SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 92-6

Pipeline Specification of a MIPS R3000 CPU

Mark Utting and Peter Kearney

April 1994

Phone: +61 7 365 1003
Fax: +61 7 365 1533

Pipeline Specification of a MIPS R3000 CPU

Mark Utting and Peter Kearney

April 26, 1994

Abstract

This document contains a specification of the behavioural and real-time aspects of a typical MIPS R3000 RISC CPU. To increase assurance of correctness relative to the operational behaviour of the actual device, this specification directly specifies the five-stage pipeline structure of the CPU, rather than taking an instruction level view. A more abstract instruction level specification has been derived from this specification and is available as a separate report. Both specifications are written using *functional logic*, which extends classical logic by supporting reasoning about predicates that have an implicit parameter.

Contents

1 Introduction

This document contains a specification of the behavioural and real-time aspects of a typical MIPS R3000 RISC CPU. To increase assurance of correctness relative to the operational behaviour of the actual device, this specification directly specifies the five-stage pipeline structure of the CPU, rather than taking an instruction level view. An instruction level specification has been derived as an abstraction of the pipeline specification given here [Utt93]. Both specifications are written using *functional logic* [SRH89], which is reviewed briefly in Section 2.

Section 4 contains an overview of the specified system and a list of assumptions that the system must satisfy before this specification is applicable. The scope of the specification is discussed in Section 5.

Section 7 contains a specification of the system co-processor (CP0). This co-processor is physically part of the R3000 CPU chip, but is conceptually an independent processor that performs memory management and handles exceptional events. The individual stages of the R3000 pipeline, plus the registers and control logic that integrate them into a cohesive unit are all specified in Section 8.

Section 9 contains specifications of the instruction and data caches and Section 10 contains a specification of how the CPU interacts with devices attached to the external system bus. Currently, the specification does not describe any particular devices on the external system bus, but they could easily be added.

Appendix A contains definitions of the cache and memory interfaces that are used throughout the specification and Appendix B contains a definition of the function that is used to translate virtual addresses into physical addresses. Appendix C contains a brief discussion of how the specification of the CPU interfaces relates to the detailed electrical timing diagrams of a typical R3000 system.

2 Functional Logic

This section presents a brief review of functional logic and its use for reasoning about real-time systems. The reader is referred to [SRH89] and [KSAL91] for fuller accounts.

The basic idea of functional logic is to reason classically about assertions that have an implicit parameter. For example, an assertion such as `x = y` essentially denotes a boolean valued function

$$(\lambda i \bullet x(i) = y(i))$$

rather than an individual boolean value. The set of implicit parameters is called the *index* set. The index set includes a value called *bad* (or *?*) which represents undefinedness. All expressions in functional logic denote total functions on the index set and partial functions on the index set are representable by using *bad* as a function value. Such functions are required to be strict with respect to *bad*, since that is sufficient for the representation of all partial functions. Terms that return the same value for all indices (other than *bad*) are called *absolute* terms.

In this document, the implicit parameters of most interest when reasoning about real-time systems are 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. We take our basic unit of time to be the clock cycle time of the MIPS R3000 CPU. Thus, this specification is effectively parameterised by the clock speed of the CPU.

In this paper, this theory of real-time systems 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. However, that is beyond the scope of this paper.

Table 1 lists the operator symbols used in this specification, ordered by their binding power. Note that `A upto B` constructs the set of integers between `A` and `B`, including `A` but excluding `B`. Table 2 lists the predefined functions used in this specification, with an informal description of their semantics. Formal definitions can be found in [KSAL91]. In this paper we distinguish definitions from axioms by the use of different keywords. Definitions are written in the form

```
    function f(Arg1,Arg2,Arg3) ===  Body.
```

Such definitions indicate that `Body` does not contain recursive calls on `f`, there is no mutual recursion between `f` and other definitions or axioms in the paper and there is only one definition of `f` in the paper.

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

Operator	Description
`^`	(postfix) dereference a location
`;`	forward functional composition
`**`	exponentiation

`*,/,mod,bitand,`
`bitor,<<,>>`
	arithmetic, bitwise logical
	and shift operators

`+,-`	arithmetic operators
`upto`	constructs a set of integers
`<,>,=<,>=,=,\=,is`	relational operators
`not`	logical negation
`and`	logical conjunction
`or`	logical disjunction
`=>`	logical implication
`<=>`	logical equivalence
`all,ex`	logical quantifiers
`letabs ... within ...`	local definition
`===`	definition

Table 1: Operator Symbols, ordered by binding power (with the tightest binding operator at the top).

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

Informally, this has the effect of evaluating `Expr` and declaring `Var` to be an object variable that is local to `Body`, with a value that is absolute and is equal to the value of `Expr`. More formally, a let construct such as

```
letabs Var Expr within Body
```

is an abbreviation for

```
(some_i x (exists Var
    Var = Expr
    and is_abs(Var)
    and Body
))
```

where `some_i` is an indexed choice quantifier [SRH89].

Function	Description
`is_abs(V)`	`V` is an absolute value.
`int(V)`	`V` is an integer (i.e., `V:int`).
`locs`	The set of locations.
`times`	The set of valid times.
`time`	The current time.
`at(T)`	An action that changes the current time to be `T`.
`next_time(P)`	The first time strictly later than the current time at which the predicate `P` is true, or `?` if there is no such time.
`prev_time(P)`	The most recent time strictly earlier than the current time at which the predicate `P` is true, or `?` if there is no such time.
`next(P)`	Equals `at(next_time(P))`
`prev(P)`	Equals `at(prev_time(P))`
`during(T1,T2,P)`	`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.
`until(T,P)`	Equals `during(time,T,P)`
`if(Cond,A,B)`	Equals `A` if `Cond` evaluates to true, otherwise it equals `B`.

Table 2: Predefined Specification Functions.

3 Computer Arithmetic

The values manipulated by a MIPS processor can be viewed as bitstrings, as unsigned positive numbers or as two's complement numbers. The hardware does not force any type distinction between these views and programs can switch from one view of a given value to another view without any explicit coercion.

In this specification, we get the same effect by modelling bitstrings and unsigned and signed computer numbers as a subset of the positive integers. For example, 32 bit bitstrings and numbers are all represented as positive integers in the range $0 \cdots 2^{32} - 1$. The following functions are useful for converting between two's complement values and mathematical integers.

```
/* the set of N-bit computer values */
function bits(N)
=== 0 upto 2**N.

/* the set of mathematical integers that can be
```

```
 * represented as N-bit twos-complement numbers.
 */
function twos_comp_range(N)
=== - 2**(N-1) upto 2**(N-1).

function twos_comp_n(I,N)
=== I + (some_i x int(x)
                  and x mod 2**N = 0
                  and (I + x) : bits(N))
    subject_to
        int(N)
        and 2 =< N
        and int(I).

function int_val_n(C,N)
=== C + if(C < 2**(N-1), 0, - 2**N)
    subject_to
        int(N)
        and 2 =< N
        and C:bits(N).

/* 32 bit values are the most common case */
function twos_comp(I)
=== twos_comp_n(I,32).

function int_val(C)
=== int_val_n(C,32).
```

4 System Overview

Figure 1 is a block diagram of the system that is specified in this document. 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.

The R3000 uses direct-mapped, write-through caches that are indexed by a physical address [KH92]. Each cache location contains a triple of the form

```
    triple(Data,Tag,Valid)
```

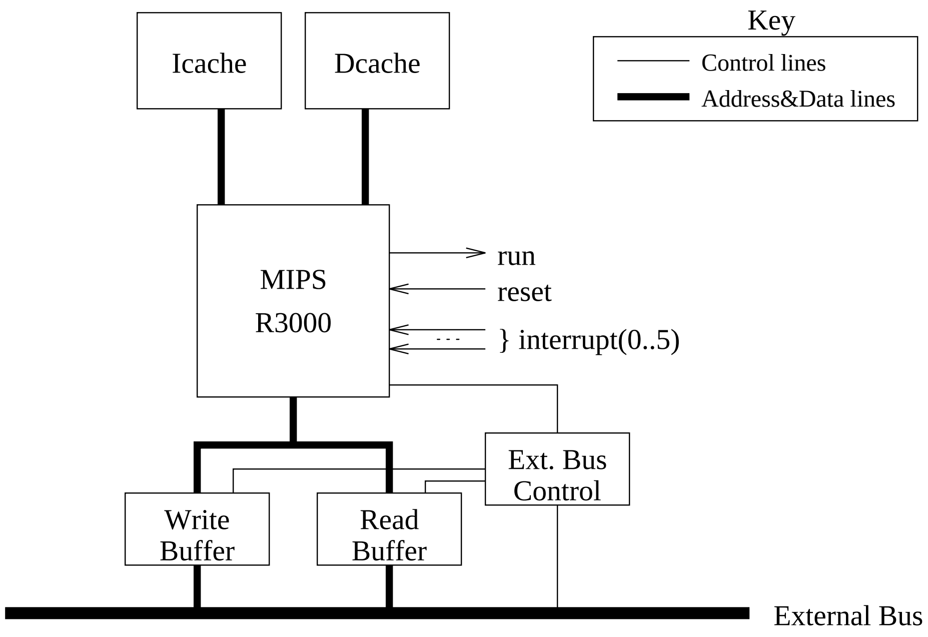


Figure 1: Block Diagram of the Specified System.

where `Data` is a 32-bit value, `Tag` is the address of that value (only the high order bits are stored) and `Valid` is a flag that is set to false when the data-tag pair needs to be updated from external memory. The following two axioms assert *localness* properties of the `triple` function, which express that `triple` is effectively a lifting to the functional level of a triple construction at the index level. The third axiom defines equality of triples in the usual way.

```
function triple(A,B,C).

axiom triple_local
=== T;triple(A,B,C)
    =   triple(T;A, T;B, T;C).

axiom triple_localplaces
=== A = D
    and B = E
    and C = F
    => triple(A,B,C) = triple(D,E,F).

% This subsumes triple_localplaces.
axiom triple_eq
```

```
=== triple(A,B,C) = triple(D,E,F)
    <=> (A = D)
        and (B = E)
        and (C = F).
```

We also define accessor functions for triples, using the `absolute_function` command which declares the function and adds the usual localness axioms (`X_local` and `X_localplaces`).

```
absolute_function triple1(Triple).
absolute_function triple2(Triple).
absolute_function triple3(Triple).
axiom triple1  ===  triple1(triple(A,B,C)) = A.
axiom triple2  ===  triple2(triple(A,B,C)) = B.
axiom triple3  ===  triple3(triple(A,B,C)) = C.
```

The interfaces between the CPU and the caches and the external bus are fully defined in Appendix A. Table 3 gives a summary of the predicates that relate to these three interfaces of the CPU. For example, the CPU could initiate a data cache read by asserting `dcache_read(A,C)`, where `A` is the result of the virtual to physical address translation and `C` determines whether or not that address is cacheable. The data cache would then assert `dcache_read_result(Loc^)`, where `Loc` is the cache location associated with the physical address being read, and the `dcache_readhit` predicate would then determine whether or not the cache read was successful.

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 `reset` and `interrupt(i)` flags are controlled by external events and read by the CPU. These flags are all defined in Section 7.

4.1 Assumptions

The work in this paper applies to R3000 based systems which satisfy the following assumptions. These assumptions are realisable using some of the presently available implementations of the R3000 architecture, together with suitable system design.

Interface	Predicate

Data Cache	`dcache_read(PhyAddr,Cacheable)`
	`dcache_read_result(Result)`
	`dcache_readhit`
	`dcache_write(PhyAddr,Data)`
	`dcache_inactive`

Instr. Cache	`icache_read(PhyAddr,Cacheable)`
	`icache_read_result(Result)`
	`icache_readhit`
	`icache_inactive`

External Bus	`ext_read(Paddr,Size)`
	`ext_write(Paddr,Data,Size)`
	`ext_inactive`
	`e_readbusy^`
	`e_writebusy^`

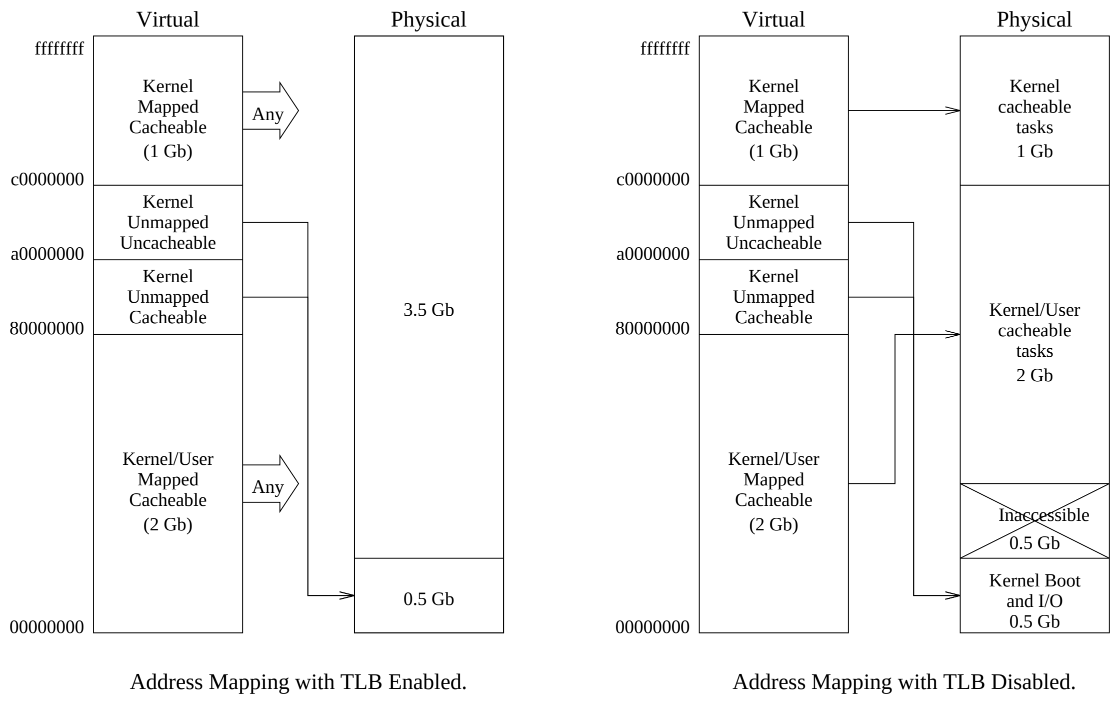
Table 3: CPU Interface Predicates.

1. The translation lookaside buffer (TLB)[1] is disabled, with a fixed mapping from virtual to physical addresses being used instead. For example, the IDT79R3500A CPU [IDT91] allows the TLB to be disabled by selecting an initialisation option when the processor is reset. Figure 2 shows the virtual to physical address mapping used by the IDT79R3500A when the TLB is enabled and when it is disabled. This mapping is specified in Appendix B.

   As well as avoiding the complexity of specifying the TLB, this assumption has the advantage of avoiding micro-TLB stalls. The micro-TLB is a two entry cache of the main TLB that is used to speed up the translation of instruction addresses. Updating the micro-TLB from the main TLB causes a stall of a single cycle. However, when the TLB is disabled, the micro-TLB does not cause the processor to stall.

2. The CPU is not stalled by any other processors or DMA controllers. The main reason for making this assumption is that multiprocessor support varies between different R3000 implementations, thus making a general purpose specification difficult.

[1]The TLB is a 64 entry content addressable cache that is normally used by the CP0 coprocessor to perform virtual to physical address translation.



Virtual
Physical
ffffffff
Kernel Mapped Cacheable (1 Gb)
Any
c0000000
Kernel Unmapped Uncacheable
a0000000
Kernel Unmapped Cacheable
80000000
3.5 Gb
Kernel/User Mapped Cacheable (2 Gb)
Any
0.5 Gb
00000000
Address Mapping with TLB Enabled.
Virtual
Physical
ffffffff
Kernel Mapped Cacheable (1 Gb)
Kernel cacheable tasks 1 Gb
c0000000
Kernel Unmapped Uncacheable
a0000000
Kernel Unmapped Cacheable
80000000
Kernel/User cacheable tasks 2 Gb
Kernel/User Mapped Cacheable (2 Gb)
Inaccessible 0.5 Gb
Kernel Boot and I/O 0.5 Gb
00000000
Address Mapping with TLB Disabled.

Figure 2: Address Translation in the IDT79R3500A CPU (taken from [IDT91, page 5.5.3]).

3. The instruction and data caches are used as the main memory of the system, rather than as caches of a larger external memory. This is appropriate for real-time systems, since it eliminates the majority of memory stalls caused by cache misses. Most R3000 systems have a write-through cache, which means that data writes are sent to the external bus as well as to the data cache. To avoid stalling the CPU on every write, a *write buffer* is typically used to decouple the CPU from the slower external bus. The write buffer captures writes from the CPU, puts them into a queue, then dequeues each write onto the slower external bus when it is available. The write buffer stalls the CPU if its queue is full and another write is attempted. However, this use of a write buffer complicates the timing aspects of store instructions, because the number of stall cycles caused by a given store instruction is dependent upon how full the write queue is, and this is in turn dependent upon how many other writes have been performed in the recent past.

   In this specification we assume a simpler write interface to the external bus. We assume that writes to uncacheable addresses cause the CPU to be stalled until the write is completed, whereas writes to cacheable addresses will be ignored and will not stall the CPU. This means that writes to uncacheable addresses (`0xa0000000 to 0xbfffffff`) are sent only to the external bus, while writes to cacheable addresses (all others) are sent only to the data cache. The main advantages of such a system are that cacheable writes (usually the majority) are always fast (and never cause stalls) and the number of stalls caused by a store instruction depends only upon the address being written to, so real-time performance is more predictable.

This assumption means that our specification is not directly applicable to R3000 implementations that include a write buffer on-chip, such as the IDT 3051 family [IDT91]. However, the majority of R3000 implementations rely upon an off-chip write buffer, which makes it possible to implement the simple design discussed above. Section 10 contains more details about the write buffer.

4. We also assume that the instruction and data caches are distinct from one another and that the mapping of physical addresses into cache locations is fixed. That is, the mapping does not change with time and is determined purely by the *AddrLo* pins of the CPU. Most system designs satisfy these assumptions.

5. In the event that this specification is extended to include partial-word store instructions (*e.g.*, store halfword or store byte) it would be desirable for the CPU to support a *storepartial* mode of operation [LC91], which allows partial words to be written direct to cache using a read-modify-write sequence. Most R3000 implementations do support such a mode and allow it to be selected during reset.[2]

6. This specification assumes a simple interaction with the external bus. We assume that all reads from the asynchronous bus are single word reads (most R3000 implementations allow this option to be selected by asserting an input pin during each read), so complex bus interactions such as burst reads (which allow several consecutive words of cache to be refilled quickly) do not occur. Parity errors and read retries are assumed not to occur. It would be possible to extend this specification to handle these kinds of errors, but such extensions are beyond the scope of our current project.

   We also assume that the external bus interface includes timeout hardware which generates a bus error if an external read or write request is not sat-

[2]In earlier MIPS implementations, based on the MIPS R2000, a partial word write to a cacheable address always caused that cache location to be marked as invalid. In such systems, partial word stores would have to be avoided (*e.g.*, by simulating them in software) if cache memory was to be used as the main memory of the system.

isfied within some reasonable time limit. This avoids the possibility of the R3000 stalling indefinitely.

5 Specification Scope

The specification is incomplete in a variety of ways, since our approach has been to specify only those features that are likely to be useful in the kinds of programs that we wish to verify. The following list indicates the main areas where the specification is incomplete.

1. It specifies only the execution of programs that are contained within the instruction cache. That is, no axioms are given for instruction cache misses. This means that the programs being verified are assumed to be executing after some initialisation phase which has loaded all the necessary code into the instruction cache.
2. Only about twenty of the most common instructions are specified, though others could easily be added.
3. The specification of the system co-processor (CP0) is minimal, but covers just enough of its functionality to enable simple interrupt handling routines to be verified.
4. Co-processors (including the floating point co-processor) are not supported by this specification. Hence, floating point instructions are not supported.

The specification could be extended to cover most of these areas simply by adding axioms that specify additional aspects of the system's behaviour and adding extra locations if necessary.

6 Instruction Decoding

The MIPS instruction set architecture has three basic instruction formats, as shown in Figure 3. The initial `op` field determines which of the three formats should be used to interpret the remainder of the instruction. For example, when the `op` field is zero, the R format is used and a second opcode field (`Funct`) determines the function to be performed.

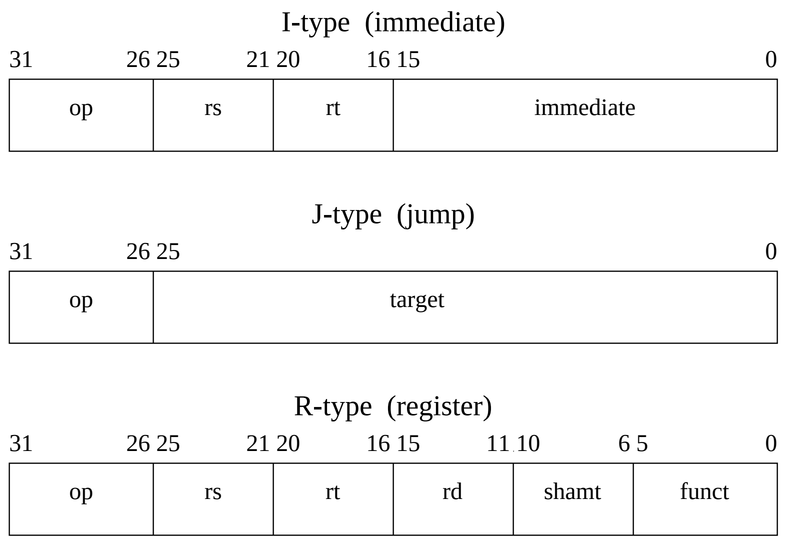


Figure 3: MIPS Instruction Formats [KH92, Page 3-1].

The following functions are used to encode each of these formats. The terminology used for each field is the same as in the MIPS documentation [KH92]: the variables `Rs` and `Rd` stand for *source* and *destination* register numbers, respectively, and `Rt` stands for a *target* register. The target register field is usually used as a second source register field, but is sometimes used to hold the destination register number for I-type instructions. The `Shamt` field of R-type instructions gives a constant shift amount for some shift instructions and contains zero for most other R-type instructions.

```
function j_type(Op,Target)
=== (Op << 26) + Target
    subject_to
        Op:bits(6)
        and Target:bits(26).

function r_type(Rs,Rt,Rd,Shamt,Funct)
=== %% bits 26..31 are always zero.
    (Rs      << 21)
    + (Rt    << 16)
    + (Rd    << 11)
    + (Shamt << 6)
    + Funct
    subject_to
        Rs:bits(5)
        and Rt:bits(5)
        and Rd:bits(5)
        and Shamt:bits(5)
        and Funct:bits(6).

function i_type(Op,Rs,Rt,Immed)
=== (Op << 26)
    + (Rs << 21)
    + (Rt << 16)
    + Immed
    subject_to
        Op:bits(6)
        and Rs:bits(5)
        and Rt:bits(5)
        and Immed:bits(16).

function add_instr(Rd,Rs,Rt)       === r_type(Rs,Rt,Rd,0,32).
function addi_instr(Rt,Rs,Immed)   === i_type( 8,Rs,Rt,Immed).
function addiu_instr(Rt,Rs,Immed)  === i_type( 9,Rs,Rt,Immed).
function addu_instr(Rd,Rs,Rt)      === r_type(Rs,Rt,Rd,0,33).
function and_instr(Rd,Rs,Rt)       === r_type(Rs,Rt,Rd,0,36).
function andi_instr(Rt,Rs,Immed)   === i_type(12,Rs,Rt,Immed).
function beq_instr(Rs,Rt,Offset)   === i_type( 4,Rs,Rt,Offset).
function bne_instr(Rs,Rt,Offset)   === i_type( 5,Rs,Rt,Offset).
function j_instr(Target)           === j_type( 2,Target).
function jal_instr(Target)         === j_type( 3,Target).
function jalr_instr(Rd,Rs)         === r_type(Rs,0,Rd,0,9).
function jr_instr(Rs)              === r_type(Rs,0,0,0,8).
function lui_instr(Rt,Immed)       === i_type(15,0,Rt,Immed).
function lw_instr(Rt,Offset,Base)  === i_type(35,Base,Rt,Offset).
function mfc0_instr(Rt,Rd)         === i_type(16,0,Rt,Rd << 10).
function mtc0_instr(Rt,Rd)         === i_type(16,4,Rt,Rd << 10).
function or_instr(Rd,Rs,Rt)        === r_type(Rs,Rt,Rd,0,37).
function ori_instr(Rt,Rs,Immed)    === i_type(13,Rs,Rt,Immed).
constant rfe_instr                 === j_type(16,(1 << 25) + 16).
function sll_instr(Rd,Rt,Shamt)    === r_type(0,Rt,Rd,Shamt,0).
function sw_instr(Rt,Offset,Base)  === i_type(43,Base,Rt,Offset).
function syscall_instr(Code)       === j_type(0,(Code << 6) + 12).

% the recommended nop instruction.
function nop_instr                 === sll_instr(0,0,0).
```

Although we have specified the instruction encoding for the MTC0 instruction, this specification does not yet define its behaviour. It has a large number of hazards that vary according to which CP0 register is being updated.

6.1 Instruction Groups

The documentation of the MIPS instruction set architecture classifies instructions into six groups [KH92, Page 2-7].

1. Load and Store Instructions
2. Computational Instructions
3. Jump and Branch Instructions
4. Special Instructions (these allow the software to initiate traps)
5. Coprocessor Instructions (*e.g.*, floating point instructions). These have coprocessor-dependent instruction formats which are not covered by this specification.
6. Coprocessor Zero Instructions. These perform operations on CP0 registers and control the memory management and exception handling facilities of the processor.

In this specification, we split the load and store instructions into separate groups and restrict group five to those instructions that relate to coprocessors other than CP0, thus ensuring that the groups are mutually exclusive. We also split the computational instructions into two groups (`comp_instr` and `hilo_instr`) according to whether or not they use the HI/LO registers. The HI and LO registers are two special purpose program-visible registers that are used for the results of multiplication and division operations. The processor automatically inserts stall cycles when a HI or LO register is accessed before a multiplication or division is complete. The `hilo_instr` group of instructions must include all multiplication and division instructions and all instructions that move values in or out of the HI or LO registers. So far, this specification does not specify any such instructions, but the use of the `hilo_instr` group allows them to be added without introducing inconsistency, provided they are defined as belonging to the `hilo_instr` group. The following constants are used to identify each group.

```
constant load_instr    === 1.
constant store_instr   === 2.
constant branch_instr  === 3.
constant comp_instr    === 4.
constant hilo_instr    === 5.
constant cp123_instr   === 6.
constant cp0_instr     === 7.
constant special_instr === 8.
constant illegal_instr === 9.
```

The following `group` function determines which group each instruction belongs to.

```
absolute_function group(Instr).

axiom add_group
=== add_instr(Rd,Rs,Rt) : ints
    => group(add_instr(Rd,Rs,Rt)) = comp_instr.

axiom addi_group
=== addi_instr(Rt,Rs,Imm) : ints
    => group(addi_instr(Rt,Rs,Imm)) = comp_instr.

axiom addiu_group
=== addiu_instr(Rt,Rs,Imm) : ints
    => group(addiu_instr(Rt,Rs,Imm)) = comp_instr.

axiom addu_group
=== addu_instr(Rd,Rs,Rt) : ints
    => group(addu_instr(Rd,Rs,Rt)) = comp_instr.

axiom and_group
=== and_instr(Rd,Rs,Rt) : ints
    => group(and_instr(Rd,Rs,Rt)) = comp_instr.

axiom andi_group
=== andi_instr(Rt,Rs,Imm) : ints
    => group(andi_instr(Rt,Rs,Imm)) = comp_instr.

axiom beq_group
=== beq_instr(Rs,Rt,Off) : ints
    => group(beq_instr(Rs,Rt,Off)) = branch_instr.

axiom bne_group
```

```
=== bne_instr(Rs,Rt,Off) : ints
    => group(bne_instr(Rs,Rt,Off)) = branch_instr.

axiom j_group
=== j_instr(Target) : ints
    => group(j_instr(Target)) = branch_instr.

axiom jal_group
=== jal_instr(Target) : ints
    => group(jal_instr(Target)) = branch_instr.

axiom jalr_group
=== jalr_instr(Rd,Rs) : ints
    => group(jalr_instr(Rd,Rs)) = branch_instr.

axiom jr_group
=== jr_instr(Rs) : ints
    => group(jr_instr(Rs)) = branch_instr.

axiom lui_group
=== lui_instr(Rt,Imm) : ints
    => group(lui_instr(Rt,Imm)) = comp_instr.

axiom lw_group
=== lw_instr(Rt,Off,Base) : ints
    => group(lw_instr(Rt,Off,Base)) = load_instr.

axiom mfc0_group
=== mfc0_instr(Rt,Rd) : ints
    => group(mfc0_instr(Rt,Rd)) = cp0_instr.

axiom mtc0_group
=== mtc0_instr(Rt,Rd) : ints
    => group(mtc0_instr(Rt,Rd)) = cp0_instr.

axiom or_group
=== or_instr(Rd,Rs,Rt) : ints
    => group(or_instr(Rd,Rs,Rt)) = comp_instr.

axiom ori_group
=== ori_instr(Rt,Rs,Imm) : ints
```

```
    => group(ori_instr(Rt,Rs,Imm)) = comp_instr.

axiom rfe_group
=== rfe_instr : ints
    => group(rfe_instr) = cp0_instr.

axiom sll_group
=== sll_instr(Rd,Rt,Shamt) : ints
    => group(sll_instr(Rd,Rt,Shamt)) = comp_instr.

axiom sw_group
=== sw_instr(Rt,Off,Base) : ints
    => group(sw_instr(Rt,Off,Base)) = store_instr.

axiom syscall_group
=== syscall_instr(Code) : ints
    => group(syscall_instr(Code)) = special_instr.
```

7 CP0 Registers

This section specifies those aspects of the CP0 co-processor that we are interested in.

We define three of the CP0 (co-processor zero) registers:

- the *Exception Program Counter* (EPC) register is a 32-bit, read-only register that contains the address where processing should resume after an interrupt has been handled. When an interrupt occurs, the EPC is usually set to the address of the interrupted instruction, but if the interrupted instruction is in a branch delay slot, the EPC is set to the preceding instruction (*i.e.*, the branch instruction).
- the *Status* register contains a mask bit for each of the external interrupts (1 means enabled), a bit called CU0 that determines whether or not CP0 is usable from user mode, a three level stack of a pair of bits called KU and IE, plus several additional fields that we ignore in this specification. The current KU bit determines whether the CPU is in kernel (KU=0) or user (KU=1) mode, while the current IE bit determines whether interrupt processing is enabled (IE=1) or disabled (IE=0).

- the *Cause* register contains a five bit code that specifies the reason for the most recent interrupt (*e.g.*, zero for external interrupts), a bit that reflects whether or not each external interrupt is asserted, plus additional fields that we shall ignore.

```
absolute_constant cp0_epc    :locs.
absolute_constant cp0_status :locs.
absolute_constant cp0_cause  :locs.
```

We also define locations that represent the external interrupt inputs to the processor, the `reset` input flag and the `run` output flag.

```
absolute_constant run        :locs.
absolute_constant reset      :locs.
absolute_function interrupt(N:0 upto 6):locs.
```

The following definitions and axiom specify how various fields are encoded into the `cp0_cause` register. For convenience, we define a predicate, `bitset(Val,N)`, that is true iff bit `N` of `Val` is set to 1.

```
function bitset(Val,N)
=== (Val >> N) mod 2 = 1.

constant cp0_cause_code
=== (cp0_cause^ >> 2) mod 2**5.

function int_asserted(N)
=== N : (0 upto 6)
    and bitset(cp0_cause^, N+10).

axiom cp0_cause_interrupt
=== N : (0 upto 6)
    => bitset(cp0_cause^, N+10) = interrupt(N).
```

Similarly, the next group of definitions specify various fields of the `cp0_status` register.

```
/* The "use-bootstrap-vector" flag */
constant cp0_status_bev
=== bitset(cp0_status^, 22).
```

```
function int_masked(Intnum)
=== Intnum : 0 upto 6
    and bitset(cp0_status^, Intnum + 10).

/* the three level stack of IE and KU flags */
constant ints_enabled      === bitset(cp0_status^, 0).
constant ints_enabled_prev === bitset(cp0_status^, 2).
constant ints_enabled_old  === bitset(cp0_status^, 4).

constant user_mode         === bitset(cp0_status^, 1).
constant user_mode_prev    === bitset(cp0_status^, 3).
constant user_mode_old     === bitset(cp0_status^, 5).

constant kernel_mode
=== not user_mode.

constant cp0_usable
=== bitset(cp0_status^,28) or kernel_mode.
```

The following constants define the various kinds of exception codes that can appear in the `cp0_cause_code` field of the CP0 Cause register (see Table 6-2 of the MIPS architecture book [KH92]).

```
constant error_interrupt        === 0.
/* errors 1..3 are TLB errors */
constant error_addr_load        === 4.
constant error_addr_store       === 5.
constant error_bus_instr        === 6.
constant error_bus_data         === 7.
constant error_syscall          === 8.
constant error_breakpoint       === 9.
constant error_reserved_instr   === 10.
constant error_coproc_unusable  === 11.
constant error_overflow         === 12.
/* error numbers 13..31 are not applicable to the R3000 */

constant no_error               === 32.
constant internal_errors        === 1 upto 32.
```

It is useful to define a predicate called `int_pending` which is true whenever there are external interrupts asserted that should be processed in the next run cycle.

```
constant int_pending
=== ints_enabled
    and (ex x x : 0 upto 6
             and int_asserted(x)
             and not int_masked(x)).
```

Instructions that manipulate CP0 registers (e.g., MFC0) refer to the CP0 registers by number. The following definitions allow us to use symbolic names for these numbers. The `cp0_reg` function maps these CP0 register numbers to the corresponding locations.

```
constant cp0_status_reg === 12.
constant cp0_cause_reg  === 13.
constant cp0_epc_reg    === 14.

absolute_function cp0_reg(Regnum:0 upto 32):locs.

axiom cp0_reg_status
=== cp0_reg(cp0_status_reg) = cp0_status.

axiom cp0_reg_cause
=== cp0_reg(cp0_cause_reg)  = cp0_cause.

axiom cp0_reg_epc
=== cp0_reg(cp0_epc_reg)    = cp0_epc.
```

The following predicate determines the effect of non-CP0 instructions on the above CP0 registers. Note that the `int_asserted` bits of the `cp0_cause` register do not necessarily remain constant, because they always reflect the status of the external interrupt signals.

```
function cp0_registers_invar(T1,T2)
=== at(T1);cp0_status^       = at(T2);cp0_status^
    and at(T1);cp0_epc^       = at(T2);cp0_epc^
    and at(T1);cp0_cause_code = at(T2);cp0_cause_code.

constant cp0_registers_invar
=== cp0_registers_invar(time,next(run^);time).
```

The following definitions give the addresses of interrupt routines within R3000 systems.

```
constant reset_excep_addr
=== seg(7) - meg(4).

constant gen_excep_addr
=== if(cp0_status_bev,
        seg(7) - meg(4) + 256,
        seg(4))
    + 128.
```

8 The R3000 CPU

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

Thus, while instruction i is being fetched, instruction $i-1$ is being decoded and the three previous instructions are in various stages of execution. In the ideal case, this 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, it is often necessary to *stall* the pipeline for one or more cycles, because of dependencies between instructions, branch instructions and various other reasons.

In a typical R3000 CPU there are up to eight different kinds of stalls, but our simplifying assumptions mean that only five kinds of stalls need to be considered:

- Partial Word Store Stalls,
- Data Cache Misses,
- Write Busy Stalls,
- Instruction Cache Misses and
- Multiplier/Divider Busy Stalls.

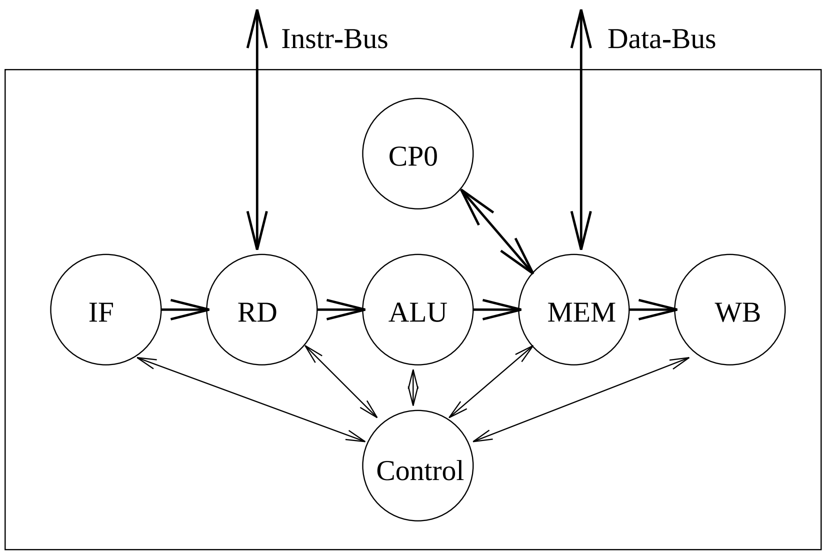


Figure 4: Simplified Diagram of the Specified CPU Structure

It is useful to define the following predicates to identify when each kind of stall occurs.

```
constant partword_stall.
constant d_miss_stall.
constant e_writebusy_stall.
constant i_miss_stall.
constant muldiv_stall.
```

Since these are the only stalls that need to be considered, when they are all false, no stalls are occurring, so the current cycle is a run cycle.

```
axiom def_run
=== run^
    <=> not partword_stall
        and not d_miss_stall
        and not e_writebusy_stall
        and not i_miss_stall
        and not muldiv_stall.
```

Figure 4 shows a simple block diagram of the internal pipeline structure of an R3000 CPU, as specified in this document. Sections 8.1 through to 8.5 specify the pipeline stages from IF to WB. Section 8.6 specifies the overall control logic of the CPU (e.g., interrupt handling and pipeline stalling).

To specify the behaviour of each pipeline stage, it is necessary to postulate some registers for holding intermediate results between the pipe stages. These registers are typically not mentioned in R3000 user manuals (e.g. [KH92]) because those manuals abstract away from the internal details of the pipeline structure. However, the registers must be represented in some form within hardware implementations of the R3000 pipeline in order for the pipeline to achieve the correct behaviour. For example, the program counter of each instruction must be carried along the pipeline as the instruction is executed, so that the address of an interrupted instruction can be determined. Our specification of the pipeline models this by postulating a set of program counter registers, `*_pc`, where `*` is the name of a pipe stage.

```
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.

```
absolute_constant alu_ireg   :locs.
absolute_constant mem_ireg   :locs.
```

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.

```
absolute_constant alu_in_bds :locs.
absolute_constant mem_in_bds :locs.
absolute_constant wb_in_bds  :locs.

absolute_constant alu_in_lds :locs.
absolute_constant mem_in_lds :locs.
```

The overall CPU control logic (see Section 8.6) maintains an `*_annul` flag for each stage, which determines whether the instruction should be annulled or executed as normal.

```
absolute_constant if_annul   :locs.
absolute_constant rd_annul   :locs.
absolute_constant alu_annul  :locs.
absolute_constant mem_annul  :locs.
absolute_constant wb_annul   :locs.
```

Each stage (except WB) outputs an error value (`*_error`) to signal whether or not an error occurred during that stage and if so, which error.

```
absolute_constant rd_error   :locs.
absolute_constant alu_error  :locs.
absolute_constant mem_error  :locs.
absolute_constant wb_error   :locs.
```

The ALU and MEM stages use pairs of *bypass* registers, that capture the destination register number and its new value so that the updating of the main register file can be postponed until the WB stage.

```
absolute_constant mem_dest_reg       :locs.
absolute_constant mem_dest_val       :locs.
absolute_constant wb_dest_reg        :locs.
absolute_constant wb_dest_val        :locs.
```

The following locations correspond to the general purpose registers of the CPU. These 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.
```

Many of the axioms in this specification refer to a predicate called `hw` that plays the same role here as the predicate `achievable` did in [KSAL91]. The `hw` predicate 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 the current cycle to be a run cycle and the `reset` flag to be false until the next run cycle (because the reset behaviour is quite different to normal execution).

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

The function `next(run^)` is used in many axioms to increment the time to the next run (*i.e.*, non-stall) cycle. This function is always well defined, because our assumptions ensure that there will always be a run cycle sometime in the future.

```
axiom next_run_good
=== hw => (next(run^) \= ?).
```

8.1 IF Stage

The main activities of the IF stage are to

- increment the program counter to the next instruction,
- translate that (virtual) address to a physical address using a fixed mapping (because we are not specifying a TLB) and
- initiate the instruction fetch from the instruction cache.

The fixed virtual to physical address mapping is a parameter to the specification and is defined by two functions called `physical_addr` and `cacheable`, which are defined in Appendix B. These functions correspond to the function called *AddrTranslation* in the MIPS documentation [KH92].

It is the ALU stage of every instruction that sets the next value of the `rd_pc` location. 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 partial cycles, we weaken the specification of the ALU stage so that it is only required to calculate the new `rd_pc` value by the *end* of the cycle. The only effect of this weaker specification is that the following IF axiom accesses the `rd_pc` location at the end of the cycle rather than during it, so appears to be instantaneous.

```
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^,
                  icache_inactive,
                  icache_read(physical_addr(rd_pc^),
                              cacheable(rd_pc^)))
          ).

axiom if_stage_error
=== running
    and not if_annul^
    and not is_word_addr(next(run^);rd_pc^)
    => next(run^);
          (rd_error^ = error_addr_load
           and icache_inactive
          ).
```

Note that although instruction address translation is performed during the IF stage, testing that the instruction address is a valid user mode address is deferred until the end of the instruction's ALU stage, since the preceding instruction may modify the `user_mode` flag during its MEM stage.

8.2 RD Stage

In a MIPS R3000 processor, the main three activities that occur during 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 (see Section 9), with the results being placed into locations `i_data`, `i_tag` and `i_valid`.

The specification does not describe the decoding of instructions. This avoids the need to define new instruction formats for decoded instructions.

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. As mentioned earlier (Section 5), we do not define how instruction cache misses are handled.

```
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^ .
```

The next few axioms specify the value of the `alu_error` output of the RD stage. This is calculated at the end of the RD stage, after the instruction has been decoded. Although a co-processor instruction may generate a *co-processor unusable* exception, this condition is checked at the end of the instruction's ALU stage, rather than here in the RD stage, since the preceding instruction may modify the `user_mode` flag during its MEM stage.

```
axiom rd_no_error
=== running
    and not rd_annul^
    and not group(next(run^);alu_ireg^) = special_instr
    and not group(next(run^);alu_ireg^) = illegal_instr
    =>  next(run^);alu_error^ = no_error.

axiom rd_error_syscall
=== running
    and not rd_annul^
    and next(run^);alu_ireg^ = syscall_instr(Code)
    =>  next(run^);alu_error^ = error_syscall.

axiom rd_error_reserved_instr
=== running
    and not rd_annul^
    and group(next(run^);alu_ireg^) = illegal_instr
    =>  next(run^);alu_error^ = error_reserved_instr.
```

8.2.1 Instruction Cache Misses

As stated earlier, the code that we wish to verify will all be contained within the instruction cache, so the following axiom does not specify the behaviour of the CPU for instruction cache misses.

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

8.2.2 Multiply/Divide Unit Stalls

Instructions that do not access the HI or LO registers, or that have been annulled, do not cause multiply/divide stalls.

```
axiom muldiv_nostall
=== running
    and (rd_annul^
         or not group(next(run^);alu_ireg^) = hilo_instr)
    => until( next(run^);time,
              not muldiv_stall).
```

8.3 ALU Stage

The ALU stage is the main stage where the axioms are instruction-dependent, since instruction decoding is completed only at the end of the preceding RD stage. The instruction-dependent axioms are given in Section 8.3.1. The following axioms describe the aspects of the ALU stage that are instruction-independent.

```
axiom next_mem_pc
=== running => next(run^);mem_pc^ = alu_pc^ .

axiom next_mem_ireg
=== running => next(run^);mem_ireg^ = alu_ireg^ .

axiom next_mem_in_bds
```

```
=== running => next(run^);mem_in_bds^ = alu_in_bds^ .

axiom next_mem_in_lds
=== running => next(run^);mem_in_lds^ = alu_in_lds^ .
```

The next few definitions define functions and predicates for reading and writing the various *bypass* registers used in the ALU stage. Together with the definition of `mem_put` and `mem_noresult` in Section 8.4 and the axioms of the WB stage, these definitions encapsulate all bypassing, and the special nature of reads and writes to register zero. Instructions that produce a result during the ALU stage (*e.g.*, arithmetic instructions) are described using the `alu_put` predicate, whereas instructions that produce a result during the MEM stage (*e.g.*, load instructions and moves from co-processors) are described using the `alu_delayed_put` predicate. The latter predicate sets the target register and ensures that the following instruction cannot usefully read or write to the same register. Note that the usual result of the `alu_get` function is a 32 bit unsigned number (which can be converted to a signed integer using the `int_val` function if necessary).

```
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)^
    )))).

function alu_put(Reg,Val)
=== (not (alu_in_lds^ and Reg = mem_dest_reg^)
    => (next(run^);mem_dest_reg^ = Reg
        and next(run^);mem_dest_val^ = Val
        and not next(run^);alu_in_lds^)).

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

constant alu_noresult
=== next(run^);mem_dest_reg^ = 0.
```

These predicates and functions are used by the instruction-dependent ALU axioms in Section 8.3.1. The ALU stage axiom of every instruction is required to access general purpose registers only via the `alu_get` function and to assert exactly one of the `alu_put`, `alu_delayed_put` or `alu_noresult` predicates.

For branching instructions, it is the ALU stage that determines whether or not a branch is to be taken. The following two predicates define the effect of each possibility.

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

function branch(Cond,Dest)
=== next(run^);rd_pc^
        = if(Cond and not alu_annul^,
              Dest,
              twos_comp(rd_pc^ + 4)).
```

The ALU stage of every branch or jump instruction should use the `branch` predicate to set the target address of the second-to-next instruction (*i.e.*, the instruction following the branch delay slot). If the `Cond` argument to the `branch` predicate is false, the branch will not be taken and the instruction counter will be incremented as usual. All non-branching instructions should assert `no_branch` during their ALU stage, so that the program counter is incremented.

During the ALU stage, load and store instructions calculate an effective address, translate that address into a physical address and set up a read or write to occur during the following MEM stage. The following two predicates define how this is done. They are used by the instruction-dependent axioms in Section 8.3.1. Note that the read or write is cancelled if the following MEM stage is to be annulled.

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

function setup_write(Addr,Data,Size)
=== (letabs x_data  Data within
    (letabs x_size  Size within
    (letabs x_paddr  physical_addr(Addr) within
    (letabs x_cacheable  cacheable(Addr) within
     next(run^);
        (if(not x_cacheable or mem_annul^,
                dcache_inactive,
                dcache_write(x_paddr,x_data))
         and if(mem_annul^,
                ext_inactive,
                ext_write(x_paddr,x_data,x_size))
        )
    )))).

constant no_readwrite
=== next(run^);(dcache_inactive and ext_inactive).
```

8.3.1 Instruction Specific ALU Axioms

The axioms in this section define the specific events that each instruction or group of instructions causes in the ALU stage.

```
axiom comp_instr_alu
=== running
    and not alu_annul^
    and group(alu_ireg^) = comp_instr
    =>  no_branch
        and no_readwrite
        and not next(run^);alu_in_lds^ .

axiom branch_instr_alu
=== running
    and not alu_annul^
    and group(alu_ireg^) = branch_instr
    =>  no_readwrite
        and not next(run^);alu_in_lds^
        and next(run^);alu_in_bds^ .

axiom store_instr_alu
=== running
    and not alu_annul^
    and group(alu_ireg^) = store_instr
    =>  no_branch
        and alu_noresult
        and not next(run^);alu_in_lds^ .

axiom load_instr_alu
=== running
    and not alu_annul^
    and group(alu_ireg^) = load_instr
    =>  no_branch
        and next(run^);alu_in_lds^ .
```

The ADD instruction adds the contents of two general purpose registers, `Rs` and `Rt`, and places the result in register `Rd`. An overflow exception is raised if the addition overflows.

```
axiom add_instr_alu
=== running
    and not alu_annul^
    and alu_ireg^ = add_instr(Rd,Rs,Rt)
    => (letabs x  int_val(alu_get(Rs))
                + int_val(alu_get(Rt)) within
        alu_put(Rd, twos_comp(x))
        and next(run^);mem_error^
            = if(x : twos_comp_range(32),
                no_error,
                error_overflow)).
```

The ADDI (Add Immediate) instruction adds a sign-extended 16-bit immediate value to the contents of general register `Rs` and places the result in register `Rd`. An overflow exception is raised if the addition overflows.

```
axiom addi_instr_alu
=== running
    and not alu_annul^
    and alu_ireg^ = addi_instr(Rt,Rs,Immed)
    => (letabs x  int_val(alu_get(Rs))
                + int_val_n(Immed,16) within
        alu_put(Rt, twos_comp(x))
        and next(run^);mem_error^
            = if(x : twos_comp_range(32),
                no_error,
                error_overflow)).
```

The ADDIU and ADDU instructions are similar to the ADDI and ADD instructions, except that no checking for overflow is performed.

```
axiom addiu_instr_alu
=== running
    and not alu_annul^
    and alu_ireg^ = addiu_instr(Rt,Rs,Immed)
    =>  alu_put(Rt, twos_comp(alu_get(Rs)
                             + int_val_n(Immed,16)))
        and next(run^);mem_error^ = no_error.

axiom addu_instr_alu
=== running
    and not alu_annul^
    and alu_ireg^ = addu_instr(Rd,Rs,Rt)
    =>  alu_put(Rd, twos_comp(alu_get(Rs) + alu_get(Rt)))
        and next(run^);mem_error^ = no_error.
```

The AND and ANDI instructions perform a logical bitwise *and* operation upon their two arguments.

```
axiom and_instr_alu
=== running
    and not alu_annul^
    and alu_ireg^ = and_instr(Rd,Rs,Rt)
    =>  alu_put(Rd, alu_get(Rs) bitand alu_get(Rt))
        and next(run^);mem_error^ = no_error.

axiom andi_instr_alu
=== running
    and not alu_annul^
    and alu_ireg^ = andi_instr(Rt,Rs,Immed)
    =>  alu_put(Rt, alu_get(Rs) bitand Immed)
        and next(run^);mem_error^ = no_error.
```

The branch instructions all branch relative to the address of their branch delay slot. We specify only the BEQ (Branch if Equal) and BNE (Branch if not Equal) instructions, but others could easily be added.

```
axiom beq_instr_alu
=== running
    and not alu_annul^
    and alu_ireg^ = beq_instr(Rs,Rt,Offset)
    and not alu_in_bds^
    => (letabs x  twos_comp(alu_pc^ + int_val_n(Offset,16)*4)
        within
        alu_noresult
        and branch(alu_get(Rs)=alu_get(Rt), x)).

axiom bne_instr_alu
=== running
    and not alu_annul^
    and alu_ireg^ = bne_instr(Rs,Rt,Offset)
    and not alu_in_bds^
    => (letabs x  twos_comp(alu_pc^ + int_val_n(Offset,16)*4)
        within
        alu_noresult
        and branch(alu_get(Rs)\=alu_get(Rt), x)).
```

The J (Jump) instruction causes an unconditional jump to a destination address that is formed by shifting the 26-bit `Target` field of the instruction left two bits and combining it with the high-order 4 bits of the delay slot address.

```
axiom j_instr_alu
=== running
    and not alu_annul^
    and alu_ireg^ = j_instr(Target)
    and not alu_in_bds^
    => (letabs x  twos_comp(rd_pc^ bitand (15 << 28) + Target*4)
        within
        alu_noresult
        and branch(true,x)).
```

The JAL (Jump and Link) instruction is similar to the jump instruction, but also puts the return address into register 31.

```
axiom jal_instr_alu
=== running
    and not alu_annul^
    and alu_ireg^ = jal_instr(Target)
    and not alu_in_bds^
    => (letabs x  twos_comp(rd_pc^ bitand (15 << 28) + Target*4)
        within
        alu_put(31, twos_comp(x + 8))
        and branch(true,x)).
```

The JALR (Jump and Link Register) and JR (Jump Register) instructions get the jump target from a register. In the presence of interrupts branch instructions can sometimes be executed more than once. Thus the MIPS architecture requires the destination and source registers of the JALR instruction to be distinct.

```
axiom jalr_instr_alu
=== running
    and not alu_annul^
    and alu_ireg^ = jalr_instr(Rd,Rs)
    and Rs \= Rd
    and not alu_in_bds^
    =>  alu_put(Rd, twos_comp(alu_pc^ + 8))
        and branch(true,alu_get(Rs)).

axiom jr_instr_alu
=== running
    and not alu_annul^
    and alu_ireg^ = jr_instr(Rs)
    and not alu_in_bds^
    =>  alu_noresult
        and branch(true,alu_get(Rs)).
```

The LUI (Load Upper Immediate) instruction loads a 16-bit value into the upper half of a given register.

```
axiom lui_instr_alu
=== running
    and not alu_annul^
    and alu_ireg^ = lui_instr(Rt,Immed)
    =>  alu_put(Rt, Immed << 16).
```

The ALU stage of the LW (Load Word) instruction uses `setup_read` to perform address translation and initiate 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.

axiom lw_instr_alu_error
=== 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 not (is_word_addr(Addr)
             and (next(run^);user_mode => Addr < 2**31))
    =>  next(run^);mem_error^ = error_addr_load
        and no_readwrite.
```

The MFC0 instruction allows a value to be moved from a CP0 register into a general purpose CPU register. Like a load instruction, the value is not available until the MEM stage (though it is read from the CP0 register during the ALU stage), so we use `alu_delayed_put` to prevent the following instruction from accessing the target register of the MFC0 instruction.

```
axiom mfc0_instr_alu
=== running
    and not int_pending
    and not alu_annul^
    and alu_ireg^ = mfc0_instr(Rt,Rd)
    and next(run^);(user_mode => cp0_usable)
    =>  alu_delayed_put(Rd)
        and no_branch
        and no_readwrite.
```

The OR and ORI instructions perform bitwise logical OR on their arguments.

```
axiom or_instr_alu
=== running
    and not alu_annul^
    and alu_ireg^ = or_instr(Rd,Rs,Rt)
    =>  alu_put(Rd, alu_get(Rs) bitor alu_get(Rt)).
```

```
axiom ori_instr_alu
=== running
    and not alu_annul^
    and alu_ireg^ = ori_instr(Rt,Rs,Immed)
    =>  alu_put(Rt, alu_get(Rs) bitor Immed).
```

The RFE instruction pops the three level stack of interrupt enable and user/kernel mode status bits. It is typically used within the delay slot of a jump instruction when returning from an interrupt handling routine.

```
axiom rfe_instr_alu
=== running
    and not int_pending
    and not alu_annul^
    and alu_ireg^ = rfe_instr
    and alu_in_bds^
    and next(run^);(user_mode => cp0_usable)
    =>  alu_noresult
        and no_branch
        and no_readwrite.
```

The SLL (Shift Left Logical) instruction shifts the contents of a register left by a fixed number of bits.

```
axiom sll_instr_alu
=== running
    and not alu_annul^
    and alu_ireg^ = sll_instr(Rd,Rt,Shamt)
    =>  alu_put(Rd, twos_comp(alu_get(Rs) << Shamt)).
```

The SW (Store Word) instruction stores an entire word into the data cache and/or the external bus.

```
axiom sw_instr_alu
=== running
    and not alu_annul^
    and alu_ireg^ = sw_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)
    =>  next(run^);mem_error^ = no_error
```

```
        and setup_write(Addr, alu_get(Rt), 4).

axiom sw_instr_alu_error
=== running
    and not alu_annul^
    and alu_ireg^ = sw_instr(Rt,Offset,Base)
    and Addr = twos_comp(alu_get(Base) + int_val_n(Offset,16))
    and not (is_word_addr(Addr)
             and (next(run^);user_mode => Addr < 2**31))
    =>  next(run^);mem_error^ = error_addr_store
        and no_readwrite.
```

8.4 MEM Stage

Apart from load and store instructions, which access the data cache during the MEM stage, most instructions do not cause any activity in the MEM stage.

```
axiom next_wb_pc
=== running => next(run^);wb_pc^ = mem_pc^ .

axiom next_wb_in_bds
=== running => next(run^);wb_in_bds^ = mem_in_bds^ .
```

Instructions that produce a result in the MEM stage pass that result to the following `mem_put` predicate, whereas all other instructions assert `mem_noresult`. Note that the `mem_noresult` predicate moves the bypass registers along the pipeline, whereas the `mem_put` predicate overwrites the bypass result register with its argument value.

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

constant mem_noresult
=== (next(run^);wb_dest_reg^ = mem_dest_reg^
    and next(run^);wb_dest_val^ = mem_dest_val^).
```

The only error that can be generated during the MEM stage is a bus error. The following axioms define several sufficient conditions for this error not to occur.

```
axiom mem_no_error
=== running
    and not mem_annul^
    and dcache_inactive
    =>  next(run^);wb_error^ = no_error.

axiom mem_no_error2
=== running
    and not mem_annul^
    and group(mem_ireg^) = load_instr
    and dcache_readhit
    =>  next(run^);wb_error^ = no_error.

axiom mem_no_error3
=== running
    and not mem_annul^
    and (group(mem_ireg^) = load_instr
        or group(mem_ireg^) = store_instr)
    and until(next(run^);time, not e_bus_error^)
    =>  next(run^);wb_error^ = no_error.
```

8.4.1 Data Cache Misses

The axioms in this section specify when the `d_miss_stall` predicate is true and how the data cache and external memory bus behave when it is true.

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

axiom d_miss_stall
=== running
    and not mem_annul^
    and group(mem_ireg^) = load_instr
    and dcache_read(PhyAddr,Cacheable)
    and is_abs(PhyAddr)
    and is_abs(Cacheable)
    and not dcache_readhit
    and Fixup = next_time(not e_readbusy^) + 1
    and until(Fixup+1, not reset^ and not e_bus_error^)
    and at(Fixup);(not e_readbusy^)  /* avoid retries */
    =>  during(time+1, Fixup,
                d_miss_stall
                and ext_read(PhyAddr,4)
                and dcache_inactive)
        and at(Fixup);
               (d_miss_stall
                and ext_inactive
                and if(Cacheable,
                      dcache_write(PhyAddr,e_data^),
                      dcache_inactive))
        and during(Fixup+1, next(run^);time,
                not d_miss_stall).
```

8.4.2 Write Busy and Partial Word Stalls

The axioms in this section specify how external memory bus writes and write busy stalls are handled. Writes of partial words are handled differently and are not fully specified here, since the instructions that cause them may not be required in our applications.

```
axiom write_nostall1
=== running
    and (mem_annul^
         or group(mem_ireg^) \= store_instr)
    =>  until(next(run^);time,
              not e_writebusy_stall
              and not partword_stall).

axiom write_nostall2
=== running
    and not mem_annul^
    and ext_write(Addr,Data,4)
    and not e_writebusy^
    =>  until(next(run^);time,
               not e_writebusy_stall
               and not partword_stall).
```

```
axiom write_stall
=== running
    and not mem_annul^
    and ext_write(Addr,Data,4)
    and is_abs(Addr)
    and is_abs(Data)
    and e_writebusy^
    and Fixup = next_time(not e_writebusy^) + 1
    and until(Fixup+1, not reset^ and not e_bus_error^)
    and at(Fixup);(not e_writebusy^)  /* avoid retries */
    =>  during(time+1, Fixup,
                e_writebusy_stall
                and not partword_stall
                and ext_inactive
                and dcache_inactive)
            and at(Fixup);
               (e_writebusy_stall
                and not partword_stall
                and ext_write(Addr,Data,4)
                and dcache_inactive)
            and during(Fixup+1, next(run^);time,
                not e_writebusy_stall
                and not partword_stall).
```

8.4.3 Instruction Specific MEM Axioms

The following non-load instructions do not produce results in the MEM stage.

```
axiom mem_noresult
=== running
    and not mem_annul^
    and (group(mem_ireg^) = store_instr
        or group(mem_ireg^) = comp_instr
        or group(mem_ireg^) = hilo_instr
        or group(mem_ireg^) = branch_instr
        )
    => mem_noresult.
```

The next two axioms define the behaviour of a LW (Load Word) instruction for the two possible cases: when a data cache hit occurs and when a miss occurs. The `e_read` and `e_data` locations are defined in Appendix A.

```
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^).

axiom lw_instr_mem_miss
=== running
    and not mem_annul^
    and mem_ireg^ = lw_instr(Rt,Offset,Base)
    and not dcache_readhit
    and until(next(run^);time, not e_bus_error^)
    =>  mem_put(next(d_miss_stall and not e_read^);e_data^).
```

The following axioms specify how the CP0 registers are updated during the MEM stage and the effects of various CP0-related instructions.

```
axiom non_cp0_instr_mem
=== running
    and not mem_annul^
    and group(mem_ireg^) \= cp0_instr
    =>  cp0_registers_invar.


axiom mfc0_instr_mem
=== running and prev(run^);running
    and not mem_annul^
    and mem_ireg^ = mfc0_instr(Rt,Rd)
    and is_abs(Rt)
    and (user_mode => cp0_usable)
    =>  mem_put(prev(run^);cp0_reg(Rt)^)
        and cp0_registers_invar.

axiom rfe_instr_mem
=== running
    and not mem_annul^
    and mem_in_bds^
    and mem_ireg^ = rfe_instr
    and (user_mode => cp0_usable)
    =>  mem_noresult
        and next(run^);(cp0_status^ >> 6) = (cp0_status^ >> 6)
        and next(run^);ints_enabled       = ints_enabled_prev
        and next(run^);ints_enabled_prev  = ints_enabled_old
        and next(run^);ints_enabled_old   = ints_enabled_old
        and next(run^);user_mode          = user_mode_prev
        and next(run^);user_mode_prev     = user_mode_old
        and next(run^);user_mode_old      = user_mode_old
        and next(run^);cp0_cause_code     = cp0_cause_code
        and next(run^);cp0_epc^           = cp0_epc.
```

8.5 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.

```
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)^)
       ).

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)^
       ).
```

8.6 CPU Control Logic

In this section the overall control logic of the CPU is specified. This control logic uses the `*_error` outputs of each pipeline stage, plus various global conditions such as the `reset` and interrupt inputs, to determine whether or not the normal flow of instructions through the pipeline should be interrupted, and whether or not each pipeline stage should be annulled.

```
constant min_reset_cycles
=== 6.

axiom reset
=== hw
    and during(time - min_reset_cycles, time, reset^)
    and until(time+2, not reset^)
    =>  during(time - min_reset_cycles, time+2,
            run^
            and icache_inactive
            and dcache_inactive
            and ext_inactive
            )
        and at(time+2);
           (run^
            and not if_annul^
            and rd_annul^
            and alu_annul^
            and mem_annul^
            and wb_annul^
            and rd_error^ = no_error
            and alu_error^ = no_error
            and mem_error^ = no_error
            and wb_error^ = no_error
            and next(run^);rd_pc^ = reset_excep_addr
            and cp0_status_bev
            and kernel_mode
            and not ints_enabled
           ).
```

The following axioms specify what happens when an interrupt is taken. The interrupted instruction is the one that has just completed the ALU stage and is entering the MEM stage. The interrupted instruction and the following two instructions are annulled. However, the instruction entering the WB stage is completed.

```
function start_interrupt(Cause,PC,InBDS)
=== next(run^);cp0_epc^ = twos_comp(PC - if(InBDS, 4, 0))
    and next(run^);rd_pc^ = gen_excep_addr
    and next(run^);kernel_mode
    and next(run^);(not ints_enabled)
    and next(run^);ints_enabled_prev  = ints_enabled
    and next(run^);ints_enabled_old   = ints_enabled_prev
    and next(run^);user_mode_prev     = user_mode
    and next(run^);user_mode_old      = user_mode_prev
    and next(run^);(cp0_status^ >> 6) = (cp0_status^ >> 6)
    and next(run^);cp0_cause_code     = Cause.


constant mem_interrupt
=== wb_error^ : internal_errors.

function alu_interrupt(Cause)
=== not mem_interrupt
    and not prev(run^);mem_annul^
    and if(not is_word_addr(mem_pc^),
             Cause = error_addr_load,
        if(user_mode and 2**31 =< mem_pc^,
             Cause = error_addr_load,
        if(group(mem_ireg^) = cp0_instr and not cp0_usable,
             Cause = error_coproc_unusable,
        if(mem_error^ : internal_errors,
             Cause = mem_error^,
        if(int_pending,
             Cause = error_interrupt,
        /* else */
             false))))).

function no_interrupt
=== not mem_interrupt
    and not (ex x alu_interrupt(x)).


axiom mem_interrupt
=== running
```

```
    and mem_interrupt
    =>  start_interrupt(wb_error^, wb_pc^, wb_in_bds^)
        and wb_annul^
        and mem_annul^
        and alu_annul^
        and rd_annul^
        and next(run^);wb_error^  = no_error
        and next(run^);mem_error^ = no_error
        and next(run^);alu_error^ = no_error.

axiom alu_interrupt
=== running and prev(run^);running
    and alu_interrupt(Cause)
    =>  start_interrupt(Cause, mem_pc^, mem_in_bds^)
        and wb_annul^ = prev(run^);mem_annul
        and mem_annul^
        and alu_annul^
        and rd_annul^
        and next(run^);wb_error^  = no_error
        and next(run^);mem_error^ = no_error
        and next(run^);alu_error^ = no_error.
```

The remaining axioms in this section define the normal behaviour of the pipeline when no interrupts are taken. Previous axioms have already specified that the `*_pc` and `*_in_bds` values of each pipeline stage are always copied along the pipeline. However, the following axioms show that the `*_annul` and `*_error` values are only copied along the pipeline under certain conditions.

```
axiom next_wb_annul
=== prev(run^);running
    and no_interrupt
    =>  wb_annul^ = prev(run^);mem_annul^ .

axiom next_mem_annul
=== prev(run^);running
    and no_interrupt
    =>  mem_annul^ = prev(run^);alu_annul^ .

axiom next_alu_annul
=== prev(run^);running
    and no_interrupt
    =>  alu_annul^
```

```
        = (alu_error^ : internal_errors
           or prev(run^);rd_annul^).

axiom next_rd_annul
=== prev(run^);running
    and no_interrupt
    =>  rd_annul^
        = (alu_error^ : internal_errors
           or rd_error^ : internal_errors
           or prev(run^);if_annul^).

axiom next_if_annul
=== running
    and not if_annul^
    => next(run^);(not if_annul^).

axiom next_wb_error
=== running
    and no_interrupt
    and mem_annul^
    =>  next(run^);wb_error^ = mem_error^ .

axiom next_mem_error
=== running
    and no_interrupt
    and alu_annul^
    =>  next(run^);mem_error^ = alu_error^ .

axiom next_alu_error
=== running
    and no_interrupt
    and rd_annul^
    =>  next(run^);alu_error^ = rd_error^ .


axiom annulled_mem
=== running
    and no_interrupt
    and mem_annul^
    =>  mem_noresult
        and cp0_registers_invar.
```

```
axiom annulled_alu
=== running
    and alu_annul^
    =>  alu_noresult
        and no_branch
        and no_readwrite.
```

9 Instruction and Data Caches

Here we specify simple instruction and data caches.

The `iloc` and `dloc` functions define the mappings from physical addresses (as on the AdrLo bus) to cache locations. Their precise definition depends upon the size and structure of the caches in a given system. For example, if they are one-to-one functions, then the cache size will be the maximum possible (*i.e.*, `addrlosize**2` bytes). The following axioms capture the general properties of these functions (see Assumption 4).

```
absolute_function iloc(Paddr:bits(addrlosize)):locs.
absolute_function dloc(Paddr:bits(addrlosize)):locs.

axiom caches_disjoint
=== A:bits(addrlosize)
    and aligned(A,4)
    and B:bits(addrlosize)
    and aligned(B,4)
    =>  iloc(A) \= dloc(B).

constant icache_inv
=== (all x is_abs(x)
             and x:bits(addrlosize)
             and aligned(x,4)
    =>  at(time+1);iloc(x)^ = iloc(x)^).

constant dcache_inv
=== (all x is_abs(x)
             and x:bits(addrlosize)
             and aligned(x,4)
    =>  at(time+1);dloc(x)^ = dloc(x)^).
```

```
axiom icache_read
=== hw and i_read^
    =>  icache_read_result(iloc(i_addrlo^)^).

axiom icache_nowrite
=== hw and not i_write^
    =>  icache_inv.

axiom dcache_read
=== hw and d_read^
    =>  dcache_read_result(dloc(d_addrlo^)^).

axiom dcache_write
=== hw and d_write^
    => (letabs x_addr d_addrlo^ within
        at(time+1);dloc(x_addr)^ = triple(d_data^,d_tag^,d_valid^)
        and (all x is_abs(x)
                   and x:bits(addrlosize)
                   and aligned(x,4)
                   and dloc(x) \= dloc(x_addr)
             =>  at(time+1);dloc(x)^ = dloc(x)^)).

axiom dcache_nowrite
=== hw
    and not d_write^
    =>  dcache_inv.
```

10 Write Buffer

As discussed in Section 4.1, to simplify the real-time behaviour of store instructions, we use a simple interface to the external bus that discards writes to cacheable addresses, but stalls writes to uncacheable addresses until the write is completed. The following axioms specify the effect of writing to a cacheable address. The effect of writing to uncacheable addresses is not specified, since our applications do not currently require access to the external bus.

```
axiom e_write_cacheable
=== hw
    and e_write^
    and cacheable(e_addr^)
```

```
=>  until(next_time(e_write^), not e_writebusy^).
```

11 Conclusions

We have specified a detailed model of a typical MIPS R3000 CPU. The specification is more complex than the usual assembler level model of an R3000 CPU, because the pipeline is specified explicitly. However, it was necessary to specify the pipeline in order to get a clear understanding of how the pipelining interacts with interrupts and various kinds of stalls. We have also derived a more abstract instruction level specification from this pipeline specification [Utt93] and have used that specification to verify functional and real-time properties of a small scheduler program [FKU94].

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. Thanks also to John Staples for his detailed comments on the specification.

References

[FKU94] Colin Fidge, Peter Kearney, and Mark Utting. Formal specification and interactive proof of a simple real-time scheduler. Technical Report 94-11, Software Verification Research Centre, Department of Computer Science, University of Queensland, April 1994.

[IDT91] IDT. *1991 RISC Databook*. Integrated Device Technology, Inc, 3236 Scott Boulevard, Santa Clara, California 95054, 1991. Fax: (408) 492-8674.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[KSAL91] Peter Kearney, John Staples, Abdu Abbas, and Chuchang 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, Australia, August 1991. (Revised May 1992).

[LC91] LSI Logic Corporation. *LR3000 and LR3000A MIPS RISC Microprocessor User's Manual.* LSI Logic Corporation, Literature Distribution, M/S D-102, 1551 McCarthy Boulevard, Milpitas, CA 95035. Fax: (408) 433-6802, 1991.

[SRH89] John Staples, Peter J. Robinson, and Daniel 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, Australia, December 1989. (revised September 1991). Accepted for publication in Formal Aspects of Computing.

[Utt93] Mark Utting. Instruction level specification of a MIPS R3000 CPU. Technical Report 93-25, Software Verification Research Centre, Department of Computer Science, University of Queensland, December 1993.

A CPU Interfaces

The main interfaces to a MIPS R3000 CPU are its connections to the instruction and data caches and to the external bus (which allows slower devices to be accessed). Each of these connections is logically composed of address lines, data lines and various control lines. Due to packaging tradeoffs, the address and data lines of these three interfaces are typically multiplexed together, but in this specification they are demultiplexed and represented as distinct locations. This simplifies the specification and makes the Harvard architecture of the CPU explicit. That is, without using separate locations for the instruction and data cache interfaces, it would not be possible to model an access to both instruction and data caches in a single clock cycle.

The following function is used to extract the tag value from a (physical) address.

```
function tagof(Addr) === Addr >> 12.
```

A.1 Data Cache Interface

The data cache interface is defined using the following locations:

```
absolute_constant d_paddr       :locs. /* physical address */
absolute_constant d_cacheable   :locs.
absolute_constant d_addrlo      :locs.
absolute_constant d_data        :locs.
absolute_constant d_tag         :locs.
absolute_constant d_read        :locs.
absolute_constant d_write       :locs.
absolute_constant d_valid       :locs.
```

The above locations should be set only via asserting one of the following predicates. The `addrlosize` constant gives the number of address lines on the AddrLo bus (*e.g.*, 18 for an LSI R3000A with extended cache mode enabled [LC91], or 24 for an IDT79R3001 CPU [IDT91]). It is a parameter to this specification.

```
absolute_constant addrlosize : 16 upto 32.

function dcache_read(PhyAddr,Cacheable)
=== is_word_addr(PhyAddr)
    and d_paddr^       = PhyAddr
    and d_addrlo^      = PhyAddr mod 2**addrlosize
    and d_cacheable^   = Cacheable
    and d_read^
    and not d_write^ .

function dcache_read_result(CacheEntry)
=== d_data^      = triple1(CacheEntry)
    and d_tag^    = triple2(CacheEntry)
    and d_valid^  = triple3(CacheEntry).

constant dcache_readhit
=== d_valid^
    and d_cacheable^
    and d_tag^ = tagof(d_paddr^).

function dcache_write(Paddr,Data)
=== is_word_addr(Paddr)
    and d_addrlo^      = Paddr mod 2**addrlosize
    and d_tag^         = tagof(Paddr)
    and d_data^        = Data
    and d_valid^
    and d_write^
    and not d_read^ .

constant dcache_inactive
=== not d_write^ .   /* we allow reads */
```

A.2 Instruction Cache Interface

The instruction cache interface is almost identical to the data cache interface, except that we do not define the `icache_write` predicate because the specification does not provide any way of writing to the instruction cache.

```
absolute_constant i_paddr       :locs. /* physical address */
absolute_constant i_cacheable   :locs.
absolute_constant i_addrlo      :locs.
absolute_constant i_data        :locs.
absolute_constant i_tag         :locs.
absolute_constant i_read        :locs.
absolute_constant i_write       :locs.
absolute_constant i_valid       :locs.

function icache_read(PhyAddr,Cacheable)
=== is_word_addr(PhyAddr)
    and i_paddr^       = PhyAddr
    and i_addrlo^      = PhyAddr mod 2**addrlosize
    and i_cacheable^   = Cacheable
    and i_read^
    and not i_write^ .

function icache_read_result(CacheEntry)
=== i_data^       = triple1(CacheEntry)
    and i_tag^    = triple2(CacheEntry)
    and i_valid^  = triple3(CacheEntry).

constant icache_readhit
=== (i_valid^
    and i_cacheable^
    and i_tag^ = tagof(i_paddr^)).

constant icache_inactive
=== not i_write^ .   /* we allow reads */
```

A.3 External Bus Interface

The interface to the external bus (or *Memory Bus*) is defined using the following locations:

```
absolute_constant e_addr      :locs.
absolute_constant e_data      :locs.
absolute_constant e_read      :locs.
absolute_constant e_write     :locs.
absolute_constant e_readbusy  :locs.
absolute_constant e_writebusy :locs.
absolute_constant e_bus_error :locs.
```

The `e_readbusy` and `e_writebusy` flags are controlled by the external device and are read only by the CPU. The `e_data` location is set by the CPU during writes, but by the external device during reads. The remaining locations are set only by the CPU and always via one of the following predicates. The `Size` argument should be the number of bytes to be read or written. The specification does not use this size information yet, but would need to do so if we wished to specify load byte or load halfword instructions that access uncached addresses.

```
function is_addr_size(Addr,Size)
=== Addr:bits(32)
    and (Size = 1
        or (Size = 2 and aligned(Addr,2))
        or (Size = 3 and (Addr mod 4) : 0 upto 2)
        or (Size = 4 and aligned(Addr,4))
        ).

function ext_read(Paddr,Size)
=== is_addr_size(Paddr,Size)
    and e_addr^ = Paddr
    and e_read^
    and not e_write^ .

function ext_write(Paddr,Data,Size)
=== is_addr_size(Paddr,Size)
    and e_addr^ = Paddr
    and e_data^ = Data
    and e_write^
    and not e_read^ .

constant ext_inactive
=== not e_read^
    and not e_write^ .
```

B Address Translation

The following axioms define the translation of virtual addresses to physical addresses and which (virtual) addresses are cacheable.

```
function aligned(Addr,N)
=== Addr mod N = 0.

function is_word_addr(Addr)
=== Addr:bits(32)
    and aligned(Addr,4).

function kb(N)  === N * 2**10.

function meg(N) === N * 2**20.

function seg(N) === N * meg(512).

constant kuseg === seg(0) upto seg(4).
constant kseg0 === seg(4) upto seg(5).
constant kseg1 === seg(5) upto seg(6).
constant kseg2 === seg(6) upto seg(8).

function physical_addr(Addr)
=== if(Addr:kuseg,  Addr+seg(2),
    if(Addr:kseg0,  Addr-seg(4),
    if(Addr:kseg1,  Addr-seg(5),
    if(Addr:kseg2,  Addr,
       ?))))
    subject_to is_word_addr(Addr).

function cacheable(Addr)
=== not Addr:kseg1
    subject_to is_word_addr(Addr).
```

C Relationship to the Hardware

The specification that has been presented in this document is an abstraction of the actual behaviour of a typical R3000 system.

The time scale used in the specification is the same as the CPU cycle time, with each new time unit starting on the falling edge of the $\overline{SysOut}$ pin. Note that the $\overline{SysOut}$ clock is the clock to which all devices on the external bus are synchronised.

Table 5 shows the typical relationships between the values of some of the locations used in the specification and the time dependent signals on the corresponding CPU pins. It illustrates how those CPU pins that are time-multiplexed (*e.g.*, the data pins) are represented using multiple locations in the specification.

Similar relationships exist for those specification locations related to the external bus and the `run`, `reset` and `interrupt(i)` locations.

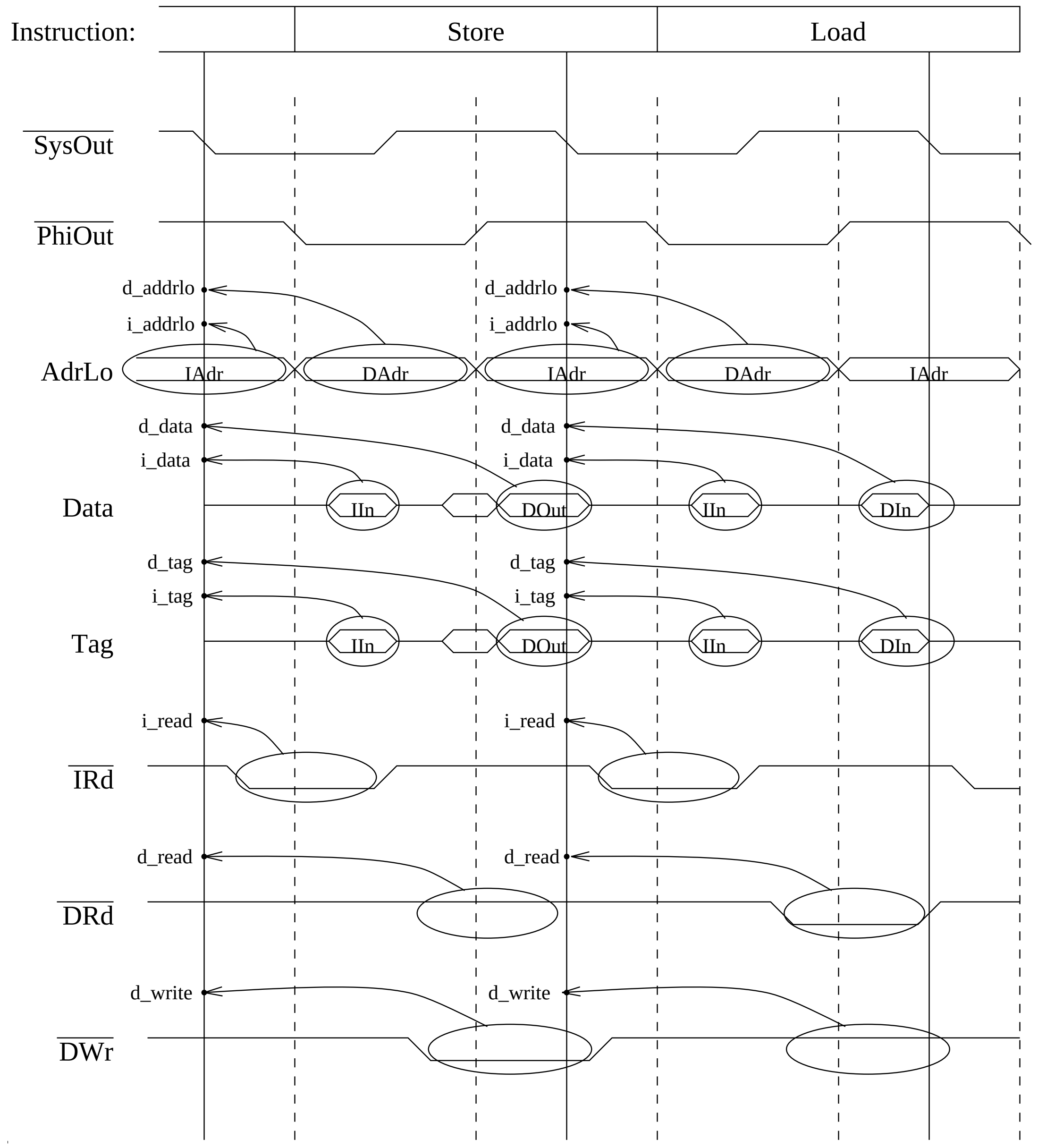


Figure 5: Relationships between specification locations and CPU pin signals for the instruction and data caches. The `d_valid` and `i_valid` locations are not shown, but they have the same relationships to the CPU Tag Valid pin (TagV) as the `d_tag` and `i_tag` locations do to the CPU Tag pins. This diagram is based on the 'Detailed Cache Operation Timing' diagram for the LSI LR3000A CPU [LC91, Page 12-10], but is typical of other R3000 implementations that support external caches.