R4x00
Interface Design Guide

Revision: 1.2
Dated: 1992/06/24

The MIPS 64-bit R4x00 processors can offer remarkable performance. Their system interface is unusual, and the efficiency of the interface and the memory system has an enormous impact on system throughput. But this interface is unusual and the design-orientated documentation is cryptic and hard to follow. This document, written by designers who have built three generations of R4x00 systems, tells you how to keep things simple and succeed.

This manual was originally written by Algorithmics under contract to Siemens GmbH, Semiconductor Division. Following Siemens' decision to pull out of manufacture of R4000 CPUs, the application note was withdrawn. The manual is now made available to you by kind permission of Siemens GmbH.

Algorithmics Ltd
3 Drayton Park
London N5 1NU
ENGLAND.

Phone: +44 71 700 3301
Fax: +44 71 700 3400
Email: dom@algor.co.uk

Contents

Contents 3

1. Introduction to the R4x00 6
1.1. Naming Conventions 7

2. R4x00 System Goals 8
Figure 2.1 Components of an R4x00 system 8

3. Signal Summary 10

4. Clocking and signal timing 13
Figure 4.1 Clock wiring cookbook 13
4.1. Internal clock handling 14
Figure 4.2 Timing relationships between clocks, inputs and outputs 14
4.2. Clock synchronisation targets 14
4.3. R4x00 input setup and hold requirements 15
4.4. Sampling R4x00 output signals 15

5. About the bus 17
5.1. Transaction types used for the R4x00 17
5.2. Signals used 18
Figure 5.1 Write and read cycle (full speed) 18
5.3. Command encoding 18
Table 5.1: SysCmd Codes for uncached read/write requests 19
Table 5.2: Partial word transfer encodings 19
Table 5.3: SysCmd Codes for cached read/write requests 19
Table 5.4: SysCmd Codes for data 19
Table 5.5: SysCmd null code to release system interface 21
5.4. Request flow control 21
Figure 5.2 Bus request flow control timing 21
5.5. Data rate flow control 23
5.6. Data protection: Parity, ECC or none 23

6. Read cycle 24
6.1. How it works 24
Figure 6.1 Structure of a read cycle 24
Figure 6.2 Burst read for cache refill 25
Figure 6.3 Principles of sub-block ordering 26
Table 6.1: Sub-block ordering of data 27
6.2. Reducing read latency 27
Figure 6.4 Low-latency DRAM read - critical signals 27
6.3. Error reporting 27

7. Write cycle 29
7.1. How it works 29
Figure 7.1 Structure of a write cycle 29
7.2. Implementation strategies 30

Data timing 30
Partial word writes 30
7.3. Write errors 30

8. Three R4x00 system recipes 31
Figure 8.1 Where the pins go on a R4x00 system 31
8.1. Tightly coupled single-bank memory system 31
Figure 8.2 System with tightly coupled single bank memory 31
8.2. A simple (bank-interleaved) discrete solution 34
Figure 8.3 Bank interleaving 34
8.3. ASIC-based solution 35
Figure 8.4 Use of an ASIC in the data path 35

9. External Requests 37
Figure 9.1 Using ExtRqst to tristate the bus* 37
9.1. CPU response to external request 38
9.2. Tri-stating the bus without affecting the CPU 39
9.3. Access to the interrupt register 39
Table 9.1: Data encodings for write to interrupt location 40

10. Starting up the R4x00 41
10.1. Configuration Options 41
Table 10.1: Configuration data stream encoding for R4400 42
Table 10.2: System data write pattern encodings 42
10.2. Choosing configuration options 43
Figure 10.1 Wiring IOOut to IIn 45
10.3. Reset/configuration stream timing 46
Figure 10.2 Reset timing of critical signals 46
10.4. Implementation strategies 47

11. A 3.3V CPU in a 5V system 48
11.1. Ground rules 48
11.2. Limiting input signal transitions 48
11.3. Generating 3.3V power from the 5V rail 49
conts
Appendix A: SysCmd codes used by the R4x00 50
Table A.1: SysCmd encodings 50
conts
Appendix B: Alternative configuration stream data sets 51
Figure B.1 Configuration data stream encoding for the R4600 51
conts
Appendix C: Software cache management with DMA 52
conts
Appendix D: Glossary 53
conts
Appendix E: Analogue design considerations 55
High speed CMOS design: edge rates, ground bounce 55
Reflections and other long-track effects 56
Power supply noise and special decoupling 56
conts

Appendix F: Thermal considerations 57

References.....58

Index to signals.....59

1. Introduction to the R4x00

The MIPS R4x00 is a family of processors which has spanned performance levels from about 30-100 times the “SpecMark” unit of a DEC VAX 11/780 minicomputer.

This manual is about those CPUs which retain all or the great majority of the system interface defined for the R4000 components introduced in 1991; they are called R4400PC, R4600 (from IDT, Toshiba and NKK) and Vr4200 (from NEC). Although it doesn’t have much discussion of the “backdoor” secondary cache interface fitted to the high-end members of the family (R4400SC, R4400MC) the system interface is rather similar and much of this material applies to them too.

1995 will see the launch of a couple of R4x00 derivatives with modified or 32-bit buses. This document does not describe these, though you may find it a helpful insight into the strange world of MIPS.

Since the interface spans such a large performance range, it at times puts throughput before simplicity.

An R4x00 processor unit contains:

- *CPU core* : roughly equivalent to any 32-bit MIPS processor plus floating point accelerator. However, the R4x00 can manipulate 64-bit integer data and 64-bit virtual addresses. The hardware interface uses a 64-bit wide data path and its operation is unaffected by whether the programmer/compiler uses the 64-bit extensions to the instruction set.

  The CPU *pipeline* progresses at *twice* the input clock frequency. This means that a processor using a “50MHz” input clock can run instructions at a peak rate of 100M instructions/second.

  R4x00 family members differ in pipeline organisation; the R4400 has an 8-stage pipeline, while the R4600/R4200 are 5-stage. At the same clock rate, the shorter pipeline performs better; the longer pipeline can improve performance only where it makes higher clock rates possible.

  Different suppliers use inconsistent clock-rate suffixes on their parts. Some parts (eg IDT’s R4600) are always described by the pipeline clock rate; but R4400 parts were traditionally described by their *input* clock rate - don’t be deceived, they’re not running at half the speed.

- *On-chip instruction and data caches* : 8-16Kbytes each, depending on CPU type. The data cache is a *write-back* cache; unlike the MIPS R3000, store operations are not immediately forwarded to memory, but the data is held in the cache until the cache storage is either needed for some other data, or is explicitly flushed.

- *Bus interface* : a synchronous, address-data multiplexed, 64-bit wide highway connects the R4x00 to the rest of the system. The bus interface clock frequency may be different from (typically lower than) the input clock frequency; it is obtained by dividing the internal double-speed pipeline clock by 2, 3, 4 or even bigger multiples[1].

  It is this interface which is described in this document.

High-end R4x00 processors (R4400SC, R4400MC) are designed to work with an external *secondary cache;* they contain secondary cache control circuits and an interface allowing the secondary cache to be built up using standard fast SRAM components. This document does not describe these processors. In particular, this document omits all of the facilities provided by the R4400MC to implement a *shared memory multiprocessor* system with *coherent caches.*

However, the bus interface signalling mechanisms *are* the same across the family; so this document may provide a useful introduction to the R4400MC designer.

[1] This feature, now familiar in every PC, was first featured on the R4000.

1.1. Naming Conventions

- *Signals* : the names of signals are written in *SMALL ITALICS*. Buses are referred to with a numeric range; so if there is a bus *SysAD(0:63)* that implies the existence of the 64 signals *SysAD0* through *SysAD63;* but we will usually leave off the numbers when referring to the whole bus and call it *SysAD.*

  Signals which are active low have names which end with a "*" (asterisk).

  There is a signal index so you can see which is referred to where.

- *Conventional terms* : where a word or phrase appears in *italic* you may well find it in the Glossary.

2. R4x00 System Goals

The small-package R4x00 CPUs are intended for “small” single-processor systems where the secondary cache and the big package required to support it are not cost effective. With onchip floating point and considerable processor power, it was conceived largely for desktop workstations and small Unix servers; but most design-ins (to the suppliers’ evident suprise) have been into high-end embedded applications from high-speed networking, through raster image processors, to arcade games.

We expect most R4x00 systems to contain the same basic CPU and memory units, as shown in Figure 2.1:

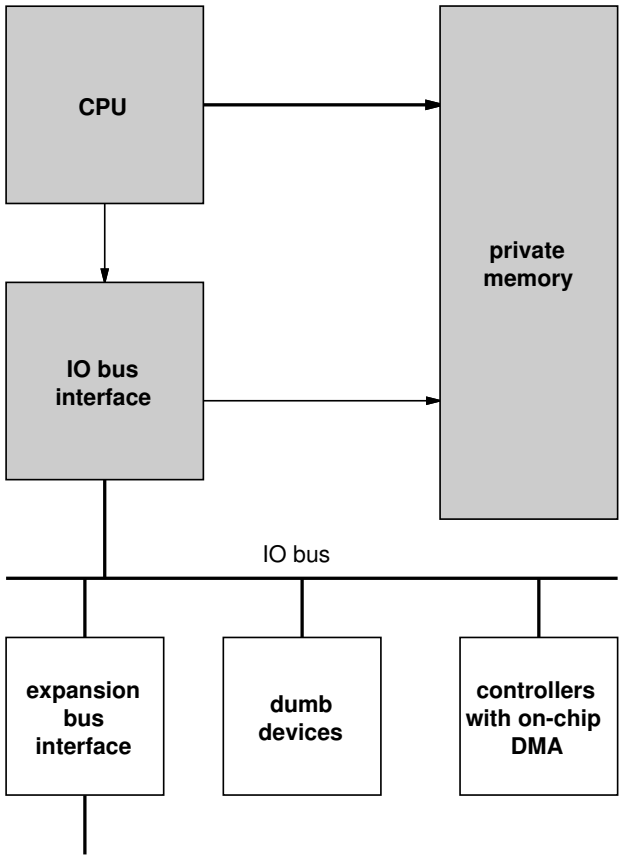

Figure 2.1 Components of an R4x00 system

Notes on Figure 2.1

- *Processor/private memory interface* : to extract the best performance from the R4x00 it must have a low-latency, high-bandwidth connection to a large memory. The relatively small on-chip caches and the very high performance while running cached means that cache misses occur with about the same frequency as the memory references of an uncached CISC processor of 10 years ago; older readers will remember the importance of minimising “wait states” on these CPUs.

  Even in the better R4x00 systems the processor will spend more than half its time waiting for memory transfers.

- *Processor/IO system interface* : IO system transfer rates are far lower and are usually limited by the IO device or controller. The major concern will be to convert the R4x00 bus into something more readily attached to standard peripheral controllers, which is why the example diagram includes an “IO bus”.

- *IO/private memory interface* : local controllers with DMA capability or controllers attached to an expansion bus interface may be given access to the private memory array. Different ways in which this can be done are discussed in §8 below.

The need to take the interface up to very high throughput levels led the R4x00's designers to the following:

- *High and configurable interface clock rate* : ambitious designs can configure a high clock rate for maximum throughput; more economical or conservative systems can configure it lower to simplify interface design.
- *Strictly synchronous design* : every CPU output should be sampled and every CPU input produced by an edge-triggered registered device using a clock derived from the CPU interface clock outputs *TClock* and/or *RClock.*

  The timing parameters were intended to support direct connection of the R4x00 bus interface into a suitable *ASIC;* interfaces using specially-designed chip sets should offer the best performance. But although several chip sets were designed and in some cases offered for the fabulous disappearing "Windows/NT" market, none has yet established a significant following for embedded applications. Perhaps this will change in 1995.

  However, the timing parameters also make it possible to feed CPU outputs into discrete staging registers or (for control signals) a high-speed registered *programmable logic device (PLD)* A range of larger programmable devices are now capable of operating at typical R4x00 interface rates, and allow very efficient control interfaces to be built.
- *Register-to-register design* : to allow the highest possible clock rate CPU inputs are taken directly into on-chip sampling registers, and CPU outputs are resynchronised at the chip border. This allows (nearly) all CPU inputs to require the same, low, setup and hold times; and all outputs to share the same (minimal) clock-to-valid time.

  The problem is that these internal register delays cause a delay before a CPU output can reflect a change of state caused by an input event. The input must be sampled on one clock, internally processed and any consequential change issued on another clock edge; so at least two clock periods will separate the input from the output change.
- *Sophisticated, low-skew clocking* : clock-buffering and chip input/output delays can easily accumulate to steal setup and hold time from a high-frequency synchronous interface. The R4x00 uses phase-locking techniques to generate the bus interface clocks, allowing the user to compensate for buffer delays. This is described in detail in §4.
- *Fire-and-forget approach* : whenever possible the default behaviour of the R4x00 is to make outputs valid for a single clock cycle, and to assume that the external logic will sample those outputs on the rising edge at the end of that cycle. External logic which can't keep up (the usual case) must explicitly throttle the processor; see §5.4.
- *Burst reads and writes* : once the CPU is running, the majority of bus traffic consists of cache refills or write-backs. Cache transfers take place as bursts; one address is provided for a whole cache line of data. Burst transfers can occur as fast as one 64-bit datum per bus clock period.

Other attributes of the bus interface are a consequence of the complex requirements of the R4400MC part; we'll explain those as we come to them.

The net result is an interface which looks intimidating, but is relatively easy to design with once you get used to it.

3. Signal Summary

This section lists off the signals used in the R4x00 interface and discusses their characteristics. Critical or difficult signals are highlighted.

Address/Data transfer signals

SysAD(63:0),SysADC(7:0): (tristate) multiplexed address (36 bits) and data (64 bits), plus check bits. Check bits (not used during address phases) can use either per-byte *even parity* or an *error-correcting code (ECC)* encoding.

SysCmd(8:0),SysCmdP: (tristate) encodes the bus transaction type (see Table A.1 below). *SysCmdP* is an *even parity* check bit.

Note that *SysAD* and *SysCmd* are the *only* CPU outputs which are tri-stated when the CPU releases the bus.

*ValidIn**: (input to CPU) interface logic is presenting a bus operation on *SysCmd* and *SysAD.*

*ValidOut**: (from CPU) the CPU is presenting a bus operation on *SysCmd* and *SysAD.*

*ExtRqst**: (input to CPU) active to request the processor to tristate the bus (see §9).

*Release**: (from CPU) pulsed by the processor when it is tristating the bus for any reason. This happens during any read cycle (in anticipation of someone else supplying the data), but also in response to a tristate request communicated with *ExtRqst*.*

Because you should never allow *SysAD* and *SysCmd* to remain undriven, it is your duty as interface designer to ensure that whenever the CPU asserts *Release*,* your logic takes over and drives the bus to a proper logic level.

*RdRdy**: (input to CPU) when inactive, the processor will stretch any read-address command until *RdRdy** becomes active again (warning - pipelined)

*WrRdy**: (input to CPU) when inactive, the processor will stretch the address part of any write command until *WrRdy** becomes active again (warning - pipelined)

Interrupts

*Int(5:0)**: (input to CPU) any active level which is not masked causes the CPU to take the interrupt trap as soon as execution of the current instruction is complete. Software can obtain the state of these inputs through the cause register trap/error bits.

Interrupt enabling is really a software issue in the RISC R4x00; but a summary here will help readers understand the interrupt options. There are:

- a global interrupt enable bit which is reset automatically on entry to the trap handler, protecting software against premature nested traps; **and**
- a set of six mask bits (in the system status register) corresponding to each interrupt input.

System software manipulates the mask bit to produce the desired interrupt prioritisation and nesting behaviour.

Note that these hardware pins correspond to the bits called “IP7” through to “IP2” in the cause/status register; the lowest two bits provided by the cause/status register are for software use only.

*NMI**: (input to CPU) a falling edge causes a non-maskable interrupt. Since this is not masked by another exception, this can overwrite some registers which save exception state, so software can't guarantee to recover. Think of it as a soft-ish reset.

The NMI software trap is at the reset entry point.

JTAG (diagnostic boundary scan) interface

JTDI, JTDO, JTMS, JTCK: Before you get too excited; the more recent R4600/Vr4200 devices do not implement JTAG boundary scan.

The JTAG interface is to a standard defined by the IEEE, and is intended to support automated testing of assembled boards. In essence, it is a serial bus allowing you to put the chip into a mode whereby its inputs can be sampled and read out serially; and a set of values for all output pins fed in serially and then driven out. It permits the integrity of solder joints to the component (shorts and opens) to be investigated by an ATE machine without needing to understand the detailed operation of the part.

To discover how to use it you need to read [R4000 Processor Interface], and also consult the IEEE JTAG specification.

Bus interface clocks

TClock(1:0): (from CPU) use these to source the clock for your registers which drive CPU inputs.

RClock(1:0): (from CPU) optional bus interface clocks one quarter-cycle advanced relative to *TClock,* can be used to give more hold time for CPU outputs (at the cost of reduced setup time). Probably not useful above 50MHz, since the setup time vanishes before your eyes.

Note that you don't usually drive the bus clocks *TClock* and *RClock* directly from the CPU. In most systems the load which they drive is high and they must be buffered; where we need to show this we'll call them *BTClock* and *BRClock.* Having brought that to your attention, we'll be sloppy. In this manual we will use the names *TClock* and *RClock* to mean the clock you use in your system, whether buffered or not.

SyncOut,SyncIn: buffer *SyncOut* just like you mean to buffer *RClock* and *TClock,* load the output in a way similar to *TClock* and *RClock,* and feed the result back to *SyncIn.*

This will miraculously produce *TClock* and *RClock* signals with the closest possible relationship to CPU outputs and the CPU input sampling point.

Master clock and output drive speed control

MasterClock: (input to CPU) from oscillator (CMOS levels required, so use an FCT buffer).

MasterOut: (from CPU) produced soon after reset (actually, it is guaranteed stable only some time after the assertion of *VCCOk.* You should use *MasterOut* to define the inactivation of the reset signals.

IOOut,IOIn: used for an ingenious self-adaptive signal drive system, introduced with the R4000 but dropped from some later parts. The idea was to connect *IOOut* (from CPU) to *IOIn* (into CPU) via a track of length 50% longer than the longest track among the bus signals *SysAD,* etc, and at the half-way point put a capacitor load equal to the nominal worst-case load of those bus signals.

The CPU will use this to tune its output drive to meet the required clock to output valid time at *IOIn.*

Reset and Initialisation signals (§10)

*ColdReset**: (input to CPU) the basic reset-everything signal. Following power-up, it must be properly asserted before *VCCOk* is allowed to reach a logic 1 or chaos will ensue.

Once you have been through the *VCCOk* sequence you can remove *ColdReset*.* The CPU requires at least 100ms between *VCCOk* assertion and *ColdReset** deassertion, even though the configuration data stream will take much less time to run. The trailing edge of *ColdReset** must be synchronised with *MasterClock* or *MasterOut* (which is allegedly near enough to identical). If you choose *MasterOut,* don't forget that it won't run until quite late in the reset sequence, so you have to use logic which asserts *ColdReset** asynchronously and de-asserts it synchronously.

The bus interface clocks are undefined until *ColdReset** is deasserted.

VCCOk: (input to CPU) a (properly timed) low-to-high transition on *VCCOk* during a period when *ColdReset** is asserted causes many R4x00 CPUs to configure themselves by fetching data using *ModeClock* and *ModeIn.*

This must be done following power-up (allow at least 100ms of good power before *VCCOk* is asserted). You must ensure that *ColdReset** is asserted with or before *VCCOk.* With power already on reconfiguration will occur correctly provided that *VCCOk* is deasserted for a minimum of 64 *MasterClock* cycles.

ModeIn, ModeClock: is a rather heavyweight power-up configuration system inherited from the original R4000, and retained by all devices except for the Vr4200. *ModeIn* (input to CPU) carries the configuration data bitstream, advanced by *falling* edges of *ModeClock* (from the CPU). I meant "advanced"; you must ensure that the first bit in the data stream is presented on *ModeIn* at the point at which *VCCOk* is asserted.

ModeClock will be high from power-on or cold reset, making its first high-to-low transition only after *VCCOk* has been asserted.

ModeClock is quite slow (one every 256 *MasterClock* cycles). Where you have it, *ModeClock* seems to be the best way to time some of the later stages of the reset sequence.

*Reset**: (input to CPU) used to reset the execution unit and reboot the software. *Reset** must be held asserted until the bus interface clocks are properly established, so from powerup must be held throughout the period when *VCCOk* and *ColdReset** are being cycled. A minimum of 64 cycles of *MasterClock* are required for everything to stabilise; then *Reset** should be removed synchronously with *MasterClock* to let the CPU start execution.

*Reset** must be deasserted synchronously with reference to a rising edge of *MasterOut;* it is not clear whether its state very early in the reset sequence matters, but to be safe make sure that it is asserted asynchronously and kept asserted throughout the cold-reset sequence.

Miscellaneous

GrpStall, GrpRun**: appear in early documentation, but are un-defined in later revisions. Connect them to nothing.

*Fault**: (tristate) asserted when CPU is in check mode and the pin state doesn't match. Doesn't work on early R4000 processor revisions nor on later R4600 and Vr4200 types. Only important for dual redundant processor systems, and their problems are beyond the scope of this application note.

VccP, VssP: separate power and ground to feed the phase-locked loop circuitry, which is very sensitive to noise. Should be fed with a specially decoupled supply, carefully tracked; see §11.3.

4. Clocking and signal timing

This section describes the mechanisms which make it possible to run the interface at very high clock rates. This reduces to the provision of clocks to the interface logic which very accurately track the bus interface signals. The clock wiring is shown in Figure 4.1:

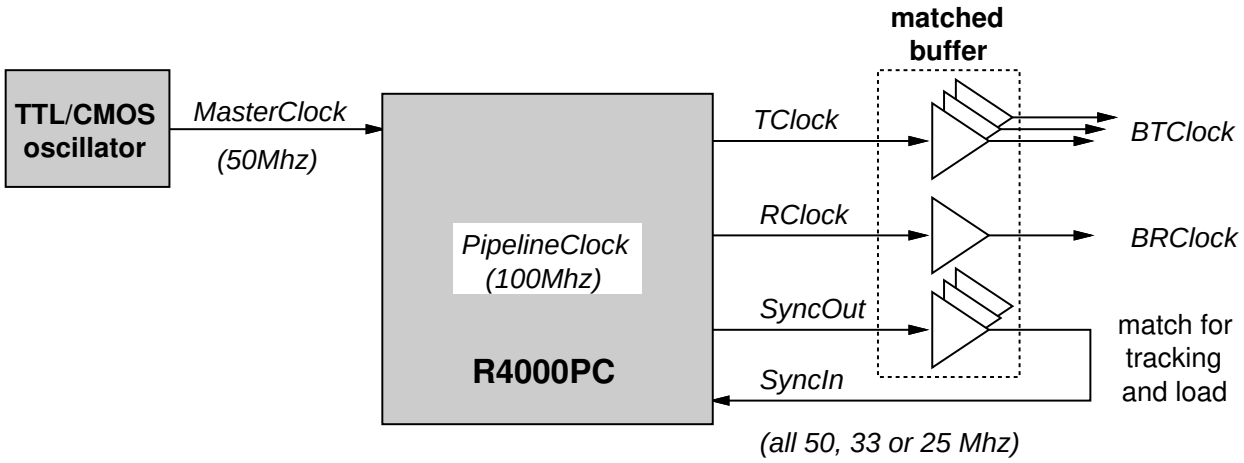

Figure 4.1 Clock wiring cookbook

Note the following:

- the CPU is fed by *MasterClock;* when anyone refers to a “50Mhz” R4x00 they mean the maximum usable *MasterClock* frequency. The numbers on Figure 4.1" are examples, of course.

  Like most complex CMOS devices, the R4x00 appreciates a clock input with a good logic 1 high level. We recommend that you buffer the clock through a high-speed CMOS (“FCT” or similar) gate.

- *MasterClock* is doubled by a *phase locked loop* to produce the main internal pipeline clock, called *PClock* in MIPS documentation. In fact there are faster clocks inside the R4x00, but you don’t need to know about them.

- the pipeline clock is divided by 2, 3, 4 or whatever to define the bus interface clocks. The divisor is selected during the reset-time configuration process, of which more later (§10.1).

- most of your bus interface logic will run from *TClock; RClock* is locked to the same frequency but leads by a quarter of a period, for reasons explained below. A good deal of the processor, interface and memory system will end up running synchronously to *TClock,* so even though the CPU drives two copies you will probably have to buffer it and generate multiple outputs. You should try your best to make sure that all your output clocks are similarly tracked and loaded.

  Inside the chip the bus interface runs from an invisible internal clock called *SClock;* the phase-lock mechanisms allow you to ensure that your *buffered TClock* is tightly aligned to the internal transition points.

- *SyncOut* is provided so that the CPU can compensate for the delays in your buffering scheme. You should buffer *SyncOut* exactly as you propose to buffer *TClock,* load the buffered signal with tracking and capacitance as far as possible identical to your buffered *TClock,* and then feed the result back in to *SyncIn.* The CPU will adjust the phase of *TClock, RClock* and *SyncOut* together until *SyncIn* is placed as close as possible to its internal interface clock (which is used as the reference point for setup and hold times). At this point your output clocks should be tightly aligned too.

4.1. Internal clock handling

It's time for a stroll around the R4x00 clocking mechanisms, and an introduction to the different clocks and what they should be used for. We just fudged the basic question of why there are so many clocks. First of all, what are all the clocks for?

- *MasterClock* determines the frequency of the system. But because the bus interface clocks are defined by a programmable divisor, and that divisor is configured at some point during the reset sequence, the reset sequencing must be done with reference to *MasterClock* (or possibly with *MasterOut,* which is an in-phase copy of it).
- *SyncOut, SyncIn* are just there to be looped back to allow you to compensate for any buffering needed in the real bus interface clocks.
- *TClock* is the main bus interface clock. All CPU output timings and input setup and hold requirements are referred to *TClock.*
- *RClock* runs 25% of the clock period ahead of *TClock* and can be used where external logic clocked with *TClock* would have hold time problems with fast-changing CPU outputs. This is especially likely to be a problem when capturing write data, and when interfacing any CPU output to an ASIC.

Consider the picture Figure 4.2.

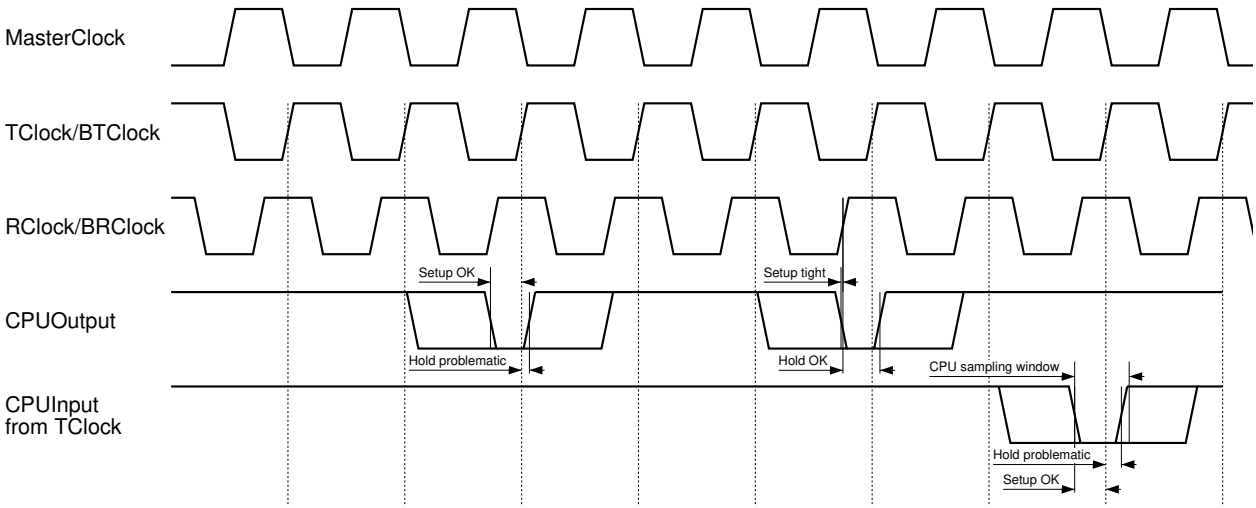

Figure 4.2 Timing relationships between clocks, inputs and outputs

The picture is based on 67Mhz CPU timings with the bus running at a 1:2 divider (bus clocks 67Mhz). The principle difficulty with the R4x00's strict register-to-register logic, for both input and output signals, is that of ensuring hold time with respect to *TClock.* The hold time problems on sampling CPU outputs can be sidestepped by using *RClock,* but this can cause further problems, which are discussed below.

4.2. Clock synchronisation targets

The R4x00 interface depends crucially on the success of the phase-locked loops in aligning output signal changes and input signal sampling points onto the buffered *TClock* signal.

If you look at the R4x00 specifications you will find the signal timings related to an internal clock called *SClock.* This doesn't exist as an interface signal, but the phase-lock arrangements mean that it is guaranteed to track *SyncIn* with a maximum skew specified by the "clock jitter" parameter (typically 0.5ns). The clock jitter degrades the tracking (with reference to *SyncIn)* of both *SClock* and the output clock *TClock* separately, so the effect of an 0.5ns maximum jitter is to worsen setup and hold requirements referenced

to the system-wide *TClock* by the combined effect, or 1ns.

By the way, don't be tempted to actually use *SyncIn* as the incarnation of the invisible *SClock;* it turns out that with some CPUs *SyncIn* may be 180° out of phase with *TClock.*

4.3. R4x00 input setup and hold requirements

You have to ensure that inputs to the R4x00 are stable throughout the *sampling window;* this window is defined by both the setup and hold parameters quoted in the data sheet (3 and 2 respectively at 50Mhz), but degraded by the skew between *TClock* and the internal, invisible *SClock* which determines the sampling time.

- *Setup time* : it is immediately apparent from Figure 4.2 that getting a signal valid in time for the start of the window is a simple matter of ensuring that the clock-to-output performance of the device generating the signal is fast enough for the chosen interface clock rate. This is not a problem; you can use 15ns devices even at 50Mhz.
- *Hold time* : ensuring that the signal remains valid to the end of the window is significantly more difficult. The clock skew generated by the R4x00 PLLs and your clock buffering and distribution system increase the required hold time to typically 3-5ns. Fast discrete registered devices don't usually guarantee any clock-to-output minimum, and if they do it will be less than this.

So let us concentrate on the hold time problem. Possible approaches are:

- *Leave signal valid for (at least) an extra clock* : much the best trick when it is possible. Try to ensure that input changes are not triggered by the same edge on which the CPU samples them. This means that any transition has to occur at some *TClock* rising edge where the CPU does not look at the value of this signal.
- *Use a delayed clock* : clock your register with a delayed derivative of *TClock;* some modern programmable devices have selectable clock delays, or you can put the clock through an additional high-speed buffer. This will work well, but can create a "chain" of hold-time problems back into your control logic.
- *Include another gate after the clocked element* : if you drive an input with a high-speed register followed by an additional high-speed gate, it is still relatively easy to meet the setup requirements; and the additional delay makes the hold time safe. If used for wide paths it will add to the component count, of course.

4.4. Sampling R4x00 output signals

As shown in Figure 4.2, setup time to *TClock* is usually adequate. But take care; the output timings depend on the *output buffer di/dt control mechanism,* whose parameters are set up at reset-time initialisation.

The mechanism affects all CPU outputs except the clocks. The outputs are driven with only the current which is necessary to meet specified timing parameters, allowing systems with lightly-loaded buses and/or carefully timed interface circuitry to reduce peak power surges, electrical noise and ground bounce. Three drive speeds are supported; they produce nominal output clock-to-output maxima of about 50%, 75% or 100% of a *MasterClock* period. See §10.1 for how to select different speeds.

Hold time is a different story, and the guaranteed period for which the outputs remain stable after the clock edge (some say 2ns, some zero, for 50Mhz parts with the fastest drive-time configuration) is insufficient to cover clock skew plus the hold requirements of most receiving devices. There are a number of tricks which can be used:

- *Slow the outputs* : at minimum output drive speed the hold time may become workable. But the venerable MIPS specification [R4000 Processor Interface] did not define minimum clock to output

timings, and the semiconductor vendors have not improved the situation.

- *Registered PLDs* : will usually sample the data OK. In this type of device the clock is hard-wired to the internal registers, but input signals are taken through the programmable logic array before being presented at the register data input. The data is therefore always more delayed than the clock, which makes PLDs resilient to marginal hold time problems; the hold time is usually specified as 0, but it is safe to assume that they can cope with signals which change even a little before the clock (a little is perhaps 10% of their nominal setup time).

  So a registered PLD clocked from one of your *TClock* signals will sample outputs cleanly. Ideally you will select a PLD which is no faster than is necessary to cope with worst-case setup times.

- *Using RClock to sample data*: this will work up to 50MHz (though the setup time is pretty low). The problem is that the resulting outputs will themselves change too early to be fed into subsequent logic stages running from *TClock,* so the problem tends to chain through your system.

  This solution is usually applicable to CPU write data. The best discrete solution seems to be to use a fast register with a clock enable. Clocked from *RClock,* the data hold time is OK; and the clock enable signal can come from the *TClock* universe. Now using the clock enable you can (if required) stretch the data over more than one clock.

- *Add a gate upstream of the register* : can be done, subject to expense. Not a good solution for data (who wants 64 extra buffers?).

- *Persuade the CPU to keep signals valid for more than one clock* : this is particularly applicable to addresses, which can be held using cycle flow control (see §5.4); and particularly inapplicable to write data, which is only defined at one clock edge[2].

See the recipes in §8 for examples of practical solutions.

[2] You may notice that the R4600 CPU, when configured to drive write data “slowly”, holds the data through the idle (“x”) bus cycles. But the *last* word of a data burst is still not extended.

5. About the bus

Since the R4x00 interface is synchronous, the bus protocols can be built on the assumption that anything presented with proper timing will be received correctly by the "other end". MIPS use the phrase *external agent* to describe the other end; we will be more informal and just call it "your interface logic", or something of the kind.

The R4x00 bus interface signals are driven either by the CPU (CPU as master) or by your logic (CPU as slave). Your logic cannot drive the bus until the R4x00 says so.

When the CPU has something to say on the bus, it is accompanied by the assertion of *ValidOut*.* When your interface logic has something to give back to the R4x00, it must assert *ValidIn*.*

Bus operation consists of:

- *Requests* : which consist of a single value asserted by either party for at least one clock period. MIPS documentation calls this a *response* if it is data being returned to the processor after a read request. This distinction seems unhelpful and we won't follow it.
- *Messages* : which consist of a sequence of requests issued by one party. For example, a CPU write consists of a write-address request followed by one or more write-data requests. Within messages the requests can be nose-to-tail, but there may also be gaps. There are some published rules about the size of the gaps but they are complex and you should probably not count on them.

  The R4x00 accepts any valid message from your logic at whatever speed you can throw it[3]. Your logic most probably can't, and a mechanism exists to stop the processor from overrunning; see §5.4.
- *Transactions* : which actually do something. A CPU write transaction is represented by one message on the bus, but a CPU read involves two; the read-address request from the CPU, and the message which returns the data.

  In principle, the bus protocols allow reads to be *split transactions;* that is, lots of writes and strange cache-coherency operations may be interleaved between the read request and the data which is returned. But this can't happen on an R4x00.

There is no general idea of a handshake; handshake protocols take up bus time and the R4x00 interface is conceived to minimise bus occupancy.

5.1. Transaction types used for the R4x00

The bus system is very general, and in the R4400MC (multiprocessor) is used to communicate all sorts of complicated and exciting messages about cache coherency. But the R4x00 is much simpler:

- as master, the R4x00 drives *read requests* (a single cycle presenting a command and an address) and *write requests,* which are multi-cycle operations with an address followed by one or more cycles of data.
- with the R4x00 as slave, your logic drives read data (single cycle or cache-line-sized bursts) back to the R4x00 in response to a read request.
- the only operation which might require an external access to a CPU internal register is to manipulate interrupt bits, and this won't be necessary with the small package variants.
- R4x00 reads should not be split - once the CPU has issued a read request the next thing it sees must be the response data.

[3] Secondary cache versions may not be able to accept cache refill data at the maximum rate, but that's another story

5.2. Signals used

Signals involved in bus transactions include:

- *SysAD* and its check bits *SysADC* provide a 64-bit path to carry data and addresses. The 36-bit addresses use only *SysAD(35:0).* You can safely ignore parity or ECC bits while *SysAD* is carrying addresses.
- the 10-bit *SysCmd* bus encodes the command. Nothing like all 1024 values are used; there are only about 30 valid codes for the R4x00. Most of you will ignore *SysCmdP;* the CPU does when it is receiving.
- *ValidOut** is asserted by the CPU as master when it is driving something on the bus; your logic should assert *ValidIn** when you are driving something on the bus.

In the simplest case bus operations are presented for one clock period, and the other party is then assumed to have collected them. This would make external logic impossibly difficult, so there are mechanisms which can “throttle” the CPU to slow down the rate at which it presents addresses and data; see §5.4 below. For the present, though, lets pretend everyone goes fast enough and has enough buffering capability.

Figure 5.1 shows what happens on the bus for a write cycle followed by a read. Much better pictures and fuller explanations are to be found in §6 and §7:

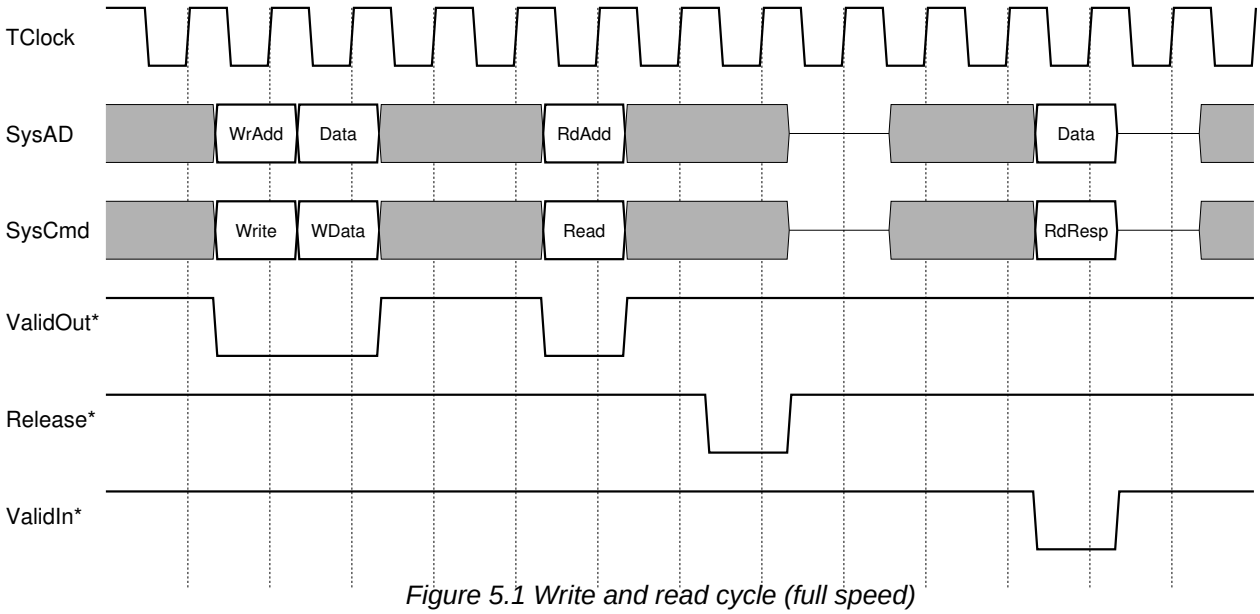

Figure 5.1 Write and read cycle (full speed)

5.3. Command encoding

The command bus *SysCmd* tells the system what to do. The bus is bit-coded, but multiple different meanings are overloaded onto bit-sets depending on context. Appendix A gives a comprehensive list of all values used by the R4x00. The bit-orientated description here is therefore broken down into:

- read or write requests for non-burst access, presented when the appropriate byte address is on *SysAD.*
- cache refill and write-back requests (ie burst reads or burst writes), presented when the appropriate byte (but always 8-byte aligned) address is on *SysAD.*
- data codes, presented when *SysAD* is carrying data to and from the processor.
- the null request code, which should be fed to the CPU to end a period where someone else has been using the bus (see §9).

General principles

- *SysCmd8* is set 1 when *SysAD* is carrying data, and 0 when it is carrying addresses.
- during address-type requests *SysCmd(7:5)* encode a request type - the only ones of interest on the R4x00 are read, write and null.

Non-burst read and write requests

8	7 — 5	4 — 3	2 — 0
0	000=read, 010=write	11	width

Table 5.1: SysCmd Codes for uncached read/write requests

The *width* code determines how many bytes are being transferred across the bus. Note that although regular MIPS instructions can produce only 1-byte, 2-byte, 4-byte and 8-byte transfers (in all cases aligned on a corresponding memory address boundary) special instructions ("load/store left" and "load/store right") are available which can transfer a group of bytes of length 3, 5, 6 or 7 provided that the group either starts or finishes on a 4-byte or 8-byte boundary as applicable: see Table 5.2.

In all cases *SysAD* holds the byte address of the lowest-addressed byte being transferred.

Where *n* bytes are transferred, the width is the binary code for *(n-1)*; so code "000" means a 1-byte transfer, "111" an 8-byte transfer, and so on. Table 5.2 lays out the result; note the way in which the byte lane used depends on the processors "endian-ness" option.

Cache burst read and write requests

8	7 — 5	4 — 3	2	1 — 0
0	000=read 010=write	10	×	00 = 2×64-bit data 01 = 4×64-bit data

Table 5.3: SysCmd Codes for cached read/write requests

Bit 2 encodes information which could be meaningful only for a multiprocessor application.

The burst length corresponds to the cache line size, but it is encoded here because I-cache and D-cache line sizes can be configured differently. In many cases your interface logic will know already what burst size is to be used, so you won't need to decode the field. Note that the cache line sizes are documented as multiples of a 32–bit word: of course only 2 64–bit data are required to refill a 4×32–bit cache line.

For a processor with a secondary cache the "burst length" field could also be "10" for 8 data cycles, or "11" for 16 cycles.

Data codes

8	7	6	5	4	3	2 — 0
1	0=last 1=more coming	0=read 1=write	0=good 1=error	1=no parity check (resvd on write)	0	×

Table 5.4: SysCmd Codes for data

- *Last/more coming* : the *more coming* bit is required on each datum of a burst except for the last one. The CPU's response to a "last" data item sent too early is undefined, so don't feel tempted to try it. A burst read request must elicit the number of data items which the CPU asked for.

width (code)	*Address mod 8*	*SysAD byte lanes used (little-endian)* 0-7	8-15	16-23	24-31	32-39	40-47	48-55	56-63
1 (000)	0	•							
	1		•						
	2			•					
	3				•				
	4					•			
	5						•		
	6							•	
	7								•
2 (001)	0	•	•						
	2			•	•				
	4					•	•		
	6							•	•
3 (010)	0	•	•	•					
	1		•	•	•				
	4					•	•	•	
	5						•	•	•
4 (011)	0	•	•	•	•				
	4					•	•	•	•
5 (100)	0	•	•	•	•	•			
	3				•	•	•	•	•
6 (101)	0	•	•	•	•	•	•		
	2			•	•	•	•	•	•
7 (110)	0	•	•	•	•	•	•	•	
	1		•	•	•	•	•	•	•
8 (111)	0	•	•	•	•	•	•	•	•
		56-63	*48-55*	*40-47*	*32-39*	*24-31*	*16-23*	*8-15*	*0-7*
		SysAD byte lanes used (big-endian)							

Table 5.2: Partial word transfer encodings

You may feel that the burst size is being simultaneously signalled in several different ways. You're quite right, but you have to live with it.

- *Read/Write* : zero if this datum is in response to a read request; one if this datum is following a write request.
- *good/error* : if data is returned with the error bit set, the CPU will take a bus error trap.
- *no parity check* : set this bit to tell the CPU not to check parity on *SysAD, SysADC* for this data. You can of course do this all the time and save yourself a set of parity generators, but in doing so you will make any memory errors (perhaps in your prototypes) hard to diagnose.

Null code to return the system interface to the CPU

There is only actually one value:

8	7 5	4 3	2 0
0	011=null	00	×

Table 5.5: SysCmd null code to release system interface

5.4. Request flow control

If your interface is in danger of being overrun by R4x00 addresses or data, you need this part of the system. At the time the R4x00 utters a request on its bus it looks at the signals *WrRdy** (sampled when the message is a write) or *RdRdy** (used for all other messages). If the appropriate "..Rdy" isn't ready, the CPU will stall the interface driving *ValidOut** and the appropriate command and address bits. When your interface has collected the address (and, on a write, is ready to accept the data) you may assert *WrRdy** and/or *RdRdy** and the CPU will carry on.

The above sounds straightforward, but there is one small catch. As discussed earlier, R4x00 inputs are all registered at the chip interface and all outputs are fed directly from an output register. This means that the CPU takes one clock to sample *WrRdy** or *RdRdy** active, and one more clock to change the bus outputs in response. Figure 5.2 shows what we mean:

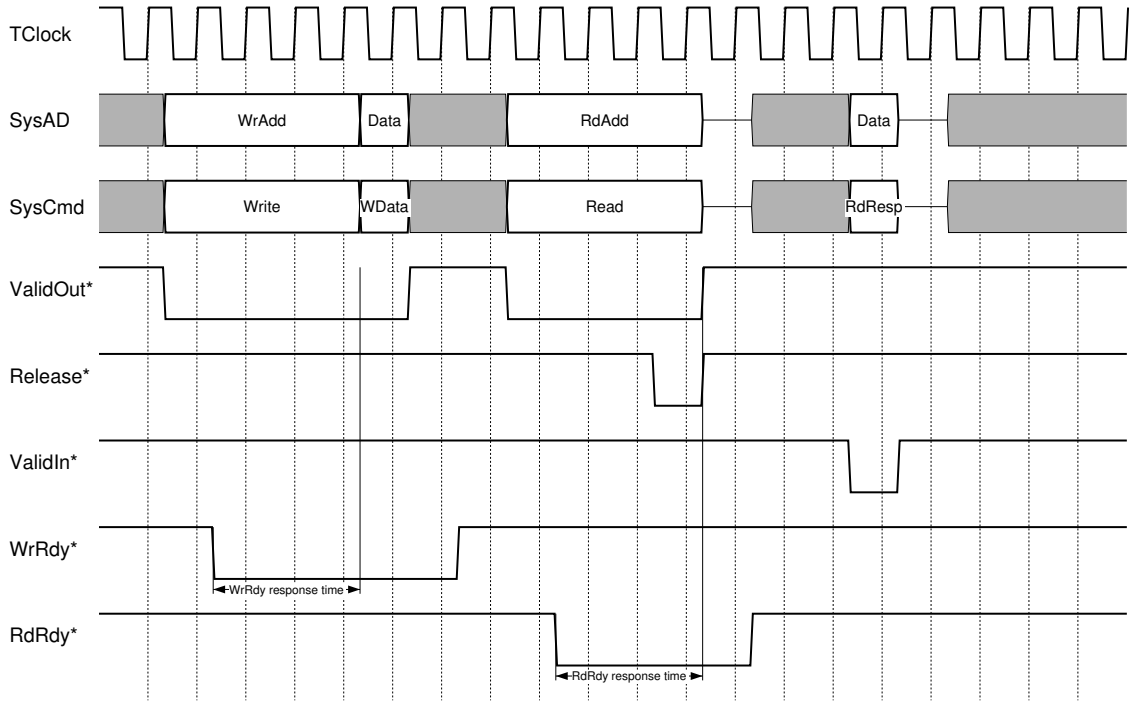

Figure 5.2 Bus request flow control timing

Notes on Figure 5.2

Figure 5.2 shows the use of the signals *RdRdy** and *WrRdy** for flow control. Note that the signals are left inactive by default, because the CPU samples them before you notice a sample has begun. So the signals are left in the deasserted state and asserted only when the CPU drives a request on the bus and the interface logic decides that it can accept the address.

The response of the CPU to *RdRdy** or *WrRdy** is relatively slow, because of the input and output register stages. On the diagram you can see that the immediate assertion of *WrRdy** in response to *ValidOut** still results in the write address phase lasting for four clock periods instead of one.

Note also that once *WrRdy** is asserted no further control can be exercised over the rate at which data is transmitted by the CPU (though the repetition rate for cache writebacks can be reduced, see the next section §5.5).

Take note that the CPU may produce a read request in the clock immediately after the last data cycle of a write. There *is* a guaranteed gap for write→write repetition, which is essential to support designs asserting *WrRdy** during one cycle in order to prevent a subsequent write from issuing. The rule is that there must be at least four clock periods between the issue cycle of one write and a valid address for the next write.

Flow control strategies

There is an optimistic and a pessimistic way of employing flow control; which you adopt will depend upon how much buffering you can provide for CPU addresses and write data.

- *Optimistic* : if you have enough buffering you can leave *RdRdy*, WrRdy** (or both) asserted most of the time, de-asserting them only when your memory or I/O interface buffers get clogged up so that there is no longer storage for the addresses and/or data.

  This may improve read latency, but unless you have a pretty slick local memory system you will find that *RdRdy** is not on the critical path.

  Keeping *WrRdy** asserted will speed up the flow of writes across the R4x00 interface. This is a particularly good thing for multiprocessing applications, where there is much more activity on the CPU interface.

- *Pessimistic* : leave *RdRdy*, WrRdy** (or both) deasserted. When you see a processor request and address your interface can prepare for it, and only when it can guarantee to be ready need it assert the signals.

  This seems much less efficient, but actually may cost very little. Read latency is probably determined by the overall address→data latency of your memory system. Deasserting *RdRdy** does not prevent the CPU from transmitting the address - the assertion of *RdRdy** and the release of the bus by the CPU will still occur before the data could have been returned.

  During writes, the CPU interface is much more efficiently used if *WrRdy** is kept asserted, but writes have to go somewhere. Again, the system performance may be little changed if the write address is kept on the system bus rather than in some downstream register in the memory system.

  Unless you have a data-path chip designed for R4x00 applications you will probably not have the data storage to run with *WrRdy** asserted.

The decision to make, then is:

1) Write timing will depend on how you capture cache write-back data. If, as is likely, write-back without flow control would require extra buffering components it is probably not worth it.

   Moreover, writes only impact CPU performance when they delay a following read. It is therefore not useful to buffer a write, and clear *SysAD,* if the write remains active in the memory system and will block a cache refill anyway.

   We think most designs might as well run with *WrRdy** normally deasserted.

2) Read timing depends on your read latency targets. If you can return local memory data in five clocks or less, then keeping *RdRdy** asserted will save clocks, at the cost of being careful to ensure that you can capture the fast-moving addresses. If your memory latency (to the point where you need to drive first data) is 6 clocks or more you will gain nothing from your efforts.

We think that most R4x00 applications will be simplified by slowing all writes. The majority should probably slow all reads, too, but the brave may, if the memory system is improbably slick, gain a few

percent performance by exploring the alternative.

5.5. Data rate flow control

Data supplied back to the R4x00 CPU during burst read cycles (cache refills) may be presented with whatever timing you like[4]. On any cycle for which *ValidIn** is asserted and accompanied by a valid read data response code on *SysCmd* the CPU will sample data.

On writes, the CPU throws data at you with a pre-arranged timing sequence. You have some control over this sequence by a boot-time configuration variable; in particular you can cause it to transmit its words at regular intervals between 1 and 4 clock periods. You are likely to have two problems:

- *Uncertain first-data timing* : the first word of data may be presented in the clock cycle directly after the address (the case shown in the timing diagrams) but there may be a gap between the two - we've observed as many as 5 idle cycles.

  The specification does not say what will happen; you just have to monitor *ValidOut*.*

  Data which directly follows the address doesn't allow anything for the startup latency of your memory system; delayed data has the problem that in edge-triggered memories such as DRAMs (where it is the activating edge of the column address strobe which determines the data sampling point) the control strobe has to be generated from the CPU's data timing.

  All this might be easier if you can afford a FIFO store which will store the entire write cycle (address and data), but system performance is unlikely to be enhanced and may suffer from increased contention between the CPU's last write and next read.

- *Repetition* : it is possible to build a bank-interleaved DRAM array which can accept a burst of data at the system clock rate, but at high system clock rates it can get very wide. It may be worth presenting one word every two or three cycles in order to use a "narrower" DRAM system. Because all latencies on the bus go in cycle multiples, this is the better than slowing data bursts by configuring a slower interface clock; the slower clock will affect many other latencies.

5.6. Data protection: Parity, ECC or none

The R4x00 always generates parity or ECC bits with commands, addresses and data. It can check parity/ECC on incoming data.

This provides a check on the integrity of the bus interface and other data connections. In small systems it is unlikely to be sensible to implement either scheme just to check out the data path (since the amount of logic being checked may be small compared to the amount of logic added to check it). However, if your DRAM array will be parity checked it makes sense to extend that parity protection "end to end" to and from the CPU. It may also make your memory system design easier, because the DRAM system won't have to generate and check parity "in real time" during CPU burst transfers.

[4] On a R4400SC or R4400MC system, with a secondary cache, the data response timing must not be too fast or the secondary cache interface may be overrun. Since the secondary cache timing is configurable to meet the requirements of your particular static RAMs the relationship is complex. You'll have to read [R4000 Processor Interface] for this one.

6. Read cycle

It is a pity that the major flaw in the regularity and simplicity of the R4x00 interface is that reads, the basic cycle you would like to know about first, are the place where most of the exceptions to the standard bus protocols are seen.

6.1. How it works

Figure 6.1 shows a read cycle slowed down by the use of *RdRdy*.* This is not the simplest case, but most implementations will use it:

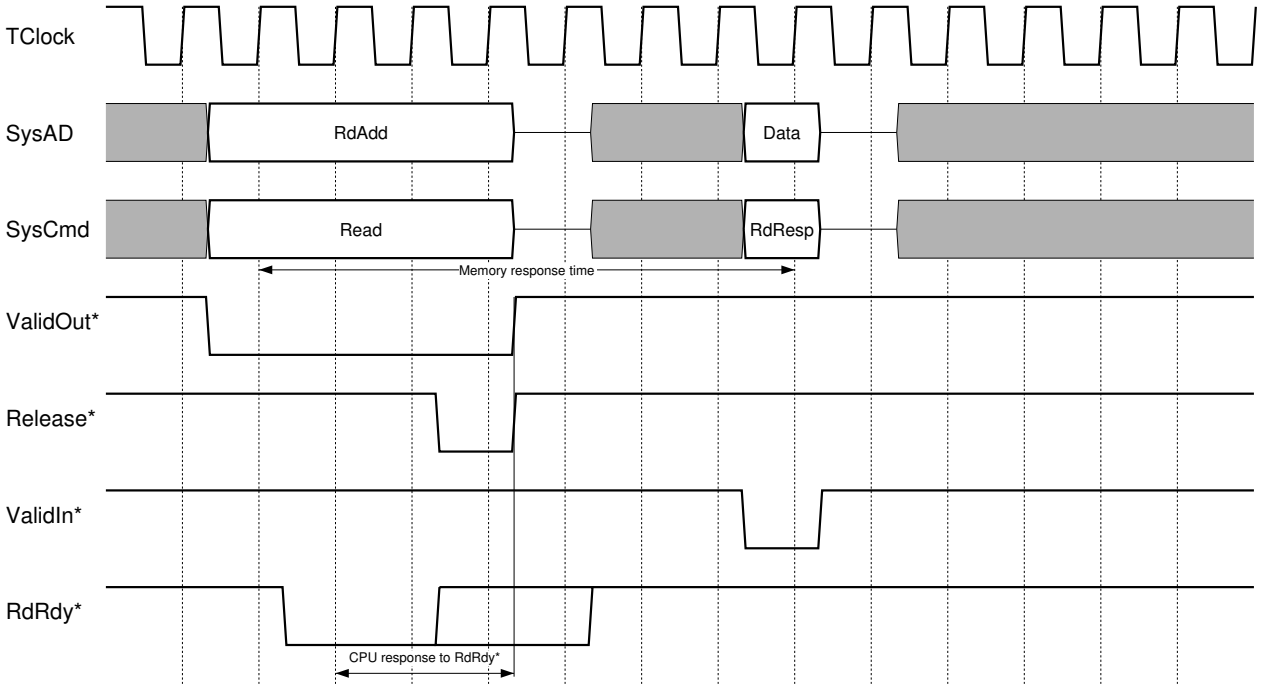

Figure 6.1 Structure of a read cycle

Notes on Figure 6.1

The sequence of events is as follows:

- *CPU request* : recognised by assertion of *ValidOut*;* at the next clock edge *SysAD* and *SysCmd* can be sampled to discover that the CPU wants to perform a read.
- *Address acknowledged* : the interface logic acknowledges the request by asserting *RdRdy*.* Because of the way the CPU samples and pipelines its inputs and outputs, the CPU's deassertion of *ValidOut** will not occur until three clocks after *RdRdy** is asserted (shown on Figure 6.1 as "CPU response to *RdRdy**"). This means that the address is valid for four clock periods, so need not have hold time problems sampling it.

  In principle it is only necessary to drive *RdRdy** for one clock. In practice, once the CPU has seen it the CPU regards the signal as "don't care" until two cycles before the next possible read address phase; so your logic can leave *RdRdy** asserted for at least one extra clock time and avoid a potential hold time problem.
- *Bus release* : the CPU now finishes the read request and will voluntarily float *SysAD* and *SysCmd.* It indicates this by a one-clock pulse on *Release*.* R4x00 CPU's without the secondary cache (which only do simple read requests without split transactions) normally assert *Release** in the read "issue

cycle"; that is, the last cycle in which the CPU presents a read request code and asserts *ValidOut**. But the CPU specification does not guarantee this timing, so you should be prepared for it to assert *Release** (and tri-state the buses) some time later.

It is bad practice to permit the *SysAD* and *SysCmd* buses to float to an invalid level. After one clock bus exchange time you are supposed to drive them with some stable data, though its value does not matter. You will be able to get away with a couple of clock cycles of float if you have nothing except CMOS logic on the bus.

- *Data response* : your logic sets up the data and response code and pulses *ValidIn** (here just for one clock because it was a read of a single word or partial word). The response code defines the data as "last data of response", but don't rely on this; a CPU expecting a burst response must be given the appropriate number of responses.
- *Float the buses* : you must float the buses reasonably soon, as the CPU will drive them again after only one clock of exchange time.

Figure 6.2 shows a burst read used to refill an "8-word" cache line (other burst sizes are possible, see §5.3 for how to choose burst size); MIPS "words" are 32-bit items so an 8-word line takes four 64-bit data responses to refill:

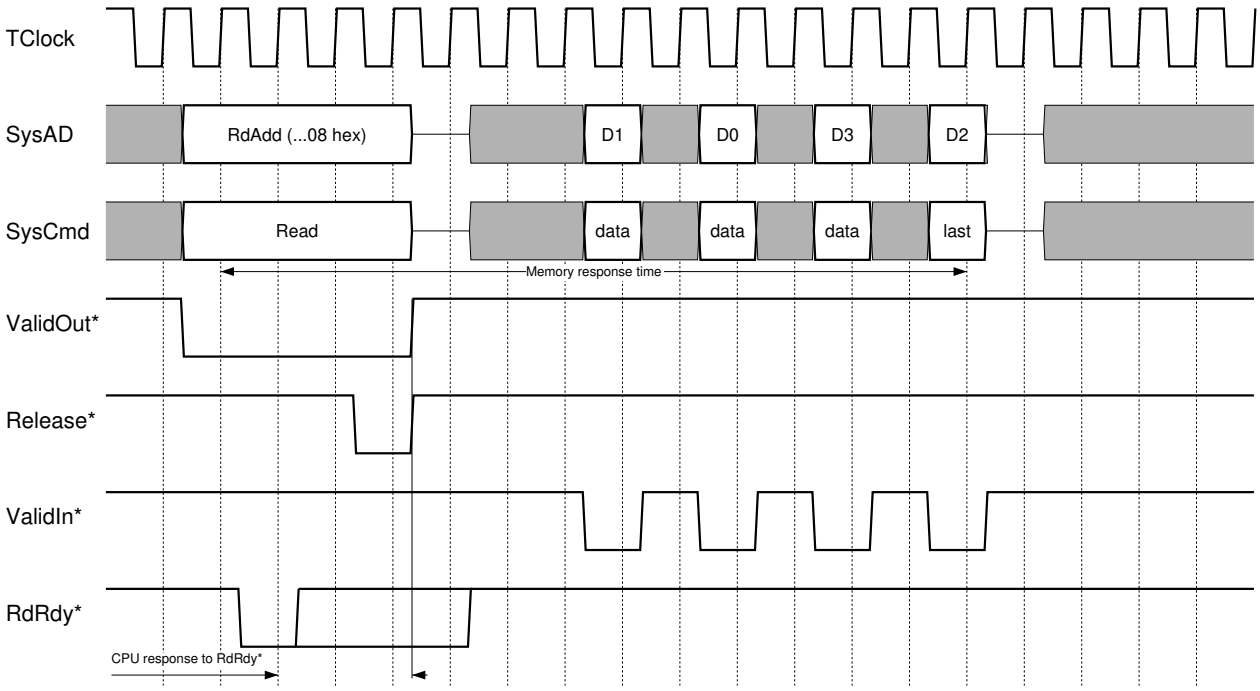

Figure 6.2 Burst read for cache refill

Notes on Figure 6.2

- *CPU release of bus* : the CPU specifications are vague about exactly when *Release** is asserted, marking the point at which the buses *SysAD* and *SysCmd* are tri-stated. It is certainly possible (and may be usual) for *Release** to be asserted during the read request *issue cycle* (the last cycle for which *ValidOut** is asserted); but no guarantee is offered and *Release** may be delayed by several clocks.
- *Burst data rate* : you may return data for a cache refill at whatever rate you like; the CPU will sample data only when *ValidIn** is valid at the rising edge of the clock.

- *Response code* : you must provide the correct response code or the CPU will get confused; however, it will not respond correctly to any response code other than the appropriate data type.
- *Data order* : the R4x00 does not refill its cache from the lowest addresses up. A cache refill will be signalled with the (8-byte boundary) address of the double-word containing the location at which it missed. The ordering scheme is called “sub-block ordering” and changes according to where you start.

Sub-Block ordering of read data

Sub-block ordering is peculiar. It achieves three goals which may be important:

- The CPU issues a word address and receives the addressed word first, followed by the rest of the enclosing cache line. This is useful to CPUs which can restart as soon as the datum that they cache-missed on is available. The R4600 does this.
- Unlike the more obvious strategy of returning data sequentially to the end of the cache line, and then “wrapping round”, sub-block ordering does not require the memory to know the length of the cache line. This motivates the use of sub-block on the i486 and derivative CPUs, where the bus protocol gives a last-data indication to memory.

  The R4x00 requires the memory system to know the length of a cache line (it can do this by decoding the read request), but by then sub-block ordering was emerging as a likely standard.
- When you start at the lowest word of the cache line, it reduces to sequential order.

A sub-block is an aligned sequence of 2, 4, 8 (and other powers of 2) words, smaller than a cache-line. In sub-block ordering the words are fetched so as to “fill” successively larger sub-blocks until we have the whole cache line. Once the first sub-block is filled each sub-block is tackled in the same order as the first one was.

Figure 6.3 shows how it works for a 4-word line.

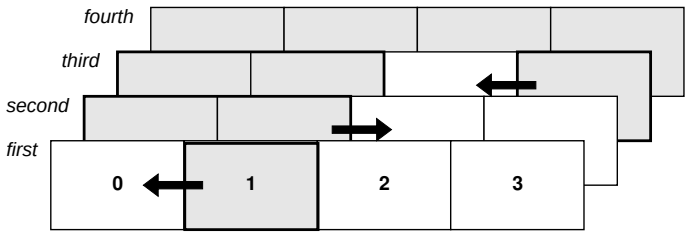

Figure 6.3 Principles of sub-block ordering

Since, where the starting address is the first word of the block, the order is just sequential, the R4x00 can request a sequential transfer just by ensuring that the address is that of the first word in the cache line. This is always true for cache line writebacks.

One irritating feature of sub-block ordering is that because the order changes according to the initial address, the memory system has to be able to sequence through different addresses too; this discourages the use of sequential-access parts such as nibble-mode DRAMs.

Table 6.1 shows what the addresses look like:

Starting doubleword address within line	*fetch order, doubleword addresses within line*					
	8-word line				*4-word line*	
0	1	2	3	4	0	1
1	1	0	3	2	1	0
2	2	3	0	1	-	
3	3	2	1	0	-	

Table 6.1: Sub-block ordering of data

6.2. Reducing read latency

Running a large program the R4x00 (1992 version with dual 8Kbyte caches) will suffer a cache miss rate of the order of 2-3%, which will turn into about 50% of real time spent waiting for a cache refill. Later processors sometimes have bigger or cleverer caches; but then they also run at higher clock rates. If you can complete a cache refill 10% more quickly this will turn into about 5% of extra system performance - not to be sneezed at.

Although more complex designs often increase bandwidth, the lowest latency usually goes with a fairly simple design - so a very ingenious R4x00 design is likely to be a mistake.

Notes on Figure 6.4

Just to set you thinking, Figure 6.4 is a timing diagram for a single-bank DRAM system for a 50Mhz bus clock, using 60ns DRAMs, and just about as fast as it can be.

- *Flow control* : *RdRdy** is inactive by default and activated in response to *ValidOut*;* even for this fast a memory system, this doesn't slow the read down at all.
- *DRAM multiplexed address* : driven by registers whose input is *SysAD* and clocked with *RClock;* here used not only to maintain hold time but to gain a few nanoseconds.
- *DRAM RAS* : *Ras** is clocked from *TClock* to provide minimal address setup time on the DRAMs. Also means that the DRAM multiplexed address change (triggered from *RClock)* is just late enough to guarantee the 10ns hold time.

  Note that *Ras** may be too early to decode the address and ascertain that this cycle is really going to private memory. No matter; you can let a *Ras** start for any cycle but don't issue a *Cas** if it turns out that no access to private memory is required.
- *DRAM CAS* : clocked from *TClock.* If only the RAS access time was a bit better you might use *RClock* to scrape the data back a clock earlier; this can be an exercise for the reader who can find a faster DRAM specification.
- *DRAM data* : will now be valid as defined by the CAS access time of 20ns. Various options exist for how to clock it, but this paper design uses the rising edge of *Cas** itself (delayed through a buffer or gate) to define the sampling point of a register driving *SysAD.*

Eight clocks to return the last 64-bit chunk of a 4-word (2×64-bit) cache line is a good target for a simple tightly-coupled memory system. In principle a similar but bank-interleaved memory system could save two clocks, providing the data from the two banks in consecutive bus cycles. Over to you....

6.3. Error reporting

Two different kinds of errors can be triggered during a read cycle:

- *Read bus error* : signalled by the "bad data" bit in the *SysCmd* encoding for the data response. This causes the processor to take a read bus error exception. This is the best way to tell the processor when you know something is wrong; bus timeout or downstream parity check fails.

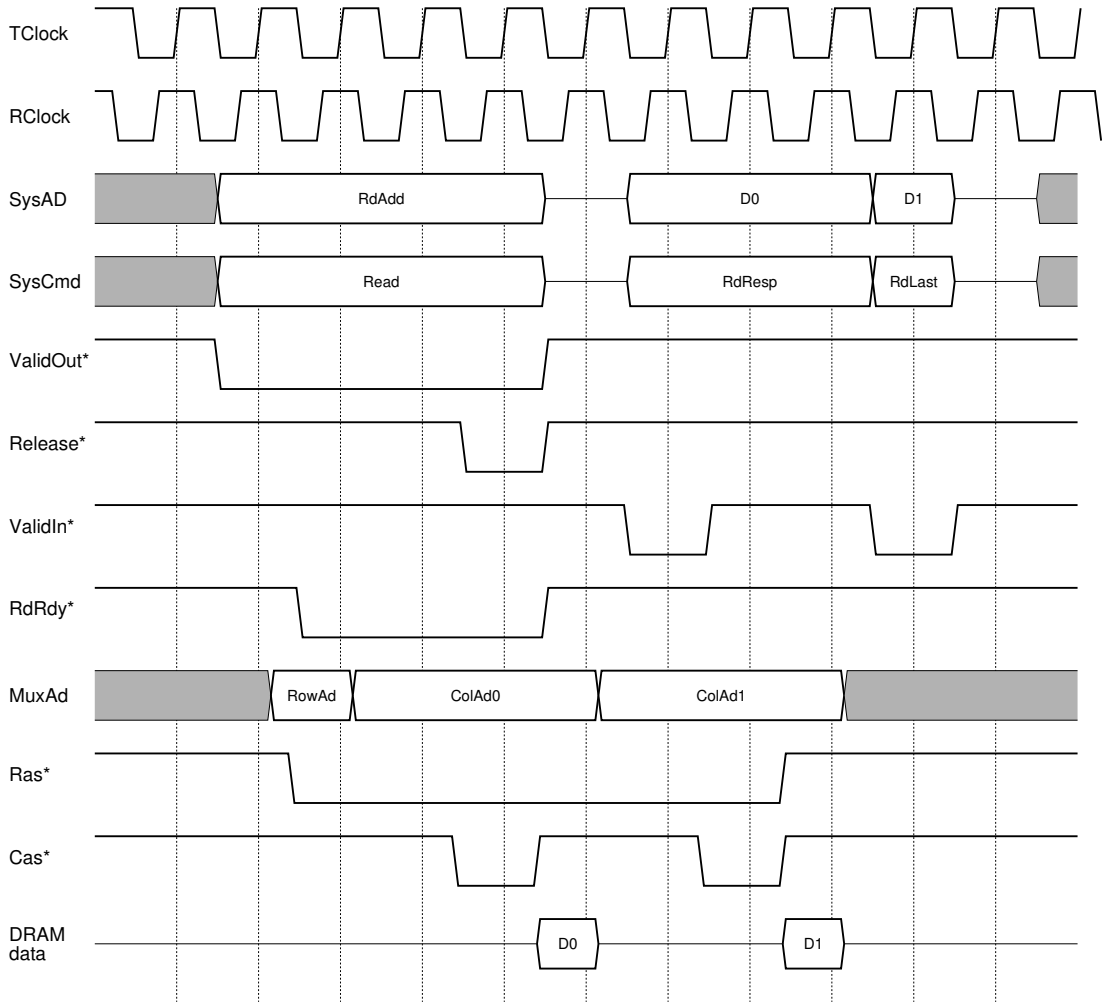

Figure 6.4 Low-latency DRAM read - critical signals

- *Bad parity on supplied data* : caused by the processor discovering bad parity on incoming data, for which the *SysCmd* response code did not disable parity checking. In this case the processor will take a cache parity error trap. Depending on your CPU type this may happen on the fetch which triggered the miss, or it may happen only when you reference the particular word which had a parity error.

  If (as is economical) you elect to take memory parity right through to the R4x00, don't forget to warn your software engineer.

Beware: although these errors can only occur on a hardware read cycle, the R4x00's cache policy means that a store instruction may stall while the hardware gets the line in you want to write. Programmers need to be aware, then, that data bus errors and cache parity errors can occur on a store.

7. Write cycle

Write cycles are simple, but fast. The R4x00 likes to throw data out and assume someone else has caught it. However, the R4x00 does cache write back only *after* it has performed a read refill (old cache line data is moved into a temporary buffer while the read takes place), so cache write backs don't stall the processor. Slow writes matter only because they are likely to hold up the processor's *next* read, worsening average read latency. So:

- memory write performance is not nearly so critical as read; and
- the significant element of write performance is latency, measured as time to complete the write and free-up the memory system.

7.1. How it works

Figure 7.1 shows a 2-cycle (4-word) cache write burst, with *WrRdy** used in the "pessimistic" manner (see §5.4 above):

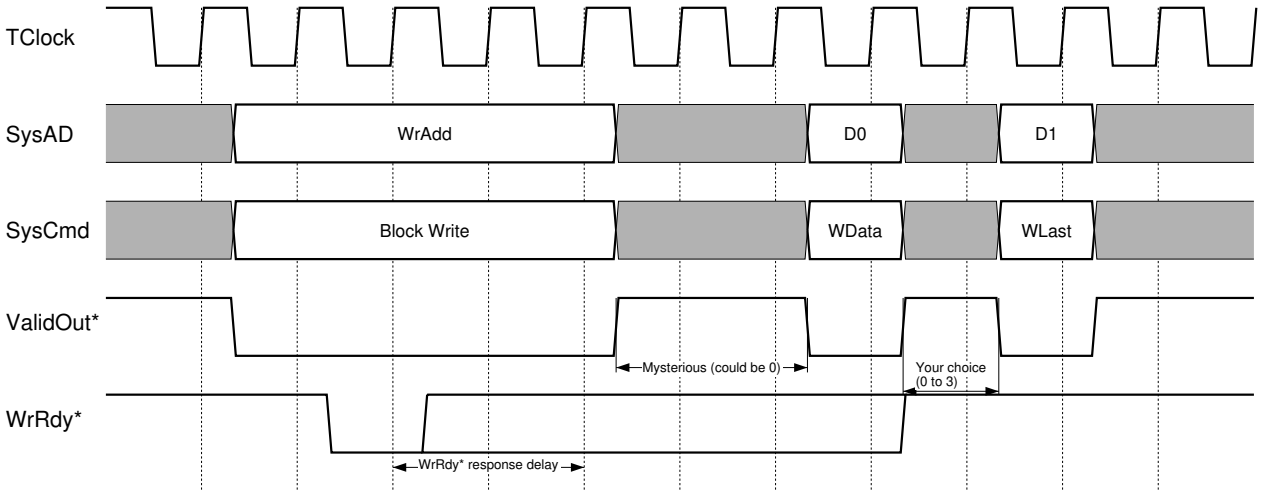

Figure 7.1 Structure of a write cycle

Notes on Figure 6.4

- *Request and address* : will be held on the bus, since *WrRdy** is initially inactive. Once *WrRdy** is activated, there is still a two-clock delay before the CPU notices and proceeds with the cycle.
- *Request to first data* : although most MIPS diagrams show write data appearing in the clock immediately following the request issue cycle, the manual is actually careful not to promise it. Early R4x00 parts exhibited a 2-cycle gap for cache writebacks (bursts), and no gap for single- or partial-word writes. We recommend that your logic be able to deal with anything from no gap to a 6-cycle gap.

  In any case, data is made valid for only one clock, marked by *ValidOut**. Because this is only one clock you have to wrestle with hold time problems.
- *Data rate* : you can select the write data rate using the reset-time configuration mechanism. This allows you to configure the gap between the two cycles of a pair (and for longer bursts configure the gap between pairs). The rate can be selected between one per clock and one per four clocks.

  The documentation implies that the data rate configuration has no effect on the request→data delay. "No guaranteed effect" might be nearer the mark.

- *Write to write timing* : Figure 7.1 shows that the signal *WrRdy** is a don't care to the CPU from the point where it gets sampled active (allowing the write to issue) through to two cycles before the last data. From that point on, though, it must be in the correct state because its value can be sampled there and lead the CPU to decide whether or not to issue a subsequent write.

  But a non-burst write can take only two cycles (address issue and one data), and it is only possible to set *WrRdy** back to inactive once you've seen the issue. This means that any subsequent write starting in the next or next-but-one clock after the data cycle would respond to the left-over *WrRdy** value and overrun. To save the write flow control strategy, the rule is that no write can start in the two clocks following an uncached write.

  Note that the R4600 CPU (one of whose design goals was to increase the uncached write transfer rate) can be setup into a mode ("pipelined writes") where this rule doesn't apply. In this mode R4600 is not 100% compatible with the normal R4000 system interface, and the board designer must always be willing to accept *two* single write cycles whenever *WrRdy** is active.

7.2. Implementation strategies

The main difficulties are:

- the uncertainty of the data timing during block writes;
- planning for partial-word writes.

Data timing

You cannot predict the timing of R4x00 data until the first valid data code is seen; after that the timing of further data will be rigidly set to whatever transmit data pattern has been selected at reset time.

Partial word writes

Memory systems using some kind of error-correcting code have to implement a partial-word write as a read-modify-write cycle. Systems using per-byte parity (or without any protection scheme) can just update the selected bytes; though this may not be as convenient as it looks since it is awkward to arrange high density DRAM stores in 9-bit units.

The R4x00 gives you little help with either approach.

7.3. Write errors

There are likely to be a number of reasons why an R4x00 write cannot be successfully carried out. Since there is no response on the bus, the only way to signal them will be as an interrupt. Apart from a write to non-existent memory, conditions to look out for include:

- *Write too wide* : too many bits being written at once to a memory space which is narrower (most I/O subsystems will be 32-bit or less);
- *Writeback to inappropriate space* : attempt to perform a burst cycle to a memory space which cannot support it;
- *Write parity error* : it is possible for the CPU itself to generate bad parity, if it writes back a cache line which itself contains an error. It should flag the bad parity with the "error" bit in the data identifier driven on *SysCmd.*

  If you are using "end-to-end" parity into the memory system you can just store the data into memory, which will produce an error if and when the data is re-used.

Note that these are conditions which will never happen in a system where the hardware and software are working correctly. There is usually a limited cost and space budget for diagnostic features, and it may be better spent somewhere else. It may not really matter if you just quietly ignore illegal writes.

8. Three R4x00 system recipes

Every designer starting to get to grips with the R4x00 faces some common problems, but one major and universal one is the width of the data and address paths and the consequent large pin and component count. As Figure 8.1 shows, just wiring up the data and address paths for a 2-way bank-interleaved gets close to 500 pins, without counting internal connections in the module which implements the address and IO bus connections:

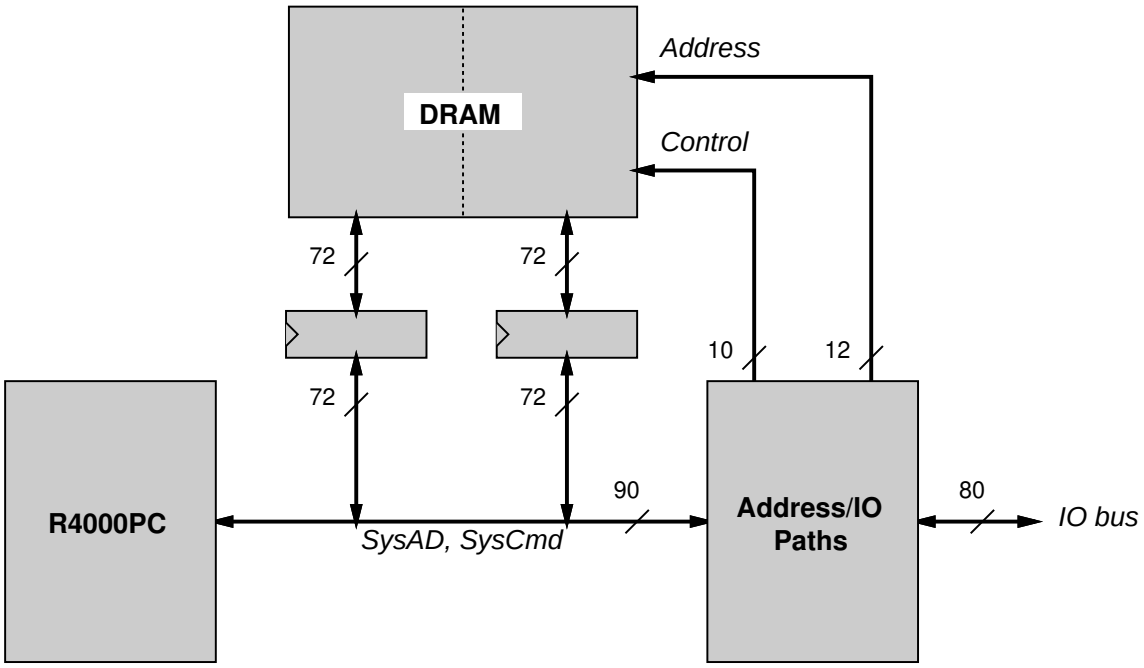

Figure 8.1 Where the pins go on a R4x00 system

Other problems include:

- *Tight clocking requirements* : output signals are not valid for very long, and inputs must be tightly controlled to meet setup and hold times. It is difficult to prevent the R4x00 interface from becoming a logic block in its own right, which it shouldn't be.
- *Hold time* : a problem on both inputs and outputs.
- *Local memory system* : it is hard to imagine any application of the R4x00 processor which does not involve several megabytes of memory. Performance is so influenced by read latency that some part of this memory must be tightly coupled to the processor.

  We think that the large majority of R4x00 implementations will involve the provision of some quantity of high-speed DRAM store. The CPU/memory combination is critical to keeping performance high and complexity low. So all the recipes here will describe different ways of interfacing the R4x00 to a DRAM array.

8.1. Tightly coupled single-bank memory system

Figure 8.2 is close to the minimal number of gates which can be used to wire an R4x00 to anything.

Notes on Figure 8.2

- *CPU and memory control* : are tightly integrated, and the memory is controlled by synchronous state machines running off *TClock.*

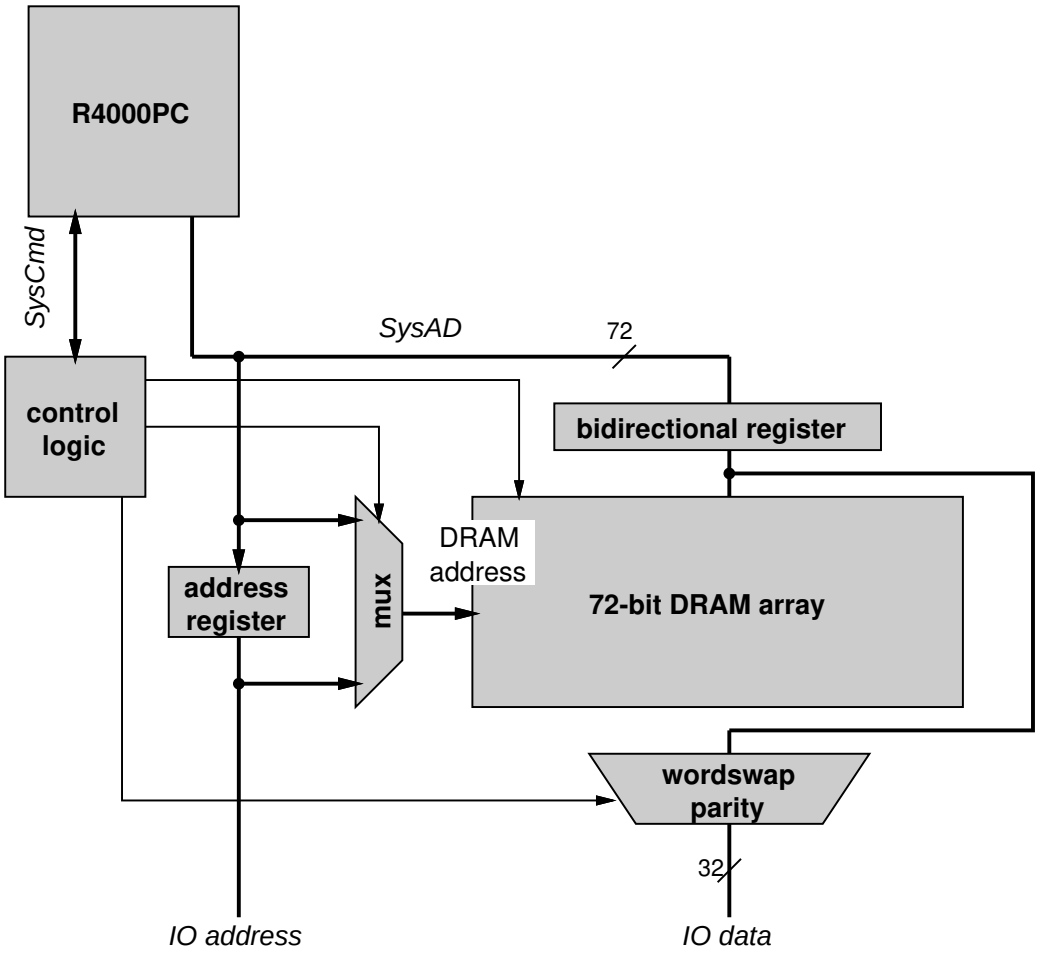

Figure 8.2 System with tightly coupled single bank memory

- *Memory parameters* : this is a 72-bit memory - more or less minimal; while it is obviously possible to wire a R4x00 up to a narrower memory it seems unlikely to be a common option.
- *Data path* : you have to have a register between the R4x00 and the DRAM data pins, because there is no way to ensure enough write data hold time from the CPU to permit DRAMs to sample the data directly.

  We recommend a register with clock enable, so at least a single datum can be captured and held. The register should be triggered from the *RClock* rising edge, but the clock enable pin can be safely generated by your fast control logic running from *TClock.*

  Since a register will also be required for IO system data transfers it is likely to make sense to take all data through the DRAM data bus; that's what is shown in this picture.

  The IO side will now have to be connected through buffers which convert between a 32-bit data bus and the 64-bit plus parity DRAM data bus.
- *Address path* : there are several different strategies for address management, but the one shown here is one of the simplest.

  Neither IO addresses nor data have to travel along *SysAD* at any time, so there is no need to use the "external request" protocol to float the R4x00 off its local buses (see 9).

- *Performance targets* : the single-bank memory can achieve a burst rate of something around 22Mhz using commodity DRAMs, and the total memory latency need not be too far away from the access time of the DRAM chips themselves.

Problems and suggestions

- *Hold time on CPU outputs* : address and data registers sampling CPU outputs can be run from *RClock;* other CPU outputs can be sampled with PLDs or with registers fronted up by other logic elements.
- *Hold time on read data* : data being passed back from memory or I/O will be valid for longer than one period of the bus clock.

Advantages and disadvantages

This is a relatively economical solution both in space and buffering, and in numbers of DRAM components employed.

The 72-bit DRAM bank leads to a lower burst bandwidth than is achievable with a bank-interleaved system. However, the R4x00 can only support cache lines of 8 32-bit words; so the longest burst read consists of only 4 64-bit data. The penalty of a memory system capable of transferring data on every other bus clock period is between 1 and 3 bus clocks (according to the cache line length).

8.2. A simple (bank-interleaved) discrete solution

If you want better memory (and hence system) performance you will need to bank interleave the memory. While some small improvement in burst rate is achievable by putting two ranks of DRAMs onto a common data bus, a large improvement follows from building a discrete registering mux/demux stage (usually with two sets of bidirectional registers with *SysAD* connections commoned).

Following the easiest path leads to the system shown in Figure 8.3.

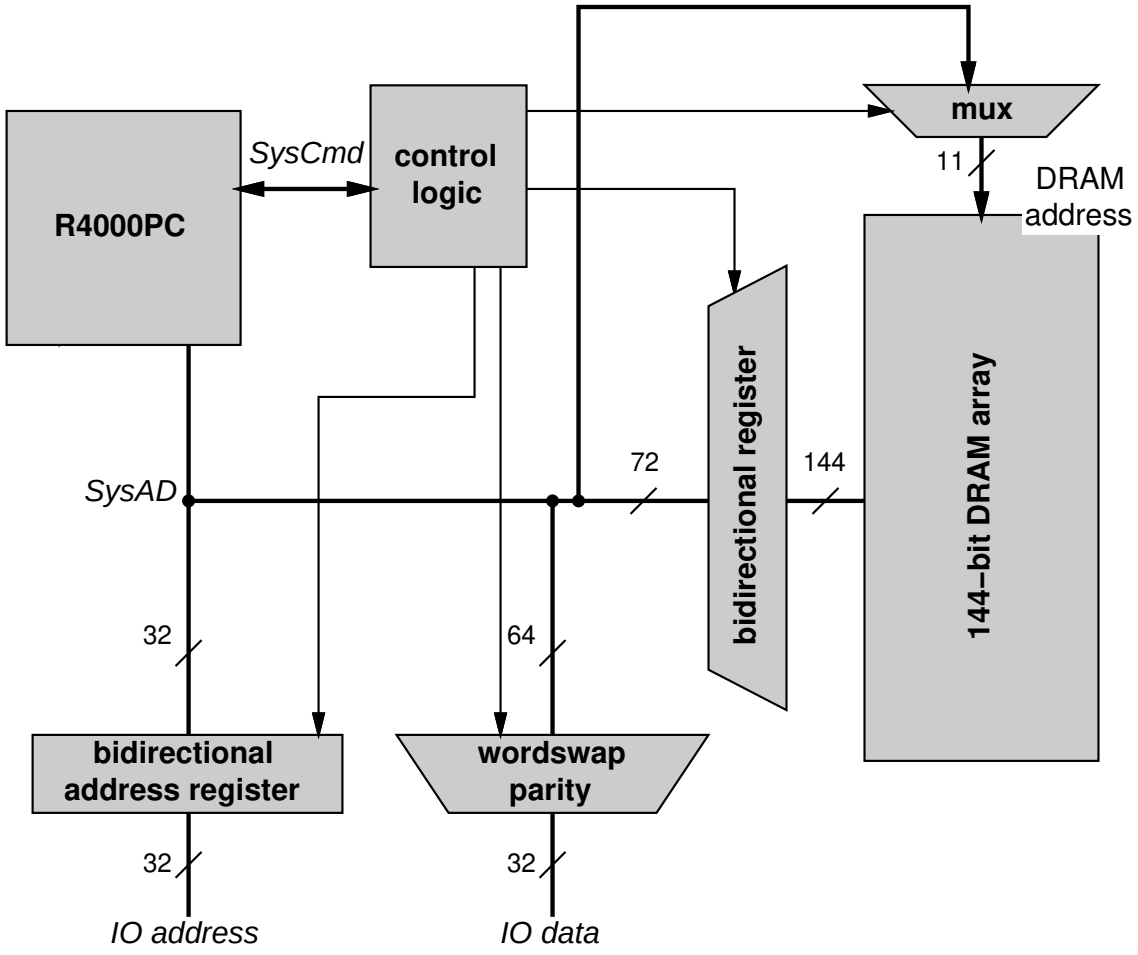

Figure 8.3 Bank interleaving

Notes on Figure 8.3

- *Memory parameters* : a 2-way interleaved memory (like this one) using 60-80ns DRAM components will run up to about 45Mhz. The benefits of higher degrees of interleave are doubtful, since the additional complexity and loading of *SysAD* may prevent you from taking advantage of the potentially higher burst transfer speed.
- *Data path* : now that the memory data bus is 128-bits plus parity, it is no longer sensible to attach the 32-bit IO bus to it; too many components will be required. It is simpler to plumb the IO bus via *SysAD,* even though it means using the external request protocol to get the R4x00 to tri-state *SysAD* while the I/O system uses it.

- *Address path* : the minimal solution is to wire the addresses via *SysAD* (since we have to tri-state it for the data) and have one set of multiplexers to generate DRAM addresses.

Problems and suggestions

- *Buffer explosion* : the 128-bit (plus parity) data path inside the DRAM array leads to a large component count, and needs lots of data buffers/registers. It needs lots of DRAMs too, and it may be a mistake to use bank interleaving except where the system memory requirement is large enough to need two banks of components.

  There is little to be done about this except to keep the data path simple and use the densest components you can get.

8.3. ASIC-based solution

You don't necessarily have to design your own ASIC; since (as we have argued here) most R4x00 CPU/memory cores share common features, there is a market for a generic memory/IO controller chip or chip set. Several chip sets have been announced; few taken through to production. Most of those were targetted for the Windows/NT workstation market, which has not materialised - and often are so dedicated to the Microsoft-defined architecture as to be unsuitable for use elsewhere. But ask round the R4x00 semiconductor vendors, and see what they can tell you about.

For high- and moderate-volume applications it may be worth building your own ASIC, though support of such high clock speeds will require a technology with good I/O buffers. The total pin count of any useful system which handles the data and address paths will be big; the solution is going to be a set of high pin-count devices, certainly not a single part.

The logical use of an ASIC is to act as a tri-port connection between CPU, local memory and the I/O bus which represents the rest of the world, as shown in Figure 8.4.

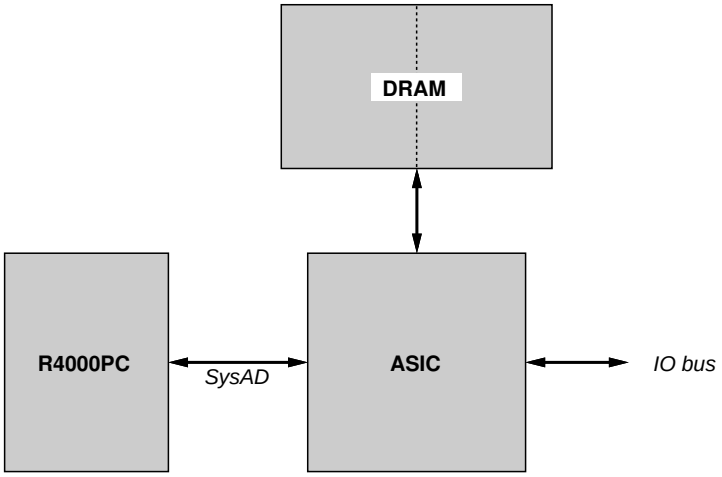

Figure 8.4 Use of an ASIC in the data path

Since this application note was first written PCI has emerged from nowhere to become the inevitable first candidate I/O bus for any system of this class. PCI's 32-bit width and 33MHz burst cycle gives it a maximum bandwidth of 120Mbyte/s; only video applications are likely to need more.

Ideally, your ASIC kit should come with a cookbook (making this application note largely superfluous.) But the critical design aims are the same. And you may find that the application note stimulates the thinking you will need to review a prospective ASIC for usefulness.

Advantages and disadvantages

The ASIC should reduce space requirements, power consumption and bring the ability to operate at higher system clock rates. It brings the possibility of adding a little storage, particularly for write bursts and IO/memory transfers, which can be of significant advantage in the system. However, unless specified carefully it can result in worse latency (and hence lower performance) than a careful, simple, discrete design.

9. External Requests

To pass information into the R4x00 (and with some systems, to use the local memory), someone else has to drive the bus. As we have already seen this opportunity is magically available on a read cycle, but there are other times when a command or data has to come from outside. This is controlled by the External Request protocol.

The CPU tri-states *SysAD* and *SysCmd* during a read cycle to allow the data response code and data to be passed back. The CPU will usually drive these buses at all other times, even though the value to which it drives them is undefined.

However, the CPU has a mechanism by which it can be requested to give up the bus to allow something else to happen. This can be done for two reasons:

- *Send a request to the CPU* : in the R4400MC (multiprocessor support) variant, it is frequently necessary for the shared bus interface to query or update the state of the CPU caches. These are all implemented as external requests.

  It is also possible for the CPU to make some of its internal state visible as registers which can be read or written by external requests. In practice this has been whittled down until only some interrupt bits can be so accessed; so that an external write into the CPU can be used to alter the interrupt state. The R4400MC and R4400SC variants have only a single dedicated interrupt pin, so this mechanism is required for efficient sophisticated interrupt signalling; the small-package variants have a pin for each interrupt, and there is no reason to do such a complicated thing.

- *Use the bus for something else* : it may be convenient, in your system, to use *SysAD* as a data highway during transfers between local memory and some DMA device or alternate processor. This is done in the bank-interleaved system recipe above (§8.2).

  Once the CPU has agreed to relinquish its bus, so long as *ValidIn** remains de-asserted the CPU will ignore the state of the *SysAD* and *SysCmd* buses.

  Once the DMA operation is completed the CPU must be re-started. Since the CPU's view is that the only reason to start this procedure is so you can issue an "external request", you should issue the "null request" code (which does nothing else to the CPU).

  Despite the intricacy, all you are achieving is to reproduce the "hold/hold-acknowledge" mechanism of an 80x86 family CPU.

Notes on Figure 9.1

Figure 9.1 shows how this can be done.

- *Latency* : the response of the CPU to *ExtRqst** is unpredictable and complex, and is discussed below.
- *You must not change your mind* : having once asserted *ExtRqst** you must keep it asserted until you see a *Release** from the CPU.
- *Bus handover* : the CPU pulses *Release** for one clock period; as *Release** is de-asserted it tri-states *SysAD* and *SysCmd.* You should wait one clock (to prevent contention) and then drive both buses. Even if you don't need to use *SysCmd* at this stage it must not be allowed to float or the CPU may be unhappy.
- *De-asserting ExtRqst** : you must do this whenever you see *Release** and it may have been a response to your *ExtRqst*.* You have two clocks to de-assert it in; the CPU ignores *ExtRqst** in the cycle following *Release*.*
- *DMA or whatever* : you can now use *SysAD* and *SysCmd* for your own purposes (taking care that they are not allowed to float to invalid signal levels).

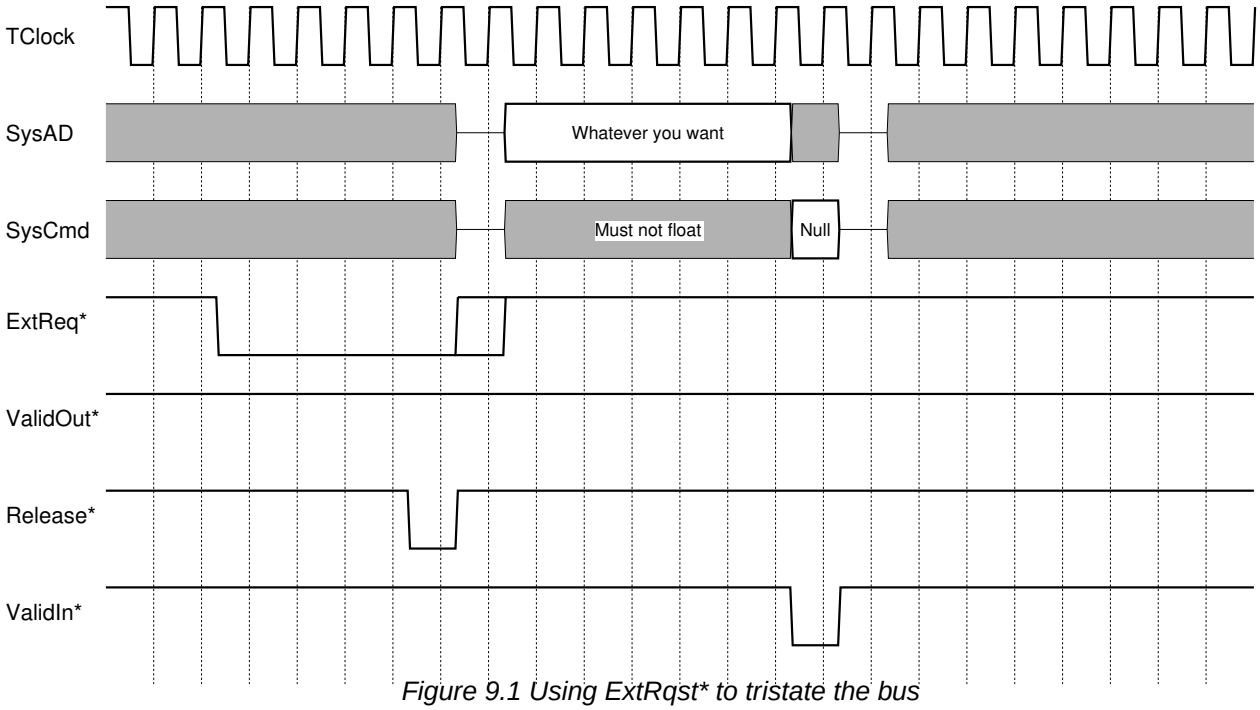

Figure 9.1 Using ExtRqst to tristate the bus*

- *Null request* : this *SysCmd* code is used (together with *ValidIn*)* to tell the CPU that it can have its buses back. Your drivers should tri-state *SysAD* and *SysCmd* as soon as the CPU has had a chance to sample the null request.

9.1. CPU response to external request

In general you should just wait for a *Release*;* (it is analogous to those CPUs where a “hold” request will, after a while and at the CPU’s discretion, produce a “hold acknowledge”. However, there are a few guarantees and one horrible ambiguity. Guarantees first:

- *Waiting for RdRdy* or WrRdy** : if the CPU is driving a read or write request, but is waiting for the assertion of *RdRdy** or *WrRdy*,* then the CPU *will* give up the bus before the read or write cycle is issued. To prevent confusion you should not assert either *RdRdy** or *WrRdy** at all, until the external request sequence finishes.

  After a short while the CPU will release the bus. When the bus is passed back to the CPU the interrupted read or write request will appear again.
- *Write in progress* : if the CPU is in the middle of a write then it will finish that cycle but will not start another. Once the cycle is complete *Release** will be asserted after a few clocks (the precise number is unpredictable).
- *Bus idle* : even when the bus is idle up to the time you assert *ExtRqst** and for a couple of clocks afterwards, it is still possible that the CPU may be internally committed to a read or write. If so everything occurs as if the cycle had already started; otherwise you will see *Release** without any further CPU activity.
- *Read in progress* : if a read cycle has progressed to the point where the CPU has released the bus, the documentation claims that you may either proceed with the read cycle (giving the CPU a read response), or proceed with an “external request” to the CPU.

  Note that since, while waiting for a read response, the CPU is not driving *SysAD* or *SysCmd,* you can always use these buses anyway.

- *Danger point - distinguishing reads and external requests* : if you issue *ExtRqst** sometime just before when the CPU reaches the issue cycle for a read request, the CPU will pulse *Release*.* But how can you tell whether this is because the CPU has responded to *ExtRqst** or is just waiting for read data?

  The official specification implies that you cannot find out. However, at least for the limited range of bus protocols supported by the R4x00, you may treat the release in the same way. You will have to observe the following rules:

  a) you *must* de-assert *ExtRqst** whenever *Release** is seen at a time when the CPU *could* have seen the external request; because of the input registers, this means "when *ExtRqst** has been active for at least two clocks".

  b) if you choose to go ahead with an external request (null or otherwise) to the CPU before you provide the read data, the protocol obliges the CPU to re-establish bus ownership. In order to be able to accept a read response, the CPU will then pulse *Release** again after the external request is processed. This always means that the CPU is tri-stating its buses again, so it is ready for the read response.

  Some information implies that the CPU will let you either give it the data and the null command in either order. But you'll always have to wait for the second *Release** pulse.

  However, the documentation is sufficiently unclear that it may be more prudent to de-assert *RdRdy** and *WrRdy** in anticipation of needing an external request. Two clocks later you can assert *ExtRqst*,* confident that you know what will happen.

- *Danger point - bug affecting cache refill/writeback pairs* : in early R4x00 versions an external request would cause problems if it slipped in between a cache-refill read and an associated cache writeback.

9.2. Tri-stating the bus without affecting the CPU

In some applications it is convenient to use the *SysAD* (and/or *SysCmd)* signals as a common highway for transfers controlled by logic other than the CPU. This can be done either by causing the processor to tri-state the buses using *ExtRqst** or by taking advantage of the bus release inherent in the read cycle. Once the processor signals that it has relinquished the buses by asserting *Release** you can do anything you like with the buses, provided that:

- you *must* drive all the bits of *SysAD* and *SysCmd* to some valid logic level (even though you may not be using *SysCmd* for any purpose); the CPU may not operate correctly if these signals float to near the logic transition level.
- during your use of the buses you must keep *ValidIn** deasserted - which is enough to prevent the CPU "seeing" any activity on its buses.
- to return the buses to the CPU you must drive *SysCmd* with the special "null but return bus" code defined in §5.3, accompanied this time by *ValidIn*.*

Note that however you accomplish DMA access to memory on an R4x00 system, you need to manage cache coherency in software. See Appendix C ("Software cache management with DMA") on page 52 for details.

9.3. Access to the interrupt register

The only thing you can do to the CPU by taking over the bus is to manipulate the interrupt bits in the CPU status register. This allows you to simulate the assertion/de-assertion of the interrupt lines *Int(5:1)*,* which are not accessible on the R4400SC and R4400MC packages.

We'd suggest that you should not to use this facility on the R4x00 unless compatibility with an R4400SC is important; the pins are available and are much simpler.

If you have to do it, here is how. You must perform a 64-bit write request. Most of the address is irrelevant, but *SysAD(6:4)* must all be zero. The data bits the CPU is sensitive to at this location are shown in Table 9.1:

63 … 23	22	21	20	19	18	17	16	15 … 7	6	5	4	3	2	1	0
ignored	write enables							ignored	set/reset bits						
	NMI	equivalent *Int**							NMI	equivalent *Int**					
		5	4	3	2	1	0			5	4	3	2	1	0

Table 9.1: Data encodings for write to interrupt location

Notes on Table 9.1

- *Write enables* : seven bit positions, with bit 22 being for the non-maskable interrupt and bits 21–16 for the six regular interrupts. Note that these affect cause/status register bits called “IP7” through “IP2”, but that these are equivalent to the R4x00 pins called *Int(5:2)*.*

  Writing a 1 to any bit allows this command to set or reset the corresponding status register bit. A 0 here means that the corresponding interrupt bit will remain unchanged.

- *Set/reset bits* : in the same order as the write enable bits. Where any write enable bit is set the corresponding set/reset bit determines whether the status register interrupt bit is made active (1) or cleared (0).

It is quite possible to use both the direct and external request interrupt mechanisms; the CPU “ORs” the two sources together to compute the cause register bits.

10. Starting up the R4x00

Some R4x00 variants require significant amounts of configuration information to be loaded before they can reasonably be expected to run any instructions, and this is done at reset time as a 1-bit serial data stream advanced by a CPU-supplied clock.

The R4x00 makes much use of internal phase lock techniques to generate high-speed clocks and interface clocks with precisely controlled phase relationships. While it is getting these going it requires a well-controlled sequence of reset signals, with a number of relatively long waits. It also follows earlier MIPS practice in refusing to guarantee the state of output clocks from power-on, until the reset signals are established in their active state.

10.1. Configuration Options

CPU configuration options are encoded into a bit stream delivered to the *ModeIn* pin during the reset sequence. Bit 0 is presented before the assertion of *VCCOk,* and the next bit should be presented on each rising edge of the *ModeClock* output. The sequence is defined as up to 256 bits long, though we think no CPU has yet shown any interest in bits after number 64. After 256 bits *ModeIn* is ignored forever (at least until it is again reset).

Table 10.1 shows how options for the R4400 are encoded[5]. Bit 0 comes first. In multi-bit codes the value is an unsigned integer made up by taking the bitfield and regarding the lowest-numbered bit as the least significant.

Bit Nos.	*Meaningful Values*	*MIPS Mnemonic*	*What it does*
0	1	**BlkOrder**	Cache refill order sub-block (sequential not available for small-package variants).
1	0/1	**EIBParMode**	1 = per-byte parity, 0 = ECC check
2	0/1	**EndBit**	0 = little-endian, 1 = big-endian
3	1	**DShMdDis**	disable “dirty shared mode” - would be relevant only to a cache coherent multiprocessor
4	1	**NoSCMode**	1 = no 2ndary cache present
5-6	0	**SysPort**	64-bit interface
7	0	**SC64BitMd**	128-bit secondary cache width
8	0	**ElSpltMd**	0 = unified secondary cache
9-10	1	**SCBlkSz**	secondary cache line size (1 = 8 words)
11-14	0–8	**XMitDatPat**	System data write rate (see Table 10.2 below)
15-17	0–2	**SysCkRatio**	Pipeline clock/bus clock divisor. 0 = 1:2, 1 = 1:3, 2 = 1:4
18	0		reserved, must be zero
19	0/1	**TimIntDis**	Determines role of *Int5** and availability of interrupt from internal timer. 0 = internal timer interrupt enabled, pin is a no-connect 1 = no internal timer interrupts, interrupt through pin.
20	1	**PotUpdDis**	Disable “potential updates” (cache-coherent multiprocessor option)

[5] Options for other parts are defined in Appendix B (“Alternative configuration stream data sets”) on page 51 below.

Bit Nos.	*Meaningful Values*	*MIPS Mnemonic*	*What it does*
21-24	0	**TwrSup**	
25-26	0	**Twr2dly**	
27-28	0	**Twr1dly**	
29	0	**Twrrck**	Secondary cache timing parameters
30-32	0	**Tdis**	
33-36	0	**Trd2cyc**	
37-40	0	**Trd1cyc**	
41-45	0		reserved, must be zero
46	0/1	**Pkg179**	1 = 179-pin PGA package 0 = 447-pin package for secondary cache
47-49	0	**Reserved**	Unused power-down feature (how much to slow the pipeline clock in power-down mode).
50-52	0-4	**Drv0_50** etc	Speed target for output signal clock→valid time: 001 = output delay 0.5 × MasterClock cycle 010 = output delay 0.75 × MasterClock cycle 100 = output delay 1 × MasterClock cycle
53-56	0–15	**InitP**	initial buffer current (pullup), 0 for biggest current. Set 15 when di/dt mechanism in use.
57-60	0–15	**InitN**	initial buffer current (pulldown), 15 for biggest current. Set 0 when di/dt mechanism is in use.
61	0/1	**EnbIDPLLR**	1 = sample IOut/IIn delay feedback during reset, then freeze
62	0/1	**EnbIDPLL**	1 = use IOut/IIn feedback at all times
63	0	**DsbIPLL**	enable clock-synchronising PLLs (you'd only turn them off for test)
64	0/1	**SRTristate**	1 = float outputs during Reset* (all outputs are always floated during ColdReset*)
65-255	0	**Reserved**	

Table 10.1: Configuration data stream encoding for R4400

Notes on Table 10.1

- *No choice* : in many cases there is only one legitimate value for the option in an R4x00, and in other cases the value is irrelevant. In both cases the table shows no choice of value, and you can just stuff in the value suggested.
- *Internal timer* : if the internal timer is enabled it will interrupt on cause register bit "IP7", which is the same bit as is affected by *Int5*.* This is likely to make *Int5** useless for any other purpose.
- *XMitDatPat* : the transmit data rate can be slowed down, with idle cycles interleaved between the words of a burst. Table 10.2 shows the encoding used; each pattern is represented by a string of "D"s and "×"s. Each "D" represents a clock cycle with data and *ValidOut** asserted; each "×" represents an idle cycle with *ValidOut** de-asserted. The options available are shown in Table 10.2. We cannot give you guidance on the timing of the *first* word of data; we think that early processors do deliver the first word later if a slower data pattern is selected, but the documentation does not guarantee this.

Config	*Write data Pattern*	
Code	*2-cycle burst*	*4-cycle burst*
0	DD	DDDD
1	DD	DD×DD
2	DD	DD××DD
3	D×D	D×D×D×D
4	DD	DD×××DD
5	DD	DD××××DD
6	D××D	D××D××D××D
7	DD	DD××××××DD
8	D×××D	D×××D×××D×××D
9–15	undefined	

Table 10.2: System data write pattern encodings

- *Drv0_50 etc* : used to tune the performance of CPU outputs. Can be set fast (**Drv0_50**) to optimise setup time to the next clock; can be set slow (**Drv1_00**) to save power, to reduce noise and to increase hold time.

  Output time tuning uses the dummy track and loading between *IOOut* and *IOIn.*

- *InitP*, *InitN* : tune output speed (by adjusting the pull-up and pull-down current) when the *IOOut, IOIn* feedback mechanism is not operating, as always happens early in the power-up sequence and may happen if you configure the processor to turn it off. You're probably better off setting both to about three-quarters of maximum; you may like to tune it once your design is stabilised.

10.2. Choosing configuration options

10.2.1. Parity vs ECC protection

Codes to check on data integrity have two purposes:

a) keep a system running correctly in the face of a very small random incidence of errors. This is the motivation for the use of error-correcting codes in large memory systems. A parity check permits the system to keep running only if combined with some kind of higher level recovery.

b) diagnose data path errors as such, before they lead to widespread and confusing consequences.

The kind of errors likely to occur on a data path in a tested system, outside of a very large memory array, are either inherently fatal (eg connectivity failure) or multi-bit in nature (eg noise of various kinds). In either case ECC cannot reliably recover and either scheme will quickly diagnose the error.

We do not, therefore, recommend anyone to use the more complex ECC scheme in an R4x00 for better protection of the data path.

However, a useful way to employ the R4x00's data checking is to use it end-to-end to and from the local memory array. In this context ECC protection may perform slightly better; the down side is the complexity of generation and checking for any non-CPU data path, and the inability to perform partial-word update to an ECC-protected memory.

We think that discrete data path implementations should use parity and not be ashamed of it. ECC might be a useful addition to an ASIC datapath chip set, so long as support for IO system transfers is comprehensive.

10.2.2. Endian-ness

The R4x00 can operate in *big-endian* mode (where its use of memory is like a Motorola 680x0 processor) or *little-endian* mode (where it behaves like an Intel 80x86 or DEC VAX processor).

The names originally referred to a characteristic of CPUs visible to software: how a byte-addressable machine supports numeric data types bigger than a byte. Big-endian machines use the lowest byte address to store the most-significant bits of the number; little-endian machines use the lowest byte address to store the least-signficant bits of the number. The difference is a significant cause of incompatibilities between CPUs, impacting the portability of software even though it is written in high level languages.

The reason this becomes a hardware issue is that the internal registers of the R4x00 CPU store 32-bit or 64-bit numeric data, and during word or double-word loads the register bits are “hard-wired” to the corresponding *SysAD* pins. During such word transfers the CPU implicitly assigns byte addresses to each 8-bit section of the multi-byte data, and hence implicitly assigns byte addresses within the 64-bit unit to 8-bit sections of *SysAD*.

When configured as little-endian bits and bytes are related like this:

byte address (mod 8)	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
SysAD **bits**	*0-7*	*8-15*	*16-23*	*24-31*	*32-39*	*40-47*	*48-55*	*56-63*

And when configured as big-endian:

byte address (mod 8)	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
SysAD **bits**	*56-63*	*48-55*	*40-47*	*32-39*	*24-31*	*16-23*	*8-15*	*0-7*

10.2.3. Bus clock rate

A pretty major design decision is to choose the bus clock rate. The CPU runs instructions at twice the *MasterClock* frequency, but the bus clocks *TClock, RClock* operate at some divisor of the internal clock. They therefore run at either the same frequency as *MasterClock,* at two-thirds or at half the frequency.

The most important parameter to optimise is the cache refill latency, or to be more exact the time (in *MasterClock* cycles) which elapses between a read request appearing on the bus and the return to the CPU of the *last* datum in the block. The clock time choice depends on three principle factors:

- your private memory system will be locked to this clock speed, so the clock should be (as far as possible) tuned so that your memory components are working near their highest speed with memory signals which are some multiple of a bus clock period long;
- too fast a clock and the interface logic gets hard to build and expensive;
- there are fixed overheads in cycles which always use up a number of bus clocks, so if the clock is too slow time gets wasted.

In the end you need to choose the basic clocking rate of your memory system, and then fix the ratio between that and the bus clock. The memory timing is so critical that it may well happen that system performance will be enhanced by running the processor at a *MasterClock* frequency which matches the memory well, even though this is slower than the CPU could run.

For example, suppose your system has a single 72-bit bank of DRAMS with a page-mode CAS cycle time minimum of 40ns. This means that, after allowing for logic skews, the bank cannot quite manage a 25Mhz burst rate; 24Mhz is achievable.

With a 50Mhz bus clock the DRAM will have to be cycled at 1/3 the bus speed, or 16.67Mhz.

A good target for startup time (up to the first data) is 6 clocks at 50Mhz. Three more words of data will then follow at 3 clock intervals, giving a 15-clock (300ns) latency to end of read.

If you ran the CPU at 48Mhz instead, the DRAM could burst at 24Mhz. Read latency is reduced to 12 clocks (250ns). The 16% reduction in read latency will give you around an 8% performance boost, more than compensating for the 4% lost from the clock rate reduction. And the system will run a fraction cooler.

10.2.4. Transmit data rate

During CPU block writes (cache write-back) there is no way to interact with the CPU to control the rate at which it supplies data. But you can configure various repetition rates with a view to fitting in with the requirements of your memory or buffering system; this is likely to be particularly relevant to a tightly coupled local memory which does not have a whole-block FIFO for data.

Two kinds of data pattern are supported:

- *Fixed separation between data cycles* : consecutive (0 separation) or separated by 1, 2 or 3 bus clock periods. Will be appropriate to non-interleaved memories.
- *Driven in pairs* : this makes sense only when using an 8×32-bit word cache line size, of course (otherwise there are only two data cycles).

  In this case you can generate pairs of data with a separation of 0 to 3 cycles, with the pairs separated by a longer period of up to 6 cycles. If your memory is bank-interleaved this allows you to run the cycles to different banks close together (saving bus time) while providing programmable recovery time for the individual banks.

See Table 10.2 for how to select different patterns. It may well be beneficial to configure your system with a faster system clock rate and with data throttled.

10.2.5. Output buffer timing control

The CPU drives about 85 signals in parallel when putting requests on the bus. Each of these has to be made valid with good setup time to the next bus clock rising edge. In order to guarantee these timings in systems with worst-case bus loading, the output drivers have to be powerful; but in more lightly loaded systems or those not running the bus at the maximum possible data rate the output drive will be unnecessarily large and lead to unnecessary noise and radiation problems.

The “big” (R4000 and R4400) versions of the R4x00[6] are equipped with circuitry which adapts the output drive, setting it at a level high enough to meet the timing specification but no higher.

This is done by providing the CPU with a “sample” output line *IOOut* which you should track and load in a similar way to the real bus outputs, but connect back to the CPU’s *IOIn* input. You are recommended to make its track 50% longer than the worst-case bus signal, and to load it with a capacitor value equal to the nominal worst-case capacitative load. Figure 10.1 shows the recommended layout, where *L* is the longest path from any other R4x00 output to a receiving device. The layout combines the maximum propagation time with a worst-case “reflected wave” trace (the one which goes nowhere) and maximum capacitative load. The CPU can then be configured to monitor the *IOOut* to *IOIn* transition and flight time, and to adjust its output drive to keep it within specification.

In most systems this monitoring and adjustment should be continuous (to allow the device to adapt to changing temperature) but if this causes problems the adjustment can be made only at reset time.

The drive targets are set as a nominal clock-to-valid maximum on bus signals of 50%, 75% or 100% of a *MasterClock* period. These are likely to be chosen as partners for the corresponding bus clock divisor settings.

[6] This facility is provided on all variants of the R4000 and R4400, but not provided on the Vr4200 or R4600.

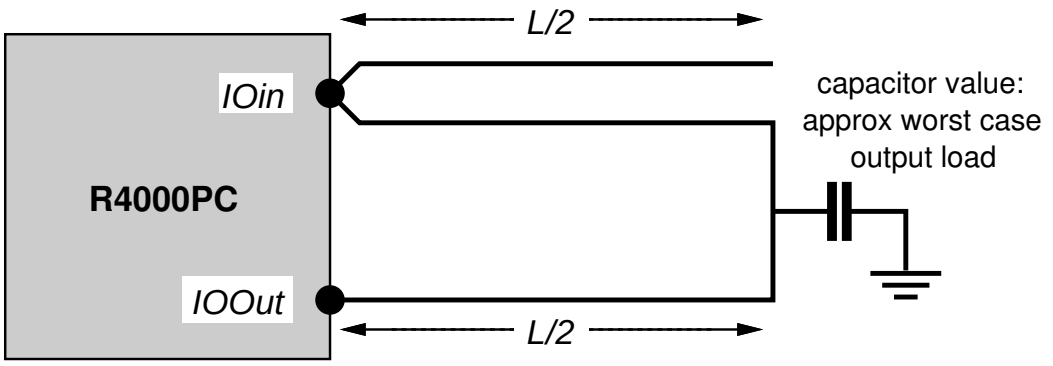

Figure 10.1 Wiring IOOut to IIn

You can turn off this whole mechanism (all **Drv** configuration bits 0), but you probably shouldn't except for testing. Even with the drive timing mechanism off you can still tune the output buffer *drive* (drive to logic 0 and logic 1 separately) to a range of levels using the **InitP**, **InitN** configurations.

10.3. Reset/configuration stream timing

The basic reset sequence is shown in Figure 10.2.

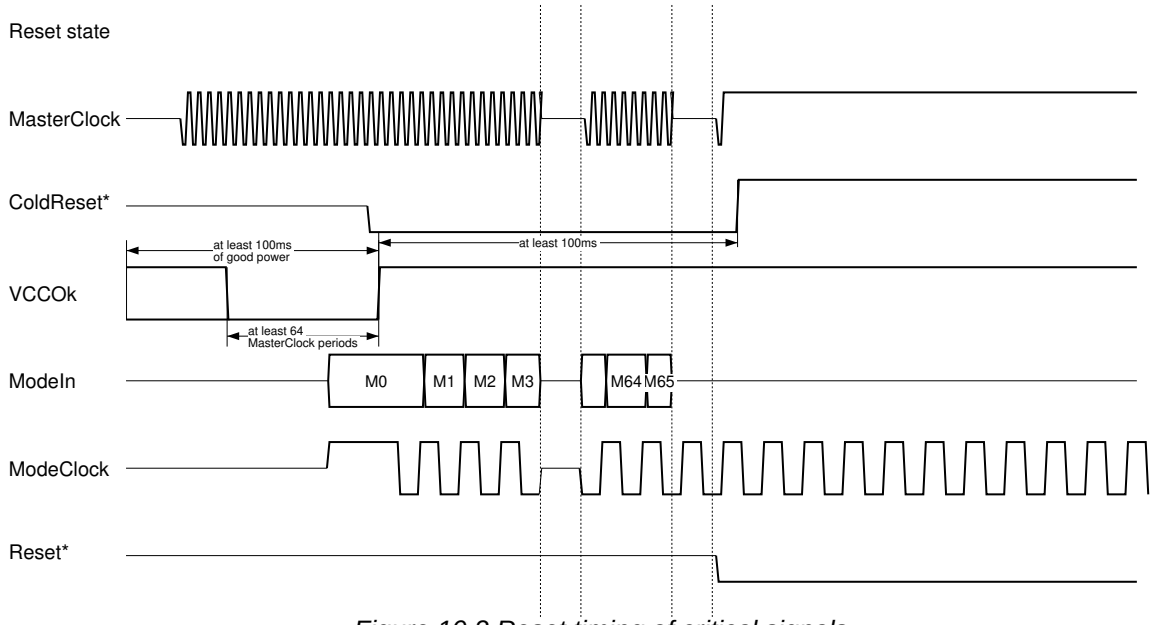

Figure 10.2 Reset timing of critical signals

Notes on Figure 10.2

- *MasterClock* must be provided from power-up.
- the full reset/reconfiguration sequence starts whenever *VCCOk* is low. Following power-up it must remain a good stable low for a continuous period of 100ms after VCC and all power supplies reach a proper level. For successful reconfiguration after power is applied, *VCCOk* must remain low for at least 64 *MasterClock* cycles.

- ensure that *ColdReset** is low at the point when *VCCOk* goes high.
- the CPU samples the first configuration bit (bit 0) around the rising edge of *VCCOk.* You should advance the configuration bitstream to the CPU on every rising edge of *ModeClock* (which runs at 1/256th of the *MasterClock* rate). Note that there is a 65th configuration bit.

  The configuration bitstream takes about 350μs to read, but *ColdReset** must be left low for much longer than that.
- not less than 100ms after the assertion of *VCCOk, ColdReset** can be made inactive. At this point the internal PLLs have settled. You must ensure that *Reset** is low before *ColdReset** is de-asserted; the transition of *ColdReset** must meet setup and hold conditions against *MasterOut,* and it is this transition which will determine the phases of the bus clocks relative to *MasterOut.*

  Since *MasterOut* may not be running early in the power-up sequence, the logic that *asserts ColdReset** had better do so asynchronously.
- *Reset** should be kept active for at least 64 *MasterClock* cycles to ensure that all the internal pipeline state is properly reset. Its de-assertion must meet setup and hold conditions to *MasterOut.* As for *ColdReset*,* it would be safer to assert it asynchronously early in the sequence.

It may be possible to perform a cold reset without reconfiguration by activating *ColdReset** while maintaining *VCCOk* high, but we wouldn't recommend it.

It is possible to cause a "warm reset" by using just the *Reset** input. The difference between a cold and warm reset is that the latter retains a good deal of internal CPU state. A more subtle distinction still exists between a warm reset and a non-maskable interrupt (NMI). Refer to [R4000 User's manual] for details.

10.4. Implementation strategies

The reset timing requirement is difficult because it has two long (100ms nominal) and two long-ish (64 *MasterClock* cycles) timing requirements. The 100ms delay between assertion of *VCCOk* and deassertion of *ColdReset** is particularly problematic. But at least you have ben warned.

The configuration bitstream will end up being stored in some sort of PROM. This is fine but you may want to be able to provide some kind of user control over some of the configuration bits; how is this to be done?

We've used two different approaches:

- There are relatively few "1" bits in a typical modebit stream; so we've encoded quite a few options in a "PAL" type device with static configuration input signals.
- In an earlier scheme we kept modebit information in a corner of the system boot PROM, with hardware to read and serialise the data at reset-time. Different options were selected by fixing the base address of the 32-byte section of boot PROM. This meant there was no limit on what bit-patterns we could generate.

  We quite liked this solution. The boot PROM is typically big enough that a little space won't be missed, and it is nice not to add another type of programmable part to your inventory. But it takes care; most of your board logic is likely to depend on the CPU bus clocks, and those clocks cannot be relied on until the CPU has configured itself.

11. A 3.3V CPU in a 5V system

Only the most power-sensitive of you will be contemplating a pure 3.3V design in the next couple of years. But you may well want to use a 3.3V CPU, because the CPU type you like best is not available in 5V, or because a 5V CPU will get too hot.

The good news is that 3.3V CMOS logic levels are compatible (for the great majority of signals) with 5V "TTL-level" logic. But the bad news is that input signal voltages which go too far above the 3.3V level are liable to cause the CPU to "latch-up" - a breakdown of a normally insulating part of the internal silicon, which will certainly stop the system working and may cause permanent damage to the CPU.

11.1. Ground rules

- *Set your power supply high* : "3.3V" parts are specified with an eye on battery-powered systems, which exhibit poor voltage regulation. They are therefore able to operate correctly on a fairly wide range of voltages, usually quoted as 3V to 3.6V.

  All the input signal problems are related to the amount by which the input signal level exceeds the chip's power supply. So if you set your low-voltage supply near to 3.6V, your design will be more robust.

- *Attend to special cases* : on most CPU types the input clock *MasterClock* has special requirements - and in this case the same signal levels are not acceptable for 5V and 3.3V parts. The 5V CPU requires a rail-to-rail clock.

  You should also be careful about the clock synchronisation feedback signal *SyncIn.*

  We'd recommend you use a nice, crude, potentiometer network to cut these signals down to a 3.3V swing when required.

- *Remember power-up* : a transient voltage difference during powerup (caused by a 3.3V supply which lags the 5V supply, for example) can be big enough to cause the CPU to latch up - and the condition is stable. Unless the supplies come up together, you're going to need to arrange that input signals are not driven until power is stable through the system.

11.2. Limiting input signal transitions

There are a number of ways you can do this:

- *Specialist converting buffer components* : you can obtain a number of different buffer types (particularly variants of the '245 bidirectional bus driver) fitted with dual power inputs and designed to connect 3.3V and 5V signal sets.

  However, as discussed right at the start of this document (in §2. ("R4x00 System Goals") on page 8), the ideal component to attach to R4x00 is a bidirectional edge-triggered register with clock enable inputs. Once you start putting all these requirements together with voltage translation, there are no parts which qualify. Maybe one has appeared since we wrote this[7].

- *Use QuickSwitch parts as voltage limiters* : a more subtle variant of the scheme uses QuickSwitch[8] ("QS") components. These devices are CMOS switches which make a bidirectional connection between two signals with an effective resistance of the order of 5Ω. They don't amplify the signal, and they exhibit a delay so close to zero that you can't measure it. But in this context they have another vital characteristic, which is that they cease to conduct when the input gets with about 0.8-1V of their power supply, effectively clamping the signal as it passes through.

---

[7] If it has, it will probably be in the IDT FCT logic catalogue.

[8] "QuickSwitch" is a trademark of Quality Semiconductor, who deserve a cheer for inventing it.

A QS part fed with about 3.9V as its power supply will clean up your 5V signals and make them usable by the 3.3V CPU. It will work on bidirectional signals (because the clamped high outputs are still well within spec as a TTL compatible logic high). Since they don't have any drive, the QS parts use very little power so can be supplied from 5V through a series resistor (we've used 39Ω to feed 11 QS' in parallel) regulated by a zener diode to ground, and appropriately decoupled.

Not all signals should be so abused; in particular, think about some other way of handling clock inputs to the CPU. But this gets our star recommendation.

- *Use bipolar-level TTL inputs* : with the CPU supply tweaked to near 3.6V, inputs from old-fashioned "74F" TTL parts are very unlikely to overshoot to such an extent as to cause trouble. The same is true of BiCMOS and low-swing CMOS parts (such as IDT's FCT-T series, or Texas BCT).

  Nonetheless, such inputs will regularly dwell at levels 0.5V or more above the CPU's power rail - and overshoots could exceed device limits. We'd recommend that you use this approach only when you need to retrofit 3.3V ability to an existing design - and then be ashamed, or careful, or both.

11.3. Generating 3.3V power from the 5V rail

3.3V R4x00 components do not use huge amounts of 3.3V power - probably 1.5A or less for the R4600/Vr4200 generation of parts. A simple linear regulator circuit can readily supply enough power without getting too hot. It has big advantages; it will bring up the 3.3V power almost simultaneously with the 5V supply (avoiding transient power-up overvoltages), and won't cause noise problems.

We strongly recommend that you use a regulator off the 5V supply on most of your mixed-voltage designs. Of course, if you need to run for as long as possible off batteries you'll need something more efficient.

Appendix A: SysCmd codes used by the R4x00

This is a complete list of command codes which may be uttered by the R4x00. You may find this an easier reference than the bit-based decoding described in §5.3. (“Command encoding”) on page 18.

SysCmd(8:0) *hex*	*bin*	*Operation*
Cached reads		
00	0 000 00 000	read, 4-word cache line refill
01	0 000 00 001	read, 8-word cache line refill
Normal uncached reads		
18	0 000 11 000	read, 1-byte
19	0 000 11 001	read, 2-byte (short)
1B	0 000 11 011	read, 4-byte (word)
1F	0 000 11 111	read, 8-byte (double-word)
Peculiar uncached reads (only from load-left, load-right)		
1A	0 000 11 010	read, 3-byte
1C	0 000 11 100	read, 5-byte
1D	0 000 11 101	read, 6-byte
1E	0 000 11 110	read, 7-byte
Cached data writes		
50	0 010 10 000	write, 4-word cache line writeback
51	0 010 10 001	write, 8-word cache line writeback
Uncached writes		
58	0 010 11 000	write, 1-byte
59	0 010 11 001	write, 2-byte
5B	0 010 11 011	write, 4-byte
5F	0 010 11 111	write, 8-byte
Peculiar uncached writes (only from store left, store right)		
5A	0 010 11 010	write, 3-byte
5C	0 010 11 100	write, 5-byte
5D	0 010 11 101	write, 6-byte
5E	0 010 11 110	write, 7-byte
Null operation		
60	0 011 00 000	null operation, bus back to CPU
Bitfields in data types		
×	1 LRE C0 000	data, where: L - 0 marks last or only data cycle R - 0 for read data, 1 for write E - 1 for data is bad C - (only into CPU) 1 to suppress *SysAD, SysADC* checking
Everyday data codes		
100	1 000 00 000	nonburst or last-in-burst read data
180	1 100 00 000	burst read data, not last
110	1 000 10 000	nonburst data without parity, don’t check it
120	1 001 00 000	nonburst bad data (use to cause a bus error)
140	1 010 00 000	nonburst or last-in-block write data
1C0	1 110 00 000	burst write data, not last

Table A.1: SysCmd encodings

In this table bit 0 comes first in time. In multi-bit codes the value is an unsigned integer made up by taking the bitfield and regarding the lowest-numbered bit as the least significant.

Bit Nos.	*Meaningful Values*	*MIPS Mnemonic*	*What it does*
0	0		Reserved
1-4	0-8	**XmitDatPat**	system data write rate (see Table 10.2)
5-7	0-6	**SysCkRatio**	Pipeline clock/bus clock divisor. 0 = 1:2 up to 6 = 1:8 - *n* means divide by (*n*+2).
8	0/1	**EndBIt**	0 = little-endian, 1 = big-endian
9-10	0-3	**Non-block write**	Single write protocol options (for all writes except cache writeback): 0 = R4000 compatible 1 = reserved, don't use it 2 = pipelined writes, as described in §7.1. ("How it works") on page 29. 3 = write re-issue (not described here, see R4600/Orion documentation)
11	0/1	**TmrIntEn**	Determines role of *Int5** and availability of interrupt from internal timer. 0 = internal timer interrupt enabled, pin is a no-connect 1 = no internal timer interrupts, interrupt through pin.
12	0		Reserved
13-14	0-3	**Drv_Out**	Output driver slew rate control. 2 = fastest 3 = faster 0 = slower 1 = slowest
15-255	0		Reserved

Figure B.1 Configuration data stream encoding for the R4600

Appendix C: Software cache management with DMA

The ingenious design of the large-package R4x00 exploits the tag store of the external secondary cache as a “duplicate tag store” for the primary cache, allowing cache coherency operations to be carried out in parallel with processor execution from the primary cache.

But this means that a R4x00 processor, without the secondary cache, is not capable of implementing the cache coherency operations. This implies that system software must manage the R4x00 caches to ensure that the CPU does not read data from the cache which has been updated in memory by some other memory master.

Because the R4x00 data cache is a writeback type, care has to be taken with both DMA reads and writes:

- *DMA from memory to peripheral* : it is essential that the contents of memory are correctly updated before a DMA operation starts. This means that any data cache contents holding lines in the DMA buffer area must be flushed back to memory before the DMA operation is commenced. The R4x00PC has special coprocessor instructions to flush lines from the cache.
- *DMA from peripheral to memory* : before the operation starts, you should ensure that all data cache lines referring to the memory are flushed (otherwise the processor might trigger back a writeback of a line in the middle of a DMA read of the same data, which would create chaos).

  Either before or after the DMA operation, but before the CPU attempts to reference any of the data which has been copied into memory, you must invalidate any cache lines referring to the buffer to ensure that programs do not read “stale” data.

ASIC: "Application-specific integrated circuit" - a chip custom made for a specific application.

bus, bus interface: when used unqualified in this document, a short way of describing the data transfer interface of the R4x00, which consists of the *SysAD, SysCmd* and associated control signals.

Coherent cache: in a multiprocessor system, coherent caches used by CPUs sharing main memory maintain the fiction that the CPUs are reading and writing main memory directly (so that a change made by one CPU is immediately visible to all others, and all CPUs have the same "coherent" view of shared memory). Ingenious mechanisms have been developed which achieve this while maintaining most of the efficiency and reduced memory traffic which the caches were put in to obtain.

error-correcting code (ECC): a kind of checksum with the characteristic that certain classes of error (for a memory system ECC like the R4x00s this means any single-bit error) produces a code from which the correct data can be re-computed.

even parity: when we say that *SysAD(7:0)* is checked by *SysADC0* using even parity, we mean that the total number of one bits in *SysAD(7:0)* together with *SysADC0* should be even. The bus is checked by computing an exclusive-OR of all 9 bits (the result should be zero); the check bit is calculated as an exclusive-OR of the 8 data bits.

external agent: MIPS-speak for the logic at the other end of *SysAD* and *SysCmd.* We try not to use MIPS-speak too much, since the style of the MIPS documentation appears to be a problem for many readers.

issue cycle: The issue cycle of a request on the R4x00 bus is the *last* bus clock cycle for which the request code is driven on *SysCmd* and *ValidOut** is asserted. If you leave the flow-control signals *WrRdy** and *RdRdy** asserted, then the issue cycle is the one and only clock cycle for which the request is driven.

Everything which happens on the bus after the R4x00 puts out a request will be related to the issue cycle timing.

Like the happiest day of your life, the issue cycle is difficult to recognise without the benefit of hindsight. This makes the concept less useful than it might otherwise be!

output buffer di/dt control mechanism: also referred to as "output drive tuning", "output speed tuning". The mechanism by which the R4x00 adjusts the current limit of its output drivers until their transition timing, as represented by the dummy loopback load from *IOOut* to *IOIn,* safely meets its configured target speed. This is referred to several times but the best description is in §10.2.5.

phase locked loop: (PLL) a circuit (usually analogue in nature) which regenerates a clock from any regularly changing input, such that the recovered clock's frequency and phase bear some predefined relationship to the input transitions. This is most commonly met with in data recovery circuits (disc and communication interfaces, for example) where the objective is to produce a beautiful clock from ugly inputs (by averaging out the ugliness over time).

In the R4x00, the PLLs serve two different functions:

- speed changing: used to phase-lock clocks running at multiple or sub-multiple frequencies;
- delay compensation: used to adjust the CPU's output clocks so that, after the user's buffering and tracking, the user's clocks are precisely aligned with the CPU's internal phases.

Pipeline: for a CPU, the pipeline is a technique for speeding instruction execution. Instructions are fetched and decoded much more rapidly than they can be executed (in the R4x00 one instruction is fetched on each internal clock, double the input clock rate). Each instruction is then passed through a set of phases (one per clock); the different bits and pieces making up the CPU have to be carefully

designed to prevent an instruction using something that an earlier instruction hasn't actually done yet.

programmable logic device (PLD): PALs and the larger devices usually called FPGAs. High speed is essential. Some FPGAs have *input* register stages which are extremely fast, so may suffer from hold time problems in an R4x00 application.

read requests: MIPS-speak for the read command on *SysCmd* with the address of the location on *SysAD.*

response: MIPS-speak for a set of data returned to the CPU on a read on *SysAD* together with a suitable type code on *SysCmd.*

sampling window: the period of time during which a signal must be in a valid state in order to guarantee that it is correctly sampled. For a signal sampled by an edge-triggered register, the window extends from the earliest clock time less the setup time, to the latest clock time plus the hold time.

secondary cache: caches can be cascaded, with a small fast cache close to the CPU refilled from a larger, slower cache (the secondary cache) refilled from large, slower-still main memory. The R4x00 family is conceived as using a dedicated off-chip secondary cache to backup the on-chip caches (which with 1992 technology are not nearly big enough). The hundreds of extra pins needed to wire up the secondary cache are omitted on the R4x00 version.

shared memory multiprocessor: a multiple-processor machine whose CPUs all share the same memory space and where the memory is a reliable vehicle for inter-CPU communication. The reliable shared memory is usually a fiction carefully maintained by coherent caches.

split transactions: a bus transaction (which is usually a read) where the bus is acquired once to broadcast the address, then potentially used for other purposes before being re-acquired to return the data. Where a bus is much faster than attached memory and is intensively used by multiple masters, split transaction protocols considerably improve total bus throughput.

write request: MIPS-speak for the message which concatenates write data phase(s) onto the write command and address.

Appendix E: Analogue design considerations

R4x00 designs will be fast and wide. The CPU technology (and the only sensible choice for the data and address paths) is high-speed TTL-compatible CMOS - high performance but tricky stuff to use.

A folklore has grown up about how to use this stuff successfully. Different problems are summarised below. Note however that where this comes up and bites you it will often be because two of the problems have joined forces by chance.

High speed CMOS design: edge rates, ground bounce

High speed TTL-compatible CMOS logic parts have very high drive, very high output slew rates, and very high input impedance. Some of the problems associated with their use have been widely discussed - ground bounce is a good example. But other effects can be at least equally bad and worthy of consideration. You should beware:

- *Fast edges* : FCT and similar parts have sub-ns rail-to-rail voltage slews; the octal parts most often used in bus applications are specified to drive up to 48mA and don't slow down much when loaded. Most signals will be predominantly loaded by other CMOS parts, so the load will be almost pure capacitance.

  The fast edges lead to substantial undershoot. The undershoot will be to some extent damped by FCT-type inputs (which have high speed clamp diodes) but more complex CMOS parts don't necessarily respond fast enough.

  This problem is eased by the increasing availability of parts whose nominal logic 1 output level is around 3.5V rather than 5V: suitable parts are "FCT-T", or the BiCMOS "ABT" logic family. A series damping resistor (typically 22-47Ω) situated close to the driver reduces the instantaneous current flow and is usually helpful. You can now buy components with integrated resistors, which saves space.

- *Instability and oscillation* : the combination of high drive and low input impedance means that high speed CMOS logic shows much greater power gain than bipolar TTL. It requires very little output→input coupling to promote instability; in particular an input floating around the logic transition level, and which directly controls an output may well oscillate with enough force to disrupt the operation of connected components[9].

  High-drive "transparent" parts should not be enabled until the inputs are at a good level.

- *Ground bounce* : a component driving several outputs may have so low a source impedance that the impedance of the ground trace to the chip, and the ground connections from the component leg to the die, are a significant proportion of the whole loop. Where several outputs make a high to low transition, the die ground voltage will "bounce" up relative to system ground.

  Once the bounce exceeds the normal noise thresholds strange things start happening. For example other output signals, nominally at logic "0", will be dragged up with the die ground and may be seen to pulse high.

  Ground bounce is at its worst with packages where the connection from ground pin to die is longest; traditional corner-ground through-hole DIL packages are particularly bad. You get much less trouble on SMT components; but on your SMT board the pad cannot be attached directly to the PCB ground plane, so the problem may merely shift from being a component problem to a board problem.

  Newer pin-outs usually incorporate extra ground pins to solve this problem; the 16-bit wide packaging format pioneered by Texas Instruments as "WideBus" is a good example. And your series resistor helps again by reducing the output current.

[9] Do not make the mistake of thinking that high-resistance pull-up resistors on tristate lines will get round this problem. Such arrangements still leave the input close to the transition level for many ns as the resistor pulls up a signal disabled from a logic low.

Reflections and other long-track effects

Transitions propagate along copper PCB tracks at a rate of about 15cm/ns. Every component connection is an impedance discontinuity, and CMOS components' high DC impedance absorbs very little of the transition energy, so at these discontinuities some of the "wave" is reflected back. The result is a period of noise which is usually reckoned to die down in approximately twice the time required to traverse the longest path out from the driver.

Some books and articles will talk of transmission lines and incident waves, and use models assuming a single driver, a track of known and constant impedance, and a calculated-impedance termination network to show how to suppress the noise. You should read these books if you must, for example, ship a fast clock 45cm across a board or backplane; but for TTL-compatible logic with multiple loads you should leave your calculator gathering dust at the back of the desk.

However, the following helps:

- *Keep tracks short* : particularly on signals whose edges are used (clocks, strobes etc). Take care if your CAE system can't tell you how long traces are. Do remember, though, that it is maximum length from driver which counts, not total length; these can be very different for a signal tracked as a "tree". What length is safe? 15cm is no problem and you'll usually get away with 20cm.

  Where manufacturer's design books start telling you that tracks must be less than 10cm, worry. Such limits are not, in practice, achievable; and so should usually be interpreted as meaning that the author doesn't know how to make the system work, and is just making sure that s/he can't be blamed!

- *Watch out for high capacitative loads* : on DC considerations alone a 48mA CMOS driver can easily drive several thousand CMOS inputs. Be sensible; 10 loads is already a lot. But if you break this rule the ground bounce will probably find you first.

- *Series resistors (again)* : series resistors usually work quite well with long-track effects too. Note, however, that a calculation from transmission line theory will ususally suggest that the best damping is obtained with a resistor in the range 68-100Ω; but this is usually excessive and slows the edges too much.

  It is probably true that resistors put in by engineers who intend to tackle signal reflections turn out to useful because of the other benefits of series resistors mentioned above. This of course lends spurious empirical backing to the idea that transmission-line theory is a generally useful tool.

Power supply noise and special decoupling

R4x00 systems will be built on boards using ground planes with adequate HF and LF decouplers. However, the R4x00 uses separate pins to get power for the analogue phase-locked loop circuits, and the supply to these pins is extremely sensitive to small amounts of HF noise. The MIPS recommendation is:

- *VCC and ground* : the PLL supply pins *VccP* and *VssP* should be connected to the corresponding board power supplies with appropriate impedance. Early-life suggestions are for 5Ω resistors, with separate decouplers close to the PLL supply pins; an inductor such as a wire link with a ferrite bead slipped around it would be good.

- *Extravagant decoupling* : three decouplers allow better coverage of the frequency range. For a spread use a 10μF tantalum, a 100nF multilayer HF ceramic, and a 1nF low-inductance ceramic. And keep them close to the pins, and use nice fat tracks.

That way you'll never know whether it mattered.

Appendix F: Thermal considerations

R4x00 parts get hot; R4400 components at high speeds may dissipate more than 7W. To operate in free air at room temperature you will need a black alloy heat sink bigger than the chip (if anyone shows you an R4400 system without a heat sink on the CPU you can be sure that it doesn't work and has probably never been switched on!).

The later, "embedded", variants run much cooler, particularly at 3.3V. 1994/95 3.3V Vr4200 (and possibly R4600) parts may reach the point where convection cooling can be made to work with appropriate product packaging.

The gold square central section of the PGA or MQFP package is thermally bonded to the die and is well placed to make a good connection to your heat sink[10].

To be able to run over a normal commercial range some kind of induced airflow is essential. Once you have an airflow established a more modest heat sink will do.

One ready-made solution is the combined fan/heatsink modules fitted to every Pentium or fast-i486 PC. They're available for both 12V and 5V supply, but 12V is cheaper and easier to get. Highly recommended for small-volume and informal applications.

You may be able to get clip-on combined heat-sink and electromagnetic shield components.

[10] Beware the small capacitor components on top of early chips; they are precision PLL capacitors and if you short them out you short *VccP* to *VssP*.

References

R4x00 Processor Interface: *MIPS R40 00 Processor Interface*, original author MIPS Computer Systems Inc. Information here is based on MIPS' revision 2.9.1 dated 7th May 1992.

R4400 Users Manual: *MIPS R4400 Microprocessor User's Manual"*, author MIPS Computer Systems Inc (1991).

BRClock....3(11)
BTClock....3(11)
Cas*....6.2(27)
ColdReset*....**12**, 10.3(47), 10.4(47), 3(12)
ExtRqst*....**10**, 3(10), 9.1(38), 9.2(39), 9(37)
Fault*....**12**
GrpRun*....**12**
GrpStall*....**12**
IOIn....**11**, 10.1(43), 10.2(45), 3(11), B(53)
IOOut....**11**, 10.1(43), 10.2(45), 3(11), B(53)
Int(5:0)*....**10**, 10.1(41), 9.3(39), B(51)
JTCK....**11**
JTDI....**11**
JTDO....**11**
JTMS....**11**
MasterClock....**11**, 10.2(44), 10.3(46), 10.4(47), 11.1(48), 3(12), 4.1(14), 4.4(15), 4(13)
MasterOut....**11**, 10.3(47), 3(11), 4.1(14)
ModeClock....**12**, 10.1(41), 10.3(47), 3(12)
ModeIn....**12**, 10.1(41), 3(12)
NMI*....**11**
PClock....4(13)
RClock(1:0)....**11**, 10.2(44), 2(9), 3(11), 4.1(14), 4.4(16), 4(13), 6.2(27), 8.1(32)
Ras*....6.2(27)
RdRdy*....**10**, 3(10), 5.4(21), 6.1(24), 6.2(27), 9.1(38), B(53)
Release*....**10**, 3(10), 6.1(24), 9.1(38), 9.2(39), 9(37)
Reset*....**12**, 10.3(47), 3(12)
SClock....4.2(14), 4.3(15), 4(13)
SyncIn....**11**, 11.1(48), 3(11), 4.1(14), 4.2(14), 4(13)
SyncOut....**11**, 3(11), 4.1(14), 4(13)
SysAD(63:0)....**10**, 1.1(7), 10.2(44), 3(10), 5.2(18), 5.3(18), 5.4(22), 6.1(24), 6.2(27), 8.1(32), 8.2(34), 9.1(38), 9.2(39), 9.3(40), 9(37), A(50), B(53)
SysADC(7:0)....**10**, 5.2(18), 5.3(20), A(50), B(53)
SysCmd(8:0)....**10**, 3(10), 5.2(18), 5.3(18), 5.5(23), 6.1(24), 6.3(27), 7.3(30), 9.1(38), 9.2(39), 9(37), B(53)
SysCmdP....**10**, 3(10), 5.2(18)
TClock(1:0)....**11**, 10.2(44), 2(9), 3(11), 4.1(14), 4.2(14), 4.3(15), 4.4(15), 4(13), 6.2(27), 8.1(31)
VCCOk....**12**, 10.1(41), 10.3(46), 10.4(47), 3(11)
ValidIn*....**10**, 5.2(18), 5.5(23), 5(17), 6.1(25), 9.2(39), 9(37)
ValidOut*....**10**, 10.1(42), 5.2(18), 5.4(21), 5.5(23), 5(17), 6.1(24), 6.2(27), 7.1(29), B(53)
VccP....**12**, E(56), F(57)
VssP....**12**, E(56), F(57)
WrRdy*....**10**, 3(10), 5.4(21), 7.1(29), 9.1(38), B(53)