

```

axiom lw_time_1
=== ireg^ = lw_instr(Rt,Offset,Base)
    and irun
    and not annul
    and Addr = twos_comp(reg(Base)^ + int_val_n(Offset,16))
    and in_dcachecache(Addr)
    and in_icachecache(next(run^);next_pc^)
    and group(next(run^);next_ireg^)\= hilo_instr
=> next(run^)= at(time+1).

```

This axiom asserts that, under the stated conditions, the next run cycle will be exactly one clock cycle in the future.

Note here the influence of instructions other than the current one on the timing of the current instruction. Specifically, to ensure that the current instruction takes one cycle, it is required that `next(run^);next_pc^` is in instruction cache. Intuitively, the instruction at the address `next(run^);next_pc^` is `rd_pc^`, the program counter of the instruction being fetched at the current time. For example in figure 2 (page 5), `next(run^);next_pc^` at cycle 4 is the address of instruction $i + 2$, being fetched at cycle 4 in the RD stage. If the required instruction is not in cache, then a stall will be caused, meaning the current instruction cannot complete in a single cycle. Note also that if the instruction after next is in the group of instructions which accesses the HI/LO registers, it *may* cause a stall, if the multiply/divide unit is busy, again meaning the current instruction cannot complete in one cycle.

Thus the above axiom contains as pre-conditions that the address determined by `next(run^);next_pc^` is in instruction cache, and that the corresponding instruction is not in the HI/LO group of instructions.

5 Conclusion

We began our work on modelling the real-time behaviour of RISC processors with the intention of modelling at the instruction level. However we found that only by modelling at the pipeline level initially and subsequently abstracting to the instruction level could we be confident of capturing all of the subtleties of instruction execution in the instruction-level model.

The instruction-level specification succeeds in hiding most of the detail of the pipeline level. However there are intrinsic limits to achieving a completely atomic and sequential model of instruction execution when *real-time* behaviour is modelled. The execution of instructions other than the current instruction may affect the real-time behaviour of the current instruction by causing stalls and thus delaying its completion.

The key idea in achieving the abstraction from the pipeline to the instruction level is to choose one pipeline stage as identifying the ‘current instruction’. The MEM stage was chosen for that role for the reasons given in 4.1.

We are currently using the instruction-level specification in the verification of a simple interrupt-driven real-time scheduler written in MIPS assembly language.

Acknowledgements

This work has been supported by the Australian Defence Science and Technology Organisation. Thanks also to Colin Fidge for helpful comments on this paper.

References

- [1] Henman, P. and Staples, J., ‘Introduction to Functional Set Theory’, Technical Report **144**, Key Centre for Software Technology, Department of Computer Science, University of Queensland, 1990, revised September 1991.
- [2] Gerry Kane and Joe Heinrich, *MIPS RISC Architecture*, Prentice Hall 1992.
- [3] Peter Kearney, John Staples and Abdu Abbas, ‘Functional Verification of Hard Real-Time Programs’, in *Algorithms, Software, Architecture*, ed. L. van Luewen, Information Processing 92, Volume I, North-Holland 1992, 113-119.
- [4] John Staples, Peter Robinson and Daniel Hazel, ‘A functional logic for higher level reasoning about computation’, to appear in *Formal Aspects of Computing*, 6, pages 1-38, 1994.
- [5] Mark Utting and Peter Kearney, ‘Pipeline Specification of a MIPS R3000 CPU’, Software Verification Research Centre Technical Report 92-6, October 1992, revised February 1994, 57 pages.
- [6] Mark Utting, ‘Instruction-level Specification of a MIPS R300 CPU’, Software Verification Research Centre Technical Report 93-26, February 1994, 27 pages.
- [7] Mark Utting and Keith Whitwell, ‘Ergo 4.0 Users Manual’, Software Verification Research Centre Technical Report 93-19, December 1993.