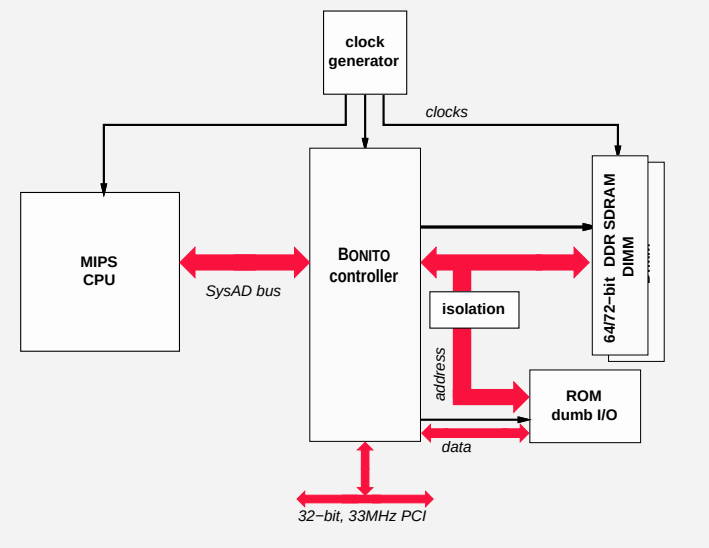
BONITO64 - “north bridge” controller for 64-bit MIPS CPUs

Rev 1.4 of 2001/07/18

- Synthesisable reusable Verilog design. This document describes the functions realised in a Xilinx “Virtex-E” FPGA, but they have also been proven as a 32-bit ASIC.
- Direct connection to MIPS CPU using any known variant of the “SysAD” interface. FPGA implementations of BONITO64 run at 100MHz or higher.
- Direct connection to 32-bit 33MHz PCI bus, conforming to Rev2.1. Independent CPU and PCI input clocks. PCI arbiter and other “host” functions available, but can also operate as a peripheral. Includes PCI mailbox registers for intelligent peripheral communication.
- High-performance synchronous DRAM memory system using one or two industry-standard modules or equivalent onboard circuit. External clock generator required for SDRAM clocks.
- Local bus for ROM and simple I/O connects “dumb” components, isolated from high-speed signals by external CMOS switch. I/O bus DMA supports “UDMA” transfers on an attached IDE disk drive, as defined in the ATA-4 standard for PC disk drives.
- Unique “I/O buffer cache” (IOBC) caches local memory locations to enhance PCI transfer performance for device controllers which are PCI bus initiators. The IOBC caches are kept coherent with main memory by the hardware, so the IOBC is invisible to run-time software.
- Configurable debug mode makes all cycles visible at a DIMM socket.
- Glueless support of CPU reset sequence.
- Powerful, simple interrupt controller and GPIO pins.
- System can be initialised and the MIPS CPU run from local ROM or over PCI. Configuration options which can’t be in software use pullups on data buses.



BONITO64 in a minimal system

Contents

Contents 2

1. Overview and Introduction 5
Manual road map 5
Conventions 5

2. Interfaces 6
2.1. CPU Port 6
2.2. PCI port 6
2.3. SDRAM connection 7
2.4. ROM and local I/O port 8
2.5. Interrupts and general-purpose I/O (GPIO) pins 9
2.6. Test interfaces 9
2.7. Debug/diagnostic facilities 9

3. Inside BONITO64 - they I/O buffer cache and copier 10
3.1. The I/O buffer cache 10
Software management 11
3.2. PCI copier 11

4. BONITO64 system information 13
Table 4.1: CPU/local bus address map 13
4.1. Address maps 13
CPU access map 13
From-PCI map 13
Figure 4.1 Memory regions and mappings between local and PCI space 13
4.2. Endianness and bus-to-bus transfers 15
BONITO64 and PCI with a big-endian CPU 15
Endianness and the SDRAM port - nothing to do 16
Endianness and ROM cycles 16
Figure 4.2 ROM data for the CPU - ROM addresses and byte lanes 16
Table 4.2: CPU reads and resulting ROM cycles 17
Endianness and local I/O accesses 17
Endianness and the I/O bus 18
Endianness and PCI transfers, CPU and bus-initiated 18
4.3. Reset-time options and “mode bits” 18
4.4. Where to boot from? Two ROMs or PCI-initiated bootstrap 19
4.5. Reprogramming BONITO64’s PCI identity 19

5. Software-accessible registers and programming 21
5.1. Register Summary 21
Table 5.1: All registers 22
General principles for BONITO64 registers (read this) 22
5.2. Power-on settable configuration register - **`bonponcfg`** 23
Table 5.2: Fields in **`bonponcfg`** 25
5.3. Configuration bits with power-on defaults - **`bongencfg`** 25
Table 5.3: Fields in **`bongencfg`** 26
5.4. Local I/O configuration 27

Table 5.4: Fields in **iodevcfg** 28
5.5. SDRAM configuration 29
Table 5.5: Modules, sides and chip selects 29
Table 5.6: Fields in **sdcfg** 31
5.6. BONITO64 registers available in PCI configuration space 33
Table 5.7: Standard PCI configuration space registers 33
Table 5.8: PCI configuration space base address register 34
Figure 5.1 Fields in **pcimembasecfg** 34
Table 5.9: Fields in **pcicmd** *- PCI configuration space "Command" register* 35
Table 5.10: PCI configuration space "Status" register 36
5.7. CPU access to PCI 37
Table 5.11: Fields in **pcimap** 37
Figure 5.2 Fields in **pcimstat** *- PCI master status register* 38
Figure 5.3 PCI address-time values for configuration cycles 39
Figure 5.4 Fields in **pcimap_cfg** 39
5.8. The PCI copier 40
Copier interrupts/status 40
Copier address registers 40
Copier control/status registers 40
Figure 5.5 Fields in **copctrl**, **copstat**, **copgo** 40
5.9. PCI access to SDRAM and IOBC programming 41
5.10. Registers for IOBC management 41
Table 5.12: PCI Cache control register 42
Table 5.13: Fields in the pcicachetag register 42
5.11. PCI access to local I/O 43
5.12. Local I/O DMA control 43
Figure 5.6 ldmastat and ldmaCtrl register layouts 43
5.13. PCI Mailbox registers 44
5.14. GPIO pins 44
Figure 5.7 Fields in **gpiodata** *and* **gpioie** 44
5.15. Interrupt control 45
Table 5.14: Fields in **intXXX** *registers* 45
Initialising the interrupt controller 46
5.16. BONITO64 reference timer 46

6. Hardware description 48
6.1. Signals 48
6.2. Pinout 53
Figure 6.1 676-pin FPGA pinout for BONITO64 53
Signal name abbreviations used in pinout table 54
6.3. Package information 54

Appendix A: Register Addresses 55

Appendix B: BONITO64's debug interface 56
How debug cycles work 56
Algorithmics' DIMM debug board 56
Figure B.1 57
Table B.1: Signals available from debug board 57
Connector pin-outs 57

Figure B.2 Connecting an HP or compatible analyser to debug board 58
Figure B.3 Pin-by-pin analyser connection to debug board 58

Appendix C: - IDE interface and UDMA transfers 60
An introduction to UDMA 60
UDMA cable termination guidelines 60
Table C.1: UDMA signal termination recommendations 60

Appendix D: - Hardware timing and the programmer 62

Appendix E: - software-visible changes from the BONITO ASIC 64
SDRAM Configuration 64
CPU clock rate independence 64
Timer 64

1. Overview and Introduction

This device has four ports: the CPU's SysAD, PCI, SDRAM and local ROM & I/O. In some packaging options some SDRAM signals are used during some local ROM & I/O cycles to provide more addresses. A minimal system block diagram is shown in the figure on page 1.

The CPU-side and PCI clocks are independent inputs, and need have no timing relationship. The SDRAM system is operated synchronously to the CPU-side clock and should be fed (with the CPU and BONITO64) with matched low-skew clock inputs.

BONITO64 handles tasks which are common to many of those systems where a MIPS CPU is used with a PCI bus and a local memory system, including:

- Managing the CPU reset sequence and supporting common configuration options.
- System bootstrap from local ROM/flash or over PCI.
- A flexible and high-performance interrupt controller, see §5.15.
- A few general-purpose I/O pins - sufficient, for example, to provide access to the signature EEROM of a memory DIMM.

BONITO64 itself does not include any specific I/O device - not even a UART. It's cheap and easy to add such devices on the I/O bus.

Compared to other MIPS system controllers, BONITO64 offers a simple, fast, memory controller, outstanding real-life PCI performance, and a high level of integration to minimise glue logic.

BONITO64 is available for license from Algorithmics as reusable Verilog code.

Manual road map

The sections are like this

- *Interfaces* : goes through each port in turn describing what they do
- *Inside BONITO64* : describes the *I/O buffer cache* (used for PCI master accesses into local memory) and the *PCI copier* (used to move data from local memory to other PCI locations).
- *BONITO64 system information* : memory maps, endianness, bootstrapping and PCI options
- *Software-accessible registers and programming* : all the registers and programming how-tos.
- *Hardware description* : signals, pinout and packaging.
- *Appendices* : A: register address summary; B: debug interface and the debug board; C: IDE interface design information; D: hardware timing and the programmer

Conventions

Software register names are written in **`bold-monospace`**; register names have no spaces and are not meant to be case-sensitive. Signals are written in *SmallItalics* and active-low signals are indicated by an asterisk suffix: *ActiveLow**. Web addresses (URLs) are shown as `monospace`.

2. Interfaces

One way to introduce this multi-faceted part is to go around its interfaces and describe each in turn. So we'll do that for the CPU, PCI, SDRAM, ROM and I/O, GPIO, test and debug interfaces.

2.1. CPU Port

BONITO64 connects directly to the MIPS "SysAD" bus. Parity is passed through to the local SDRAM system, and generated/checked on other cycles; but parity checking can be disabled if you'd rather use 64-bit (not 72-bit) memory.

BONITO64 manages the CPU's reset sequence. A complete reset of BONITO64 from its *SYSRESET** pin - or the PCI *RESET** signal where BONITO64 is configured as a peripheral - causes the CPU to be driven through a cold-reset. It's also possible to do a cold-reset of the MIPS CPU by writing a register bit; this can be used as a self-reset by CPUs which need to put themselves in a non-standard configuration: see §4.3 for more information.

In some cases a system may be managed by a host across PCI; it's possible to configure BONITO64 to wait with the local CPU stalled while an external PCI host configures the chip and uploads software to local memory; see §4.4.

BONITO64 has an internal interrupt controller which connects to two of the MIPS CPU interrupt inputs.

2.2. PCI port

BONITO64 conforms to the PCI specification (rev 2.1), can act as initiator or target on a PCI bus, and when required can perform all host roles other than clock generation - it has a PCI arbiter onboard, can source the PCI reset signal, and can initiate configuration cycles.

The CPU can directly read or write PCI space. CPU partial-word read or writes to PCI space are signaled with exactly the byte enables you programmed. The byte enables (and byte lanes) used can surprise you when your CPU is "big-endian"; see §4.2. CPU burst accesses to PCI are supported to the extent necessary to run any software uncached (some 32-bit CPUs use bursts to implement uncached 64-bit loads/stores), but software needs to be aware that PCI accesses may fail, and monitor for bus errors (on reads) and PCI errors reported by interrupts and BONITO64 registers (on writes).

PCI "configuration space" provides special address ranges used to initialise devices. BONITO64 can configure devices on the local PCI bus (type 1 cycles) and also devices connected via a PCI bridge (which need "type 2" configuration cycles). Configuration cycles are programmed through a pair of registers (one sets up the PCI address bits, and the other is a data register).

CPU writes to PCI are "posted"; there's a BONITO64 register **`pcimstat`** which software can read to check when all posted writes have been completed on PCI[1].

CPU reads from PCI space may be quite slow. Although BONITO64 handles the reads in sequence, CPUs advertised as supporting "non-blocking reads" or "multiple outstanding requests" can continue to run while waiting for data. BONITO64 will continue to provide the CPU with access to local DRAM and I/O space. But PCI operations, however delayed, are always completed in the order in which they were submitted.

BONITO64's PCI arbiter handles up to six external initiators (seven bus masters including BONITO64 itself) and operates in round-robin only. A configuration-time option allows the arbiter to be disabled, for systems whose arbiter is elsewhere; in this mode two of the arbiter signals are reconfigured to become BONITO64's own request/grant lines.

---

[1] Posted writes can cause software porting problems: see Appendix D for a discussion of problems and solutions.

BONITO64 makes some or all its local DRAM memory available to external PCI bus masters. PCI initiator/local memory transfers go through the "*I/O buffer cache*" (*IOBC*), described in §3.1 below, to maintain high throughput with minimal impact on the CPU's access to local memory.

BONITO64 provides all the mandatory standard PCI-visible configuration registers. But software running on the local MIPS CPU can modify some of these register contents and arrange that the registers are not visible until they have been changed, so that BONITO64 can take on a different identity as part of an intelligent PCI controller; see §4.5.

BONITO64 can be configured to respond as a PCI target to three PCI address regions. One is hard-wired for BONITO64's own PCI-accessible registers, and two are used for access to attached local SDRAM or I/O devices - the size, location and transfer characteristics of these two windows are configurable. PCI access to the local I/O bus is provided for diagnostics and system initialisation; no guarantees are made about performance.

BONITO64 implements mailbox registers. A PCI write to a mailbox register generates an interrupt condition for the CPU, which can retrieve the data through a locally-readable register.

2.3. SDRAM connection

The memory interface is optimised to run burst-mode SDRAM cycles in CPU native word order (sub-block or linear) using industry-standard modules. It is synchronised in frequency and phase with the CPU interface to minimise CPU access time. It can drive systems with a fan-out of up to 8 loads on each DRAM control output. Larger systems require high-drive registered buffers (74ALVCH16374 or similar) for the multiplexed address lines and address-time control signals (which go to every DRAM device). When the registered buffers are used, you must configure the SDRAM controller to delay all data timings by one clock.

In many package options the SDRAM data bus is also used to carry addresses for I/O accesses requiring more than a few address bits; in particular, for ROM cycles. BONITO64's *Isolate* signal can be used to control a bus-switch device for this purpose - see the box called "isolation" in the figure on page 1. Where this trick is used ROM and other I/O cycles using the SDRAM data bus cannot - of course - be run concurrently with an SDRAM access; but most I/O device accesses need only the hard-wired addresses.

There's no support for DRAM "open pages"; the cache refill and write-back traffic from the CPU has little locality of reference, and PCI traffic will be ferociously interleaved with CPU references. In this environment open pages cannot be expected to make a big impact on performance - but they make the SDRAM controller much more complex.

The SDRAM controller is optimised for 8×doubleword burst cycles (here and throughout this manual a "word" is 32 bits and a "doubleword" is 64 bits). The controller also runs single 64-bit reads or writes. Memory systems supporting parity generally don't directly support writing only some bytes of the SDRAM array, so BONITO64 implements writes smaller than a doubleword with a read-merge-write sequence - not visible to software but relatively slow. Programmers are not expected to use uncached accesses to SDRAM except for bootstrapping and diagnostics.

The SDRAM control signals are designed to attach to a 64/72-bit array. Parity generation and checking are supported so long as (as is usual) the CPU bus is the same width as the DRAM bus.

PCI traffic has unconditional priority over CPU traffic for SDRAM, but PCI traffic is slow enough relative to the memory system that there are always plenty of cycles left over for the CPU.

BONITO64 performs DRAM refresh cycles in pairs at regular intervals - optimally about 15μs. Like many other timings in BONITO64 the refresh interval can be set accurately only if you set up the pre-scaler **`iodevcfg.cpuclockperiod`** as described on page 27.

You can configure BONITO64 to support synchronous PC-100 SDRAMs or PC-200 DDR SDRAMs. It will support devices with up to 13 row and/or column addresses. Configuration is software driven, and is

detailed in §5.5 on page 29. If your system uses DIMMs which include the industry-standard “self-portrait” EEROM, then system software can read the on-DIMM configuration EEROM using two GPIO pins, and use that to figure out how to set up the memory controller.

2.4. ROM and local I/O port

A dedicated local 16-bit I/O data bus is provided, with five dedicated I/O bus address bits. I/O bus cycles requiring more addresses find them multiplexed (according to how BONITO64 is configured at build time) either on the I/O data bus lines at the start of the I/O cycle, or on the SDRAM data bus.

When the addresses are provided on the I/O data bus you need a latch to capture and drive the addresses.

When the SDRAM data bus is used, it’s desirable to isolate the I/O and ROM devices from the high-speed SDRAM cycles; so a signal *Isolate* is provided to control a bus switch or simple buffer on the address lines. It is always left disabled during SDRAM cycles.

BONITO64 generates Intel-style I/O device control signals, two ROM chip selects, and four I/O chip selects. ROMs must be 160ns or faster. Two fixed I/O timings are supported, corresponding to read/write pulse widths of approximately 200 and 600ns, respectively. BONITO64 does not aim to support byte addressing for I/O devices; 8-bit controllers should be wired to data bus bits *IOD0−7* and should get addresses from *IOA2* upwards. That means that device registers will be at 4-byte intervals in CPU or PCI space.

BONITO64 will perform multiple reads from a ROM to service CPU word or burst reads, allowing the system to run - and to run cached - from a single 8- or 16-bit device. The order with which ROM data is returned in a burst needs to match the CPU’s preferred burst order, which can be either “sequential” (MGB Link CPUs) or “sub-block” (*SysAD* CPUs), under the control of the **`bonponcfg.burstorder`** register bit - see Table 5.2 below.

The assembly of ROM bytes or half-words into 64-bit words on the CPU bus is done in a peculiar way which is not simply related to the “endianness” of the CPU; see §4.2 for an explanation.

Multi-cycle ROM transfers may be interrupted by SDRAM cycles, so that a CPU running from ROM does not cause unacceptable PCI access delays.

2.4.1. “DMA” for local I/O

This function may not be needed, or provided, by all Bonito implementations.

BONITO64’s local I/O bus is, quite deliberately, highly compatible with the “ISA” bus found inside all PCs. In turn that means that it is also very similar to the “IDE” disk drive interface (which started life as “ISA on a cable”).

IDE peripherals are very cheap and widely available, and IDE disk drives offer very high performance where DMA is available. Recent standards like “ATA-4” have defined a series of improved DMA protocols, and BONITO64 implements several up to and including “UDMA” (“Ultra-DMA”) with its 33-100Mbytes/s burst transfer rates.

BONITO64’s DMA accelerator automatically cycles the local bus to read or write data in bursts of up to 32 bytes of data between a local bus DMA device and an on-chip DMA buffer. The DMA buffer can be configured to auto-flush to or auto-fill from an incrementing local memory address to provide classic DMA.

Accelerator cycles on the I/O bus are requested with a *DMARQ* input and select the port to read/write with a *DMACK** signal.

DMA I/O bus cycles don’t share any DRAM bus signals, so they can be overlapped with SDRAM accesses.

2.5. Interrupts and general-purpose I/O (GPIO) pins

BONITO64 provides nine GPIO pins, programmable as input, output or tristate, and six dedicated input pins. The input pins are particularly good places to wire device interrupts, but some bidirectional pins are available to the interrupt controller too, as described in 5.15.

See the signal list for hardware connections.

2.6. Test interfaces

The chip has a JTAG interface for boundary scan testing.

2.7. Debug/diagnostic facilities

When bringing up software or fault-finding in an embedded system it can be very valuable to be able to follow CPU, PCI and other transfers on a logic analyser.

BONITO64 is designed so that a “debug board” plugged into a DIMM socket can see the address and data of any cycle in the system. The use of the SDRAM bus for this has some effect on performance, so debug tracing can be disabled with a configuration bit.

The debug board requires onboard logic - registers to capture address and data from the SDRAM pins, and logic to interpret the SDRAM-like protocol and generate address/data triggers. It’s even possible to build I/O devices onto the debug board, and have them accessible in the CPU’s memory map.

Electrically, the debug connector is similar to the DIMM module it supplants.

Debug signaling and the debug board is described in Appendix B.

3. Inside BONITO64 - they I/O buffer cache and copier

Two significant pieces of hardware are not directly related to any one of BONITO64's ports, so we haven't mentioned them yet. They're:

- the *I/O buffer cache* ("IOBC"), which is responsible for providing a temporary home for data in transit between local memory and an external PCI initiator;
- The *PCI Copier*, which provides a way to transfer large blocks of data between PCI and local memory with only the occasional involvement of the CPU.

3.1. The I/O buffer cache

> July 2001: a major redesign of the IOBC coming soon will change this software interface substantially, and make software-driven invalidation and writebacks unnecessary. More ambitiously, it will work together with CPUs offering hardware cache coherence to produce a completely software-transparent cache system.

The IOBC is a small internal cache of local SDRAM locations, but it does a job you'd more often see given to a FIFO. It is used when external PCI initiators read or write BONITO64's local SDRAM memory; most PCI controllers are "bus masters" and this is a critical part of many system workloads.

The IOBC translates the PCI's stream-like data accesses to the aligned cache block transfers supported by the local memory. It provides low-latency reads and writes for non-CPU transfers, and causes minimal disruption to the performance-critical CPU/memory traffic. The IOBC uses heuristics to schedule prefetches and write-backs to keep locally-sequential data flowing efficiently.

The IOBC has the following characteristics:

- There are only four lines, each with its own address tag and set of state flags.
- Each line holds two 8×64-bit buffers for data, which are used in "ping-pong" fashion so one buffer can be filling from or writing back to memory while the other is involved in a PCI transfer.
- All IOBC memory fills and writebacks are done as 8×64-bit bursts.

Although the cache is small - absurdly small, by CPU cache standards - it performs well (with a low "miss rate") because PCI accesses show very strong locality of reference.

The IOBC is a *write-back* cache: data written from PCI into the cache is not simultaneously forwarded to SDRAM, but retained in the cache for a while.

CPU write-back caches are usually *write-allocate* - when the CPU writes data to a location which is not currently in the cache, the cache-line is first read in from memory and the data written on top to create a complete and perfect copy of the data.

But the IOBC is not write-allocate; the IOBC never reads data from memory to service a write. Instead, the cache keeps track of every byte which has been written by the PCI side; when the IOBC line data is eventually written back to memory, the IOBC performs a read-merge-write operation if the line is incomplete.

The IOBC relies on two heuristics to improve throughput:

- *Read-ahead*: whenever a PCI master reads local memory (loading an 8×64-bit block of data into the IOBC) the IOBC will automatically schedule a read of the following block of memory into the other half of the cache line. For sequential transfers, this means that the memory access and the PCI transfer operate concurrently.
- *Write-behind*: when a PCI master writes local memory the IOBC will automatically schedule a write-back of a line when it detects a PCI write to an address in the immediately following block of memory. This allows a sequential transfer to be handled efficiently while just occupying one line of the IOBC.

Software management

Uncached MIPS CPU reads and writes to memory are “snooped” by the IOBC controller to ensure that the IOBC and local memory remain coherent.

But the snooping has a performance impact, so it isn’t done for cached accesses. As a result the IOBC’s operation is visible to drivers for PCI “bus master” devices, and the driver must sometimes intervene to prevent data being lost.

Software management is only required for I/O buffers which are cacheable to the MIPS CPU, where the driver must also intervene to writeback and/or invalidate MIPS cache contents.

The IOBC management required is this:

- After a PCI bus master writes into local memory, any IOBC line holding data from the transfer must be written back to memory;
- Before a PCI bus master reads from local memory, any IOBC line holding an old copy of data from the buffer must be invalidated.

The IOBC provides means for software to discover what data is in which line, and to invalidate or write-back any line. See §5.10. (“Registers for IOBC management”) on page 41 for programming details.

3.2. PCI copier

Some applications need to copy large chunks of memory between local and PCI memory. This process can be carried out by the CPU, but the high latency and burst semantics of PCI make this inefficient, so it’s useful to have an automated engine which can carry out the copy and interrupt the CPU on completion. Such facilities are often described as “DMA” but we prefer to reserve that acronym to describe transfers which are device-triggered.

Memory-to-memory copies are problematic on the PCI bus, because large memory-to-memory copies tend to absorb all available PCI bandwidth, increasing worst-case latency for all other bus users. BONITO64 tries to avoid this by using relatively short bursts for the copier, and giving the copier the lowest priority of any onchip process competing for the PCI or SDRAM. The short bursts are cache-line sized, which simplifies the memory controller too.

To initiate a copy the CPU writes the (word-aligned) PCI address, the cache-block-aligned local SDRAM address and a block count[2]. Flags determine the direction of the transfer and whether an interrupt should be raised on completion of the transfer.

Copy requests are “double-buffered”, so the CPU can set up a second transfer immediately it has started the first. When the first finishes, the second will start immediately - and software can arrange to get an interrupt to warn it to set up a third transfer and so on, to keep data flowing.

Once activated the copier transfers cache-line-sized lumps of data between the PCI and local memory, until the count is exhausted. The block count is limited to 16 bits, corresponding to a 2Mbyte copy; larger transfers must be made of a chain of smaller units.

The copier only bursts to/from PCI for data which fits inside a 32-byte memory “cache block”. At block boundaries it drops the PCI bus request for at least one clock to permit other PCI initiators, or other activities within BONITO64, to gain the bus, thus reducing its impact on system-wide PCI transfer latency.

When a copy operation is completed (either its count has hit zero or there has been some non-retryable bus problem) an interrupt will be raised if the programmer asked for it. When a cycle has an error the copier always raises an interrupt and stops itself.

---

[2] If the data you want to copy is not suitably aligned in local memory, you will need to copy the first few words using CPU reads/writes.

The CPU can stop the block copier under software control (though any committed PCI access is completed first). When stopped, internal registers become accessible to the CPU: they include the current PCI address, the current local memory address, the remaining count, and a flags word describing the outcome of the last PCI cycle.

4. BONITO64 system information

We're nearly at the point of defining registers; but there are three general issues to cover first. One is the address map, which is to some extent forced on any BONITO64 system; the second is the confusing issue of endianness. The third brings together issues relevant to getting the system bootstrapped, at least to the point where you're running some software on the MIPS CPU.

There are hexadecimal addresses listed in the tables below, and register addresses in Appendix A; but don't re-type them! We'd like to encourage you to download a C header file from Algorithmics' internet server at `ftp://ftp.algor.co.uk/pub/bonito/bonito64.h`; it will save you hours of typing, reduce the risk of mistakes, and means that the worldwide family of BONITO64 programmers will use the same register and field names. As an additional incentive, the online version is more likely to be up-to-date[3].

Where you see register names in **`bold fixed point`** font, they're names used in the online header file. And when you see a register name with a "dot" in it (e.g. **`sdcfg.awidth64`**), that means we're talking about the field called **`awidth64`** in the register **`sdcfg`**.

4.1. Address maps

The view of the system from the CPU is somewhat different from that as seen by a PCI bus initiator. We describe both.

CPU access map

Table 4.1 shows how CPU accesses are decoded to local and PCI resources.

In an ideal PCI system, all memory locations are dynamically set up by the system host controller at boot time. However, in many cases low PCI addresses cannot always be freely allocated; space below 1Mbyte or 16Mbyte may be required for certain PC "legacy" adapters and PC-world south bridge chips, which map ISA's 20- and 24-bit addresses into the lowest part of PCI space.

In a system which might want to use such legacy devices auto-configured PCI devices should be allocated addresses from at least 1Mbyte and perhaps 16Mbytes up.

From-PCI map

You can see how it might go together in Figure 4.1. The PCI regions identified (working from the bottom up):

- *Reserved for ISA registers/memory* : if your system uses an ISA bus, or (perhaps more cogent) any controller which needs to offer a programming model compatible with some old PC hardware, then it may need to use registers or memory locations in the low 16Mbytes, and quite likely the low 1Mbyte, of PCI space. It's therefore often wise to avoid using this region for anything else.
- *For ISA DMA access to BONITO64 SDRAM (example)* : more obscure. If your system may at some time support a DMA device which operates from an attached ISA bus, then that DMA device will itself only be able to reach PCI addresses in the low 16Mbytes; so it's useful to be able to map some of our SDRAM to this location. You can see here that it's possible to use one of the **`pcibase0-1`** registers with its mask and offset defined to access only 8Mbytes of local SDRAM.
- *BONITO64 registers* : here and subsequently, the name in bold (**`pcibase2`**) is a base register determined at configuration time.

---

[3] The great advantage of `bonito64.h` is that we use it to build our software, so has to be fairly up to date. Of course, it might have obsolete or unused definitions in; we'll try to zap them, but if the file describes something which isn't in the manual, it quite likely isn't in the chip either.

Base Address	*Size (bytes)*	*Class*	*Description*
0000 0000	256M	Memory	local SDRAM memory
1000 0000	64M	PCI_Lo0	PCI low-memory bus window for most CPU accesses to PCI space. Each of the three 64Mbyte windows can be separately positioned in PCI space with its own base register.
1400 0000	64M	PCI_Lo1	
1800 0000	64M	PCI_Lo2	
1c00 0000 1f80 0000	56M 4M	ROM	ROM (suitable for soldered flash) selected by *ROMCS1**. ROM (probably a socket, to provide a first-run bootstrap) selected by *ROMCS0**.
1fc0 0000	1M	Boot	Bootstrap memory location - starts at the magic MIPS reset-time entry point. According to reset-time configuration - see Table 5.2 - this can be mapped to the low 1M of either ROM space (*ROMCS0** or *ROMCS1**),
1fd0 0000	1Mb	PCI I/O	PCI I/O space - window to the low megabyte of PCI address range: used (and probably *only* used) to access the I/O space of an attached "ISA" bus.
1fe0 0000 1fe0 0100 1fe0 0200	256 256	BONITO64 BONITO64	BONITO64's own PCI configuration space registers available to other PCI bus masters BONITO64's internal registers. unused
1fe8 0000	512K	PCI	PCI configuration space reads/writes. Low parts of the address value driven on PCI comes from this address; high order bits from the **`pcimap_cfg`** register.
1ff0 0000 1ff4 0000 1ff8 0000 1ffc 0000	256K 256K 256K 256K	Local I/O	Local I/O bus devices decoded by *IOCS0-3** respectively
2000 0000	1.5Gb	PCI_1.5	Maps 1-1 onto PCI addresses. Most likely not very useful.
8000 0000	2Gb	PCI_2	PCI access window. Optionally mapped with either 1-for-1 addresses, or mapped down to the low 2Gb of PCI space. Available if you need access to a larger region of PCI space than is available in the lower-memory window. You'll need to program the MIPS TLB or use 64-bit pointers to get addresses bigger than `0x2000 0000` out of the CPU.
1 0000 0000†	??	Memory	vast quantities of space for larger local memories. The first 256Mbytes is a duplicate of the low part of the address map.

Table 4.1: CPU/local bus address map

This region provides the PCI view of BONITO64's internal registers. All registers are mapped at the same position relative to the PCI base as they are mapped in CPU space.

- *BONITO64 ROM, I/O (example)* : a view of the BONITO64 ROM and local I/O space. There's nothing to stop you trying to go through BONITO64 to that part of the PCI bus mapped into local memory - but it's bizarre and not much is guaranteed.
- *BONITO64 SDRAM (with and without IO caching) (examples)* : we show two large windows mapping all the SDRAM - once for accesses with the IOBC "disabled" in the sense described above, and one with IOBC caching fully enabled. You can setup BONITO64 to provide a PCI window to any power-of-two sized aligned region of local memory.

There's nothing to stop BONITO64 as initiator on the PCI bus targeting itself - such cycles are legal and return the expected data.

† For BONITO64 the implicit limitation to 256Mbytes of physical memory is unacceptable. However, the MIPS boot vector location makes it impossible to define as much as 512Mbytes consecutive memory starting at zero, and the MIPS run-time vector location makes it mandatory to have *some* local memory from address zero. The way we've done it in this memory map is not guaranteed to to be for real.

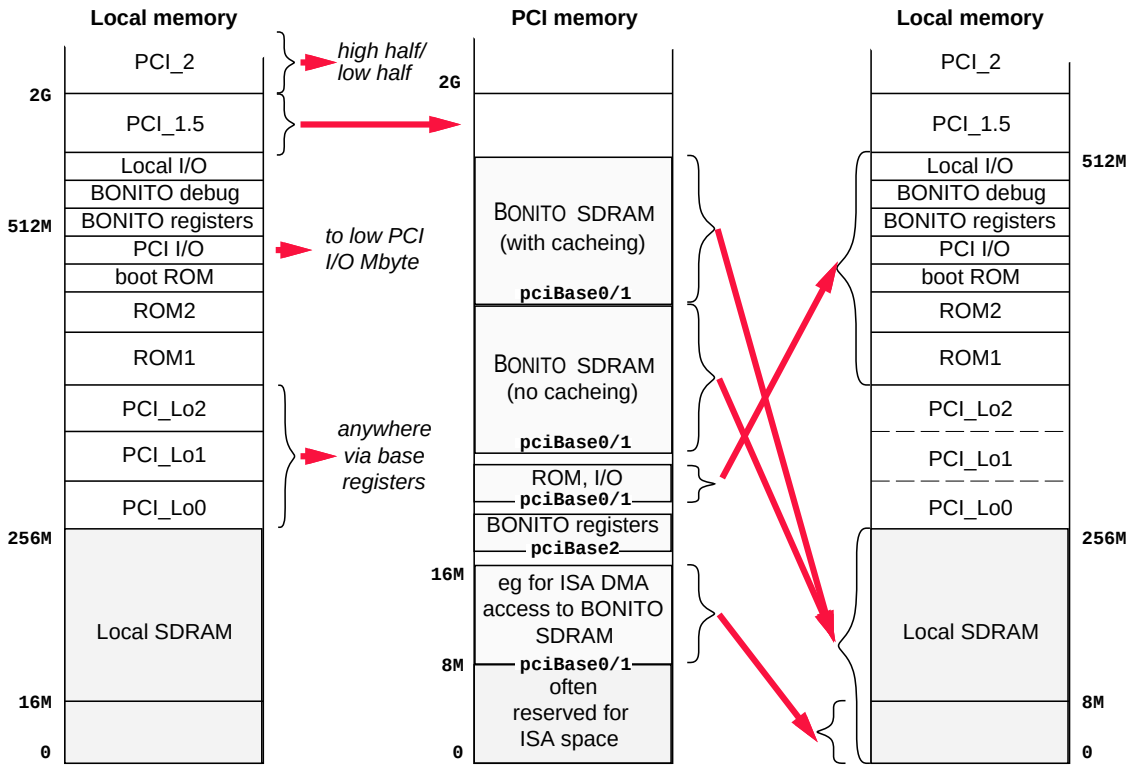


Figure 4.1 Memory regions and mappings between local and PCI space

4.2. Endianness and bus-to-bus transfers

A system has “endianness” if it supports both byte-wide and larger-integer accesses to the same memory object. A little-endian system is one where the least significant bits of the larger integer are stored in the lowest addresses, and a big-endian system is one where the most significant bits of the larger integer are found at the higher addresses. Neither is any more right than the other, though each is “obvious” in different circumstances; it’s a curse of computing that different CPUs and buses adopt opposite conventions. It’s virtuous to write software which works with either endianness, but virtue always demands sacrifice; porting a code-base which has only ever run with one convention can be hard work.

The PCI bus is little-endian; you can see that because the data bytes which travel over the byte lane *AD0-7* are the lowest-addressed bytes in the word.

Some PCI controllers do advertise facilities to help out systems with big-endian components; but everyone gets so confused about this stuff that our recommendation is to turn all those things off and sort the problem out using software and BONITO64’s facilities.

A MIPS CPU may be configured to run with either endianness. If you setup your CPU little-endian, you need read no further. But sometimes an existing code base may sway the balance the other way.

BONITO64 and PCI with a big-endian CPU

BONITO64’s operation is endianness-dependent in ways:

- *CPU interface*: the MIPS interface will not operate correctly unless BONITO64 and the CPU are both configured to the same endianness[4]. So the register bit which makes BONITO64 big-endian

[4] This is not quite true if the CPU uses a 32-bit bus; such CPUs can work in a limited fashion with a wrongly-configured BONITO64; enough to bootstrap themselves and

bonponcfg.cpubigend can be preset before an instruction is ever executed, as described in §5.2. (“Power-on settable configuration register - **bonponcfg**”) on page 23.

- *PCI bus* : is inherently little-endian.
- *IDE disk attachment* : the data is organised little-endian. Though on a hard disk which is only ever read and written through BONITO64, it won’t matter so long as it’s consistent.

First mantra (repeat until you believe it):

> When a big-endian CPU is used with a little-endian bus or peripherals, hardware can’t make the endianness problem disappear.
>
> But hardware can give you a choice: either the CPU and PCI bus will agree about the addresses and sequence of bytes, OR they will agree about the addresses, sequence and organisation of aligned 32-bit words.

You get a choice - but we very strongly recommend you follow the golden rule:

> When using a big-endian CPU enable BONITO64’s byte-lane swappers, so that PCI and CPU have a consistent view of byte addresses and sequence.

There are three separate register bits[5] to set to achieve this: **bongencfg.mstrbyteswap** and **bongencfg.byteswap** (see Table 5.3 on p.25) for PCI transfers, and **iodevcfg.wordswapbit_ide** (see Table 5.4 on p.27) for the IDE interface.

Endianness and the SDRAM port - nothing to do

In hardware terms, data is always conveyed between the CPU’s data path and the SDRAM’s data path through corresponding bit numbers. So long as you don’t reconfigure your CPU endianness “live” and expect to read data you wrote beforehand, nothing happens. If you should do anything so odd, the peculiar consequences are all inside the CPU: consult a MIPS CPU book!

Endianness and ROM cycles

BONITO64 allows MIPS CPUs to boot and run from narrow (8- and 16-bit) ROMs; when the CPU reads the ROM, BONITO64 performs multiple reads and assembles the bytes or 16-bit quantities to the width required.

Further, BONITO64 allows the CPU to run cached from ROM: when the CPU does a burst read from ROM for cache refill, BONITO64 performs enough ROM reads to provide a whole cache-line worth of data - usually 4×64-bits.

Because narrow ROM data must be unpacked to a wider bus, you can enquire into the relationship between the ROM address of an 8- or 16-bit piece of data, and the byte lane it occupies when being loaded across the CPU interface. The construction of ROM bytes (or 16-bit chunks) into a 32-bit word is fixed, with low-addressed bytes going to low-numbered byte lanes. However, the assembly of two 32-bit words into a 64-bit unit depends on the CPU’s endianness; see Figure 4.2.

---

change the endianness bits in software.

But that doesn’t work for 64-bit bus CPUs.

[5] It isn’t very sensible to have two separate bits, but that’s how the original 32-bit Bonito grew up.

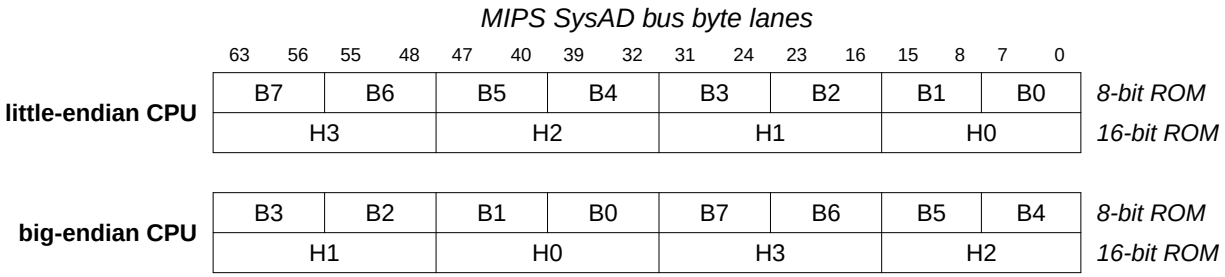


Figure 4.2 ROM data for the CPU - ROM addresses and byte lanes

In Figure 4.2 the “B0” etc numbers represent the byte address modulo 8 fed to an 8-bit ROM: the numbers “H0[6]” etc represent the word address modulo 4 fed to a 16-bit ROM.

The effect on the I/O bus of CPU reads of various width are shown in Table 4.2.

	Number of Cycles on ROM bus	
	8-bit ROM	*16-bit ROM*
CPU read width		
1 (byte)	1	1
2 (half-word)	2	1
3 (tribyte)	4	2
4 (word)	4	2
5	8	4
6	8	4
7	8	4
8 (doubleword)	8	4
burst	8×4	4×4

Table 4.2: CPU reads and resulting ROM cycles

That is, any read from ROM which can’t be achieved with a single ROM cycle is (for simplicity) implemented by reading at least 32 bits of data - the CPU will ignore the data it didn’t want. Only single-byte reads from an 8-bit device or half-word reads from a 16-bit device are guaranteed to result in a single cycle at the ROM pins - something you need to know when programming most flash devices.

When you’re programming flash memories, some reads from ROM have side-effects - in particular, the status read which tells you whether the ROM is ready for another operation. In that case you must read only data to the width of the ROM, or you will get confused.

Similarly ROM write operations - required for flash memories, of course - will only work at all if the CPU write matches the width of the ROM device; bytes for 8-bit devices, and 16-bit writes for 16-bit devices. You can always discover the width of the attached ROMs by reading the register bits **`bonponcfg.comcs1width`**/**`bonponcfg.comcs0width`** (defined on p.23).

Endianness and local I/O accesses

Connect I/O bus device’s addresses to *IOA2* and upwards, so that registers appear on 4-byte-aligned locations. Access these registers with word-wide (32-bit) load and store instructions for endianness-independent software. Of course, only the parts of the CPU’s data word which correspond to the I/O bus data bits wired to the device’s data bus are important.

[6] Old MIPS terminology: 16 bits is a “half-word”, abbreviated “H”.

When you do byte or other partial-word transfers from a MIPS CPU the active byte lanes depend on the CPU's endianness, so byte- or partial-word accesses to the local I/O bus can only be programmed correctly once you know the endianness of your CPU.

Endianness and the I/O bus

The I/O data bus has endianness if you use the 16-bit width; when two bytes of data are passed in parallel along these *IOD7−0* might be either earlier or later in byte sequence than the one found on *IOD15−8*.

Intel-type devices are likely to assume this bus is little-endian, if anything, but there's no byte-lane swapper for CPU accesses. The bus is slow enough that the software overhead of doing swapping is acceptable.

When you program DMA on the I/O bus (usually for an IDE disk interface) a byte-lane swapper is available and should usually be switched on when the MIPS CPU is big-endian: see **`iodevcfg.wordswapbit_ide`** (on p.27).

Endianness and PCI transfers, CPU and bus-initiated

When the CPU and its interface are big-endian there is bound to be trouble with accessing devices and memory over PCI bus, even though you've followed our advice and used BONITO64's byte-lane swapper, as described above.

With all PCI transfers swapped, your local CPU and PCI will share a common view of byte addressing, but bigger-than-byte integers out in PCI world - 32-bit device registers, for example - will appear byte-swapped to the big-endian MIPS CPU; your software will need to cope.

The byte-lane swapper does not affect either local CPU or PCI initiator accesses to internal BONITO64 registers; they're defined as 32-bit aligned objects and are unaffected by endianness. Software should never access BONITO64 registers with anything except 32-bit reads and writes.

4.3. Reset-time options and "mode bits"

From a cold start, some MIPS CPUs require a fairly complex reset sequence; BONITO64 takes care of it. Most MIPS CPUs require some reset-time configuration. Where this is through dedicated configuration pins, your design will need to pull those pins up or down as appropriate and BONITO64 has no role there.

But some MIPS CPUs use a couple of dedicated pins to read a stream of configuration bits at reset time. The mechanism was invented for the R4000 CPU in 1990 and is not directly compatible with low-cost serial ROMs. But here BONITO64 will handle it. All the MIPS CPUs compatible with BONITO64 require some hardware configuration to match the system they run in. Many of them retain a scheme first found in the MIPS R4000, which loads configuration bits as a serial bit-stream at reset time[7]. The R4000 was introduced when serial ROMs were relatively rare, and it's just bad luck that the simple interface the CPUs require is annoyingly incompatible with low-cost serial ROMs.

So BONITO64 provides a limited facility which - with most CPUs in most systems - will get the system up and running with an absolute minimum of additional hardware.

From a hard BONITO64 reset, the first 32 bits of the data stream are obtained by inverting the levels on the (weakly pulled-up on chip) CPU data bus *SysAD0-31*, starting with bit 0. Since the CPU will not be driving the bus at this point an external pull-down on any *SysAD* signal puts a "1" in the corresponding position in the mode bits stream. Most CPUs can be at least brought into a minimal working mode with a configuration with very few "1" bits.

[7] Some R4xx0 CPUs (such as NEC's Vr43x0 and Vr5432) don't do this; they use only static configuration augmented by internal software-writable registers.

If this allows you to get exactly the configuration you want, that's fine; but if you need something more subtle the CPU can now move on to reset itself using the **bongencfg.cpuselfreset** register bit. By writing the **bonponcfg.CPUtype** field to the special value 010, you specify that the CPU reset sequence caused by the self-reset operation will provide mode bits from the register **intpol**, borrowed for the purpose.

So long as the *SysAD*-defined settings are enough to run a simple piece of ROM software, the software can store the configuration of its choice into the *intpol* register and reconfigure the CPU accordingly.

And for very complex or controllable settings, it's also possible to configure BONITO64 to accept a simple stream of bits and use it without change.

4.4. Where to boot from? Two ROMs or PCI-initiated bootstrap

In most BONITO64 applications the MIPS CPU will bootstrap from a local ROM; you can specify either of the ROM chip select options, and choose an 8-bit or 16-bit ROM. This can't be done by software, so is part of the **bonponcfg** register which can be pre-configured by selective pullups on the I/O data bus: see §5.2. ("Power-on settable configuration register - **bonponcfg**") on page 23.

BONITO64 can be configured to allow its attached CPU to bootstrap from either of the two ROM regions provided in its standard address map (selected by *ROMCS0** or *ROMCS1**).

The two ROM regions are not equivalent. The *ROMCS1** window is bigger, and the smaller *ROMCS0** window (4Mbytes maximum) is intended for a first-time-only bootstrap, which can be very useful if your normal boot memory is to be programmed on the board in production.

However, it's possible to build a BONITO64 system with no ROM, where the MIPS CPU bootstraps from local DRAM memory. Booting from DRAM is only useful, of course, after someone has put a program into it; and this could only have been done by a PCI bus master which has taken control of the system.

The sequence for a PCI bootstrap is fairly complex, and the details are beyond the scope of this manual[8]. But the basic sequence goes like this:

1. BONITO64 should be reset but configured to preset the **bonponcfg.romBoot** field to the magic "11" value - you'll need pull-ups on a couple of *IOD* lines to get this effect.

   With this setting, the MIPS CPU will be held in reset after BONITO64 comes up.

2. The PCI-located host can now program BONITO64 from the PCI side - remember, all BONITO64 registers can be reached from PCI. The PCI host must initialise much of BONITO64 - in particular its SDRAM controller.
3. The PCI-located host can now fill SDRAM memory with a bootstrap program.
4. The PCI host re-writes **bonponcfg.romBoot** to the value "10". This will cause the MIPS CPU to be taken through its normal reset sequence, but its normal start-up address will now map to local SDRAM. From now on the MIPS CPU can take control.

4.5. Reprogramming BONITO64's PCI identity

The PCI specification lays down a standard "configuration space" and standard register format in every controller, as part of a larger scheme in support of automatic configuration of a large range of possible systems - what PC software suppliers have called "plug and play". When a PCI system is reset one CPU (the "PCI host") scans the bus reading configuration space, allocating memory space and enabling drivers as required.

[8] Some BONITO64-related software is available free from Algorithmics; see `http://www.algor.co.uk`, `ftp://ftp.algor.co.uk/pub/bonito/` or mail us at `bonito@algor.co.uk`

BONITO64 can be used in two roles. If the local MIPS CPU is the PCI bus host, then BONITO64's configuration space facilities are not very important - only the host *writes* PCI configuration space, and most of the time only the host reads it. But BONITO64 can also be used to build a subsystem - an intelligent controller. If, for example, you build a RAID disk controller you would like the host reading BONITO64 configuration registers to see a disk controller, not a "MIPS CPU bridge".

Three facilities in BONITO64 are available to help with this:

1. BONITO64's configuration-space registers - even those which the PCI specification assumes are only ever read by the host - may in fact be overwritten by the local host, thus changing its identity.
2. PCI configuration-space base registers are read by the host to establish the size of the memory regions a PCI device will share with the bus, and written by the host to establish their location within the overall PCI memory map.

   BONITO64's windows onto its internal memory may be very large; large windows are essential for some applications. But fixed-size large windows provide problems for the configuring host, which may run out of PCI memory space to allocate.

   So the apparent size of the regions mapped by BONITO64's base registers can be changed by the MIPS CPU; see the **`pcimembasecfg`** register described on p.34.
3. All this would be useless if the host were to complete its PCI bus initialisation before the local CPU had got around to setting up new values in BONITO64's configuration-space registers. So there's also an BONITO64 option - available as a reset-time configuration bit **`bonponcfg.configdis`** - which causes BONITO64 to defer host processing, by responding with a PCI "retry" to any configuration-space access. Local software should hurry to fix up BONITO64's configuration space registers before the host software times out.

5. Software-accessible registers and programming

We'll organise this programming guide starting at reset time and working forwards through the operations of the chip.

BONITO64's functions are controlled through a collection of registers. Apart from a few whose organisation is dictated by the PCI specification the registers are all 32-bits in size, even where only some of those bits have any meaning. They should always be read and written as whole 32-bit words.

All BONITO64 internal registers are accessible to both the CPU and an external PCI master[9]. Moreover, within the relevant memory region, the register offsets are the same as seen from both sides. Of course, the fact that it's possible to access registers from both sides doesn't commit us to solving problems caused by simultaneous access - you can, but often you shouldn't!

5.1. Register Summary

Table 5.1 names all the software-accessible registers of BONITO64 with a quick note as to what they do and a reference where to find them in this manual.

Register	*Page/figure*	*What is it*
bongencfg	25/Table 5.3	Early boot-time configuration, mostly PCI-related
bonponcfg	23/Table 5.2	Configuration bits which can be set either way using *IOD* pullups
copctrl	40/Figure 5.5	PCI copier registers
copdaddr	40	
copgo	40/Figure 5.5	
coppaddr	40	
copstat	40/Figure 5.5	
gpiodata	44/Figure 5.7	GPIO level read/write
gpioie	44/Figure 5.7	GPIO input/output setting
intedge	45/Table 5.14	Interrupts selected as level/edge triggered
inten	46/5.15	Separate interrupt enable bits
intenclr	46/5.15	Interrupt enables - per-bit clear
intenset	46/5.15	Interrupt enables - per-bit set
intisr	45/5.15	Readable interrupt inputs
intpol	46/5.15	Interrupt polarity
intsteer	46/5.15	Which of two CPU interrupt pins gets raised by each interrupt condition
iodevcfg	27/Table 5.4	I/O bus cycle characteristics
ldmaaddr	43	I/O bus DMA, mostly for IDE
ldmactrl	43/Figure 5.6	
ldmago	43/Figure 5.6	
ldmastat	43/Figure 5.6	
pcibadaddr	38/5.7.4	PCI address associated with some PCI cycle ending with an error

[9] A build-time option in BONITO64 allows external accesses to be denied by a pin-strap; in some systems this is an anti-piracy requirement.

Register	*Page/figure*	*What is it*
pcibase0	33/Table 5.7	Base registers in PCI configuration space, which define what regions BONITO64 makes available to other PCI initiators.
pcibase1	33/Table 5.7	
pcibase2	33/Table 5.7	
pcicachectrl	42/Table 5.12	Registers for the I/O buffer cache (also known as “PCI cache”).
pcicachetag	42/Table 5.13	
pciclass	33/Table 5.7	Standard PCI configuration registers
pcicmd	35/Table 5.9	
pcidid	33/Table 5.7	
pciexprbase	33/Table 5.7	
pciint	33/Table 5.7	
pciltimer	33/Table 5.7	
pcimail0-3	44/5.13	Mailbox registers
pcimap	37/Table 5.11	Register to fix the windows available to the local CPU to access PCI memory or devices.
pcimap_cfg	39/Figure 5.4	Used to complete the PCI address when the local CPU is using BONITO64 to perform PCI configuration cycles.
pcimembasecfg	34/Figure 5.1	Used by local host to size and position the PCI-accessible windows into BONITO64’s local memory and local I/O.
pcimstat	38/Table 5.12	How many posted writes are still pending?
sdcfg	30/Table 5.6	Set up BONITO64 to match the SDRAM shape, size, speed etc
timercfg	47/Table 5.15	Simple timer, mainly intended to let software figure out how fast system clocks are running.

Table 5.1: All registers

General principles for BONITO64 registers (read this)

Don’t re-type these register or field names; as we already said at the start of §4 above, go to Algorithmics’ ftp site and download `ftp://ftp.algor.co.uk/pub/bonito/bonito64.h`.

You’ll find all the definitions in that file:

- Are all in upper-case, an ancient C convention for “`#define`” constants.
- Are prefixed with “BONITO_”.
- In the case of register bits, further include the register name.

So a register field in the header file might be called “`BONITO_BONPONCFG_CPUTYPE`”. I’m reliably informed that 21st-century programmers like typing, but long upper-case names look pretty ugly in text, so we’ll leave it as **`bonponcfg.cputype`**.

To avoid lengthening the whole manual with endless repetition, the following general rules apply to the use of BONITO64’s registers:

- All registers are writable unless explicitly stated to be read-only.

- Wherever it is reasonable to do so - and unless the detailed description says otherwise - you can read back the value you last wrote to a BONITO64 register.
- Some options affect BONITO64's support for the CPU bootstrapping itself, so must already be correctly set when the system first comes out of reset. Register fields representing these options take their values by sampling the signals *IOD0-15* (the IO data bus) while the reset input *SYSRESET** is still active. BONITO64 has a weak internal pull-down on each *IOD* signal line, so to set one of these register fields to 1 your system should have an external pull-up resistor on the corresponding line; a 4.7kΩ resistor to 3.3V is recommended.

  These configurable bits are gathered together into the **bonponcfg** register.

  Even earlier in reset, some MIPS CPUs use a serial data stream ("mode bits") to load configuration information; see §4.3 for details on how those bits can be set by pull-downs on the *SysAD* bus, and set to more complicated values for use after a self-reset.
- Many other register bit-fields are forced to a fixed level (most often zero, occasionally 1) following reset. However, this manual will document those initial values only when they are important to early operation. Your software should take responsibility for programming all relevant registers to reasonable values early in the bootstrap sequence.
- Register bit fields which are not defined in this manual are just that - undefined; they will be marked with a "×" in the tables. They'll most often read zero, but that's not guaranteed; and they should always be written zero. Absolutely anything might happen if you write them to something other than zero.

  We make only one promise about these values: in read/write registers it will always be safe to write an undefined field with the data you just read from it.

5.2. Power-on settable configuration register - bonponcfg

The **bonponcfg** register brings together BONITO64 control bits which may be set to either 1/0 at reset time. To get a 1 value following reset in a **bonponcfg** field, you need a pull-up on the corresponding *IOD0-15* signal; to get a zero, make sure that all devices connected to the line are tri-state during reset (they normally will be). In fact, the presettable section of **bonponcfg** is really 18 bits long, and the two highest bits get their reset-time value from the ROM chip select signals *ROMCS0-1**.

bonponcfg register

Bit(s)	*Name*	`Value` / *Effect*
18	**syscontrollerrd**	Read-only bit - 1 if the *SysController** input is tied low, allowing BONITO64 to drive the PCI *RESET** line.
17-16	**romcssamp1-0**	This field has no hardware effect, but returns the power-on level of *ROMCS1-0**, so could be used to set up some software option.
14	**cpubigend**	1 to support big-endian MIPS CPU; see §4.2 for a full description of the consequences.
13	**cpuparity**	1 to enable per-byte parity checking; 0 to disable checks. It will not usually be necessary to set this bit from power-on. Note that BONITO64 *always* passes SDRAM parity straight through, and generates parity for I/O and PCI data. You'll normally set this if and only if you have a 72-bit wide memory system to hold the parity bits. SDRAM configuration is discussed in §5.5. ("SDRAM configuration") on page 29 below.

bonponcfg register

Bit(s)	*Name*	`Value` / *Effect*
12	**burstorder**	Set one of two options for the order data is returned from SDRAM to BONITO64 and thence to the CPU in a cache-line-sized burst. A value of 0 selects "sub-block" (traditional for MIPS CPUs with a *SysAD* interface); a value of 1 selects sequential, which may be used in some variant CPUs. The SDRAM mode is automatically set during BONITO64 reset. Changing this value in software won't have any effect (on the SDRAMs) until you issue an SDRAM mode-set command, as described in §5.5.1 below. Note that this value is also used to determine how ROM data is passed back to the CPU (when running cached from ROM) - and that takes effect immediately.
11	**romCs1fast**	Select "faster" operation on the ROM attached to these select signals; slow ROM requires an access time of 160ns, while fast ROM is expected to be 90ns. Don't change these fields while running from the affected ROM.
10	**romCs0fast**	
9	**romCs1width**	read-only - 1 for 16-bit ROM, 0 for 8-bit
8	**romCs0width**	
7-6	**romBoot**	Where CPU boots from; picks which memory region will be selected for accesses in MIPS' traditional `0x1FC0.0000` start address. `00` from the low 4Mbyte of ROM attached to *ROMCS1**. `01` from ROM attached to *ROMCS0**. `10` from local SDRAM. This works only if some other part of the system has filled it with appropriate MIPS code, of course; you'll start with the next value: `11` the MIPS CPU is held in reset until this field is changed to another value - usually "`10`" as above. Used for a system which expects to have software uploaded into SDRAM at power-on.
5	**config_dis**	when set 1, BONITO64 responds as target to any PCI bus configuration cycle with a "retry" response. This allows your system to hold off configuration by an external PCI host while the local CPU's bootstrap software patches the standard PCI configuration registers. You should not leave this set for more than a few ms, or your host may give up on you; dangerous but useful. See the discussion of PCI memory allocation §5.6. ("BONITO64 registers available in PCI configuration space") on page 34 for why this might be useful.
4	**is_arbiter**	when set 1, BONITO64 operates as PCI arbiter. When 0, the roles of the zero-th request and grant signals are reversed, and BONITO64 will use the services of an external arbiter.
3	**pcireset**	If BONITO64 is acting as PCI bus controller (ie controlling the reset signal), this is where you can set it. A zero value asserts the active-low PCI *Reset** signal; a "1" deasserts it. Note that in early specifications, this bit was used to configure BONITO64 as PCI bus controller. But that's now done with the dedicated signal *SysController**.

bonponcfg register

Bit(s)	*Name*	`Value` / *Effect*
2-0	**`CPUtype`**	What kind of CPU is attached? Undefined so far in BONITO64.

Table 5.2: Fields in **bonponcfg**

5.3. Configuration bits with power-on defaults - bongencfg

The **bongencfg** register brings together early-bootstrap options, but which power-up from reset to a fixed state - most often zero. Almost all of them are PCI-related.

bongencfg register

Bit(s)	*Name*	*Value*	*Effect*
17	**`shortcopytimeout`**	`0/1`	Normally (this field 0), BONITO64 will abort a locally-initiated PCI cycle if it gets a "retry" response 4000 times running - on a quiet 33MHz PCI bus this will take about 800μs. Change this field to "1" if that's not enough, and raise the number of retries to 16000.
16	**`noretrytimeout`**	`0/1`	Normally (with this field 0), BONITO64 will not retry indefinitely on a locally-initiated PCI cycle but will time out as described above. Set this field 1 to disable that timeout. This risks lock-up, but makes the software simpler.
15	**`buserren`**	`0/1`	Set 1 to cause any CPU-initiated PCI bus read which terminates without data to cause a MIPS "bus error" exception.
14	**`mstrbyteswap`**		Set 1 to enable the PCI byte-lane swapper for transfers between PCI and local memory or CPU, when BONITO64 is the PCI bus initiator (master). Except in bizarre circumstances and after deep thought, this should be set 1 if and only if your CPU is big-endian; see §4.2. Only in even more bizarre circumstances should it ever be set differently from the **`byteswap`** bit described below, which controls transfers where BONITO64 acts as target on the PCI bus.
13	**`cachestop`**†	`0/1`	When 1, all PCI-initiated accesses to local memory are denied with a "retry" signal, so that the I/O buffer cache state can only be affected by CPU activity. Probably only for IOBC test code.
12	**`pciqueue`**†	`1`	Write 1; the zero value is solely for IOBC diagnostics.
11-10	**`cachealg`**†		Diagnostics only - leave these alone
9	**`wbehinden`**†	`1`	Value 0 is for IOBC diagnostics only. If you set this zero, data written by a PCI initiator to BONITO64's local memory is not automatically written back when the next cache-line sized chunk of memory is written.

† July2001: fields in Table 5.3 marked with a "†" (dagger) are IOBC related and likely to change quite soon. Check header files and software examples.

bongencfg register

Bit(s)	*Name*	*Value*	*Effect*
8	**prefetchen†**	1	Value 0 is for IOBC diagnostics only. IOBC read prefetch control. Set 1 to allow BONITO64 to read ahead by a cache line. When a PCI initiator is reading a stream of data from local memory, this greatly improves performance by fetching what will probably be the next line of data from memory in parallel with the PCI transfer.
7	**uncached†**	0/1	Set 1 to enable the IOBC. 0 is for diagnostic code only.
6	**byteswap**	0/1	Set 1 to enable the PCI byte-lane swapper for transfers between PCI and local memory or CPU, when BONITO64 is acting as the PCI bus target (slave). Like the **mstrbyteswap** bit described above, this should be set 1 if and only if your CPU is big-endian.
5	**irqa_from_int1**	0/1	set 1 to cause the PCI interrupt signal *INTA** (as an output) to be driven from the second, *Int1**, output from BONITO64's interrupt controller. If set to 0 *INTA** (if it's an output at all), will be explicitly controlled by the special "interrupt enable" bit **inten.irqa**. It's cheating to use this register (all its other bits control interrupt masking) but convenient because that register bit can be set and cleared by single write cycles to the **intenclr**/**intenset** addresses. For compatibility reasons many BONITO64 implementations will OR that value with the value of **bongencfg.force_irqa**, but that's deprecated.
4	**irqa_isout**	0/1	1 to drive *INTA**, 0 for it to be an interrupt. It is readable as one of the input conditions of the interrupt controller, see §5.15.
3	**force_irqa**	0/1	Set 1 to drive *INTA** active. Note that since PCI interrupt signals are defined "open-collector" they are only in fact driven low; in the absence of any drive by any of possibly multiple connected devices a pull-up produces a high level. So the 0 value means "don't drive the signal".
2	**cpuselfreset**	0/1	Write a "1" to cause BONITO64 to cold-reset the MIPS CPU. Likely to be used only very soon after a real power-on reset, in systems where the CPU configuration is changed by software very early in bootstrap.
1	**snoopen**	0/1	Set 1 to enable the IOBC to snoop uncached CPU accesses. Sometimes helpful and usually harmless. Should normally be set.
0	**debugmode**	0/1	*Powers-up to 1*. Enable debug mode, in which all CPU accesses and some PCI ones become visible on an attached debug board - see §2.7. There may be some cost in performance or power, so turn this off in a system if you know no debug board will be used.

Table 5.3: Fields in **bongencfg**

5.4. Local I/O configuration

BONITO64 has a local I/O bus which is used to access ROMs, but is otherwise mostly independent of its other ports. Four CPU memory regions correspond to four different possible chip selects (*IOCS0-3**) - and of course there are the ROM regions.

The local I/O bus has a 16-bit data bus. ROMs are treated specially, and when read look like 64-bit-wide byte-addressed memory. But the rest of the I/O space - used for device registers - looks very much like a chunk of 32-bit memory, with only (at most) the lower 16 bits of any word populated. We strongly recommend that you program it with word-wide accesses.

The bus uses a simple Intel-style signaling system, where chip selects qualify addresses but devices respond only when they see one of the separate read and write strobes *IORD**/*IOWR** go active.

Addresses, chip selects and (for writes) data are valid for some time before the falling edge of the strobe, and are held stable until some time after the rising edge of the strobe - where “some time” is two CPU clock periods plus or minus a little skew. This means that complex designs needing more decodes or a different protocol have time to decode the chip selects and addresses and provide a clean signal before the strobe is activated.

Read data is sampled by BONITO64 at the rising edge of the *IORD** signal.

Write data is driven by BONITO64 with the same timing as addresses. Devices differ on when they sample the write data; at latest, they will accept data on the rising (deasserting) edge of *IOWR**, but many devices will acquire data sometime during the strobe.

The only configurable timing in this structure is the width of the read/write pulse. This may be one of two values; nominally 200ns and 600ns, corresponding to “fast” and “slow” devices.

In addition, the read/write pulse can be extended by using the *IORDY* signal. To slow the cycle *IORDY* should be driven low at least two CPU clock times before the cycle would normally have ended - the cycle will end quite quickly once *IORDY* is returned to the high state, but note that BONITO64 still samples data on the rising edge of *IORD** for normal cycles.

Configuration for the different local I/O bus decodes is handled by registers as shown in Table 5.4.

`iodevcfg` register

Bit(s)	*Name*	*Value*	*Effect*
31-26	**`cpuclockperiod`**		Set this register to the period of the master clock *ClockIn*, rounded to the nearest nanosecond, to make the I/O bus timings and DRAM refresh period reasonably accurate. Low values (zero, in particular) are not interpreted, and result in slow I/O cycles and over-frequent refreshes - harmless as a default, but not efficient. You can set this field as low as 8 (corresponding to 125MHz).
25	**`udma_tenv_ide`**	`0/1`	Adds an extra clock period into part of the UDMA protocol (the “*tENV*” minimum). Must be set “1” when the CPU is running above 100MHz.

iodevcfg register

Bit(s)	*Name*	*Value*	*Effect*
24-21	**dmaoff_ide**	0-2	IDE transfer timing: a value of *n*: **In UDMA modes**: sets the UDMA pause-to-stop time ("*tRP*" in our copy of the UDMA specification) to *n* CPU clocks. **In regular DMA modes** sets the period between successive DMA transfer to *(2×n)* CPU clocks.
20-16	**dmaon_ide**	0-2	IDE data transfer rate control: a value of *n*: **UDMA modes:** sets the UDMA to write at one half-word per *(n+1)* CPU clock periods. The UDMA read speed is determined by the disk drive and the speed at which BONITO64 can write the data received to memory. **Regular DMA** sets the width of the *IORD*/IOWR** strobes to *(2×n)* CPU clock periods.
15	**modebit_ide**	0/1	Set 1 to enable "UDMA" transfers to an IDE disk. Set 0 to use conventional DMA, either "word at a time" or what the disk manuals call "multiple" DMA.
14	**wordswapbit_ide**	0/1	Set 1 to swap bytes when DMA'ing from the I/O data bus. You should probably set this when using an IDE disk with a big-endian MIPS CPU.
13	**speedbit_ide**		not implemented
12	**buffbit_ide**		not implemented
11 8 5 2	**moreabits_cs3** **moreabits_cs2** **moreabits_cs1** **moreabits_cs0**	0/1 0/1 0/1 0/1	set 1 to drive the whole CPU address on *DD31-0* during an I/O access associated with each of the I/O chip selects *IOCS0-3**, independently; set 0 when the device only needs the addresses on *IOA4-0*.
10 7 4 1	**speedbit_cs3** **speedbit_cs2** **speedbit_cs1** **speedbit_cs0**	0/1 0/1 0/1 0/1	set 1 if this is a "fast" device (nominal 200ns read/write strobe); 0 for a "slow" device (nominal 600ns).
9 6 3 0	**buffbit_cs3** **buffbit_cs2** **buffbit_cs1** **buffbit_cs0**	0/1 0/1 0/1 0/1	set 1 if the device selected by this chip select is located behind a bidirectional buffer controlled by the '245-type signals *IODIR* and *IODEN**; 0 otherwise.

Table 5.4: Fields in **iodevcfg**

ROM cycles are different. The *IORD** pulse width for ROM is also programmable to either 90ns or 160ns (nominal), but is set up in the **bonponcfg** register described in section Table 5.2 - ROM speed is important from the very first cycles.

The address bits *IOA0-5* are not only valid for simple ROM cycles but count up to support burst reads from ROM when the CPU is running cached. During ROM cycles the whole ROM address is always available on the SDRAM data bus *DD0-31*. See the note in the endianness section (§4.2) on how I/O byte addresses relate to CPU addresses.

5.5. SDRAM configuration

BONITO64 can be configured with a range of SDRAM memory systems. We need some standard names for talking about the chunks of SDRAM you build the system out of, and the SDRAMs themselves already have “banks” and “rows” inside. So by analogy with the DIMM modules (which many designs will use) we’ll talk about *modules* each of which has one or two *sides* (distinguished by separate chip selects)[10]. Where we want to talk about one of the 32-bit halves of a 64-bit side we’ll call it a *half-side*. In this manual we’ll talk about *modules* “A” and “B”, and *sides* “0” and “1”.

If you solder the chips to the board then you may complain about the use of the word “module” - but it’s the best we could think of.

BONITO64 can directly support two modules - 32- or 64-bits wide - and each may have two sides - that is, there are four chip selects. The chip sets are called *DCS0L**/*DCS0H** and *DCS1L**/*DCS1H**, which can be confusing - the names match those used on DIMM modules better than they match the conventions of this manual. They connect like this:

	Side	
	0	*1*
Module A	`DCS0L*`	`DCS0H*`
Module B	`DCS1L*`	`DCS1H*`

Table 5.5: Modules, sides and chip selects

You’ll probably prefer to set up BONITO64 so that the memory map includes the whole SDRAM system size with no holes or wrap-arounds. So the relevant parameters of the SDRAM system are:

- Is there just one, or are there two modules?
- Does this design use a registered driver to buffer multiplexed addresses and other control signals[11] (useful for larger memory arrays?)

And for each module:

- What (if any) unusual configuration options must be fed to the SDRAMs at power-up (by writing the SDRAM “mode register”)? PC-100/133 SDRAMs, for example, can return data either 2 or 3 clock periods after the column address - but the actual timing to be used must be programmed into the parts when the whole system is initialised after reset.

  Other configuration settings in the mode register are available by running a special cycle: see §5.5.1 below.
- Does it have one or two sides?
- How many internal banks are there in its constituent SDRAM components - two or four?
- How many *MuxAD* addresses does the SDRAM decode in its first (*Ras**) phase? Components supported by BONITO64 decode between 11 and 14.
- How many *MuxAD* addresses does the SDRAM decode in its second (*Cas**) phase? BONITO64 can work with between 8 and 11.

That’s quite complicated to allow for in the design, but also quite complicated for software to find out about.

---

[10] Some DIMM modules provide two chunks of DRAM all soldered to the same side of the board, while others have a single chunk of DRAM split between top and bottom; but we’d still call use the word “side” for any any DRAM or group of DRAMs which share a chip select. Sorry; we have to call them something.

[11] Even DDR SDRAMs have control signals which are presented on the rising edges of the supplied clock - only the data burst uses both clock edges.

By a convention initiated by IBM and sanctified by PC-100, modern DIMM modules carry “self-portrait” data encoded in a tiny on-DIMM EEROM device, accessed through a 2-wire interface. Software can drive BONITO64’s GPIO pins read and write the EEROM.

Memories built onboard or with proprietary modules will not usually have such information in EEROM - the board designer should consider how much information is needed and how software should detect any variation in DRAM types.

To complete configuration of BONITO64’s SDRAM controller you’re also going to need some timing information, determined by your hardware design engineer, such as whether the memory array uses external registers to provide higher drive for address and control signals.

The controller can handle four physical groups of SDRAMs. But inspired by the organisation of DIMMs, BONITO64’s registers assume that the two sides within each module have identical organisation and timing.

Modules must be 64/72-bits wide.

BONITO64 implements parity checking on the memory array - it passes parity through on CPU cycles (MIPS CPUs will check the incoming parity), and generates/checks it on all other cycles. You can disable parity checking if your memory is only 64 bits wide - it’s done through the **`bonponcfg.cpuparity`** bit described on p.23.

You tell BONITO64 about the characteristics of each module through the register **`sdcfg`** shown in Table 5.6. The whole register is cleared to zero on power-up.

`sdcfg` register

Bit(s)	*Name*	*Value*	*Effect*
31	**`drammodeset_done`**	`0/1`	Read as part of controlling an SDRAM mode-set command, as described in 5.5.1 below.
27-26	**`dramrfshmult`**		Sets the interval between pairs of SDRAM refresh cycles, as multiples of a nominal value of 7.8μs. (The correctness of the nominal value depends in turn on your setting of **`iodevcfg.cpuclockperiod`** shown in Table 5.4.
		`0`	3.9μs
		`1`	7.8μs
		`2`	15.6μs (default value from reset)
		`3`	31.2μs
25-24	**`dramburstlen`**		Used to program the SDRAM chips for their burst length, as described in 5.5.1 below. This is only important if you set the SDRAMs into “sequential” burst order (see **`bonponcfg.burstorder`**).
		`0`	1 data phase
		`1`	2 data phases (default)
		`2`	4 data phases
		`3`	8 data phases
23	**`dramparity`**	`0/1`	Set 1 to enable parity generation/checking in the DRAM system.
22	**`dramextregs`**	`0/1`	Set 1 if your system uses high-drive registers to boost multiplexed addresses and shared control signals to the SDRAM modules.
21	**`drammodeset`**	`0/1`	Used to run an SDRAM mode-set command, as described in 5.5.1 below.
20-19	**`extraswidth`**	`0-1`	set 1 to extend the minimum DRAM cycle time from 5 to 6 clocks. The default (0 $\rightarrow$ 5 clocks) works with PC-100 SDRAMs at 100MHz. May be required when 5$\times$ your clock period isn’t enough to meet the DRAM parameter often called *tRP*. Values 2-3 lead to undefined results.

sdcfg register

Bit(s)	*Name*	*Value*	*Effect*
18	**extprech**	0/1	Usually 0, which should be fine for PC-100 SDRAMs at up to 100MHz. A value of 1 inserts an extra clock in the precharge period (the minimum period between SDRAM cycles), when 2× your clock period isn't enough to meet the SDRAMs *tRP* timing parameter.
17	**extrascas**	0/1	Set 1 to add another clock period between the row address (row-select command) and column address (read/write command) phases. Has no effect on writes. PC-100 SDRAMs don't need this set at 100MHz. Your system needs it if 2× your clock period is less than the minimum value of the SDRAM's *tRCD* timing parameter.
16	**extrddata**	0/1	CAS latency for SDRAMs; to make this effective you need to run a SDRAM mode-set command, as described in 5.5.1 below. PC-100 SDRAMs requiring both CAS latency 2 and CAS latency 3 at 100MHz are available. 0 CAS latency 2 clocks. 1 CAS latency 3 clocks (power-on default)
			SDRAM shape fields - 'a' for the first module, selected by *DCS0L**/*DCS0H**, and 'b' for the second, selected by *DCS1L**/*DCS1H**.
15 7	**bwidth64** **awidth64**	 0 1	b module data width. a module data width 32 bits wide 64 bits wide
14 6	**babsent** **aabsent**	0/1	Set this if no memory is fitted in the B chip selects ("module") Set this if no memory is fitted to the A chip selects ("module"). Required if the memory is all in the B module.
13 5	**bsides** **asides**	 0 1	"sides" in this module - separate bits for sides 0 and 1 just side 0 fitted (or none) both sides fitted
12 4	**bbankbit** **abankbit**	 0 1	No of internal banks in SDRAM components Two four
11-10 3-2	**bcolbits** **acolbits**	 00 01 10 11	no of column addresses at SDRAM† 8 9 10 11
9-8 1-0	**browbits** **arowbits**	 00 01 10 11	no of row addresses at SDRAM† 11 12 13 14

Table 5.6: Fields in **sdcfg**

Note that from system reset **sdcfg** is cleared. This is intended to leave the SDRAM configuration in a default state which will yield 4/8Mbytes of functioning memory, so long as any usable devices are

† Some SDRAM manufacturers count an internal bank-select address - the same as the BONITO64 signals called *DBA0-1* - into their "row" and "column" address counts, so you'll need to subtract one from those values before programming Read carefully.

connected to the chip-select *DCS0L**. This can be used for some bootstrap functions.

5.5.1. Configuring (setting modes in) the SDRAM chips

SDRAM devices have some interface options which are usually programmed into the chip when the system is powered up, and then left alone. Regular SDRAMs have only three options of any interest at all to BONITO64 systems:

- *CAS latency*: defines how soon the first data of a burst should be returned. The SDRAM chip doesn't know its own access time, and you may need to set this as either 2 or 3 clocks. Your hardware designer should know which. BONITO64 defaults to a 3-clock period - safe, but maybe you could go faster.

  This is controlled by the register bit **`sdcfg.extrddata`**, set 0 for latency=2 and 1 for latency=3.

- *Data burst order*: SDRAMs can return burst data either sequentially (wrapping at the end of an alignment unit) or in "sub-block" order. You need to program this to match the requirements of your CPU - all the MIPS CPUs we can tell you about use sub-block order.

  This is controlled by the register bit **`bonponcfg.burstorder`**; 0 for sub-block, 1 for sequential

- *Data burst length*: if you selected sequential order, you also need to tell the SDRAM how big your cache lines are, so it knows when to wrap.

To set up your SDRAMs, you need to

1. Write the values of your choice to the register fields above.
2. Set the **`sdcfg.drammodeset`** bit to 1. This will cause the SDRAM mode-set command to be scheduled at the end of the next SDRAM refresh operation. You may have to wait as much as 30μs for this to happen.
3. Poll the **`sdcfg.drammodeset_done`** bit, looking for a value of 1. When you see it, the mode-set operation has happened.
4. Set the **`sdcfg.drammodeset`** bit back to zero.

5.6. BONITO64 registers available in PCI configuration space

The registers shown in Table 5.7 conform to the PCI 2.1 standard[12]. BONITO64 does not use any non-standard configuration space registers; all the device-dependent programming is accessed through the “Bonito Registers” region, at a PCI address determined by the setting of the **pcibase2** base register.

As well as being available to other PCI initiators through PCI configuration space cycles, these registers can be read and written by the CPU; they’re available to the local CPU in the lowest 256 bytes of BONITO64’s internal register block. Some registers which are read-only from the PCI side are writable from the local side.

By reset-time option, BONITO64 can be caused to initially reject configuration cycles with a “retry” response. This is intended to provide time for a local CPU to write non-standard values into the configuration registers. You need to do this very early in the bootstrap sequence, or the configuration host may time out.

	31 … 16	15 … 0	
pciDiD	Device ID 00D5h	Vendor ID DF53†	00h
pcicmd	Status	Command	04h
pciclass	Class Code	Revision ID	08h
pciltimer	0 / Header Type	Latency Timer / 0	0Ch
	Base address registers		
pcibase0	Local SDRAM or I/O, up to 256Mbyte		10h
pcibase1	Local SDRAM or I/O, up to 256Mbyte		14h
pcibase2	BONITO64 register bank, 64Kbyte		18h
	unused		1Ch
	×		28h
	0	0	2Ch
	unused		
	Reserved		34h 38h
pciint	Max_Lat / Min_Gnt	Interrupt Pin = 1 / Interrupt Line	3Ch

Table 5.7: Standard PCI configuration space registers

5.6.1. BONITO64 device and vendor ID

FPGA controllers used by Algorithmics in development return an unauthorised vendor ID of `0xDF53`.

The **pciDiD** register can be written with a new value to reflect the vendor and device ID code of a subsystem supplier.

[12] Note that FPGA versions of BONITO64 will be delinquent; because it is a soft-logic part it will take some 40ms after power-on before it functions. If the host attempts PCI configuration during this time, BONITO64 will not respond.

† The value shown is the unauthorised “vendor ID” Algorithmics put in FPGA prototypes. Most BONITO64 users would put their own vendor ID here.

5.6.2. Base address registers and PCI

The PCI specification requires that base address registers behave like this:

31 — 4	3	2 — 1	0
Base Address	P	00	0

Table 5.8: PCI configuration space base address register

The “P” bit is set for memory regions which are memory-like (always return a whole word of data regardless of byte enables, reads have no side effects, and writes can be arbitrarily merged and combined).

Address regions on PCI are always expected to be naturally aligned by their size (being a power of 2) - a 256Mbyte memory region may only be allocated on a 256Mbyte boundary, for example. Configuration hosts figure out how much memory can be mapped by a particular base register by writing an all-ones pattern, and reading it back; address bits which are don’t-care when matching the base address return 0. So for a 256Mbyte region, you write `0xFFFF.FFF0` and read back `0xF000.0000`.

5.6.3. BONITO64 base address registers

BONITO64 can map regions of up to 256Mbytes. But offering a space that big can cause problems to the configuration host, because it can’t really do anything except map the whole region - and that may consume so much address space that configuration fails.

So by default BONITO64’s memory regions each only offer a 8Mbyte window; the window can be re positioned within BONITO64’s local memory using an offset programmed in the register **`pcimembasecfg`**. Moreover, the local host can also change the apparent size of the windows in systems where it can change **`pcimembasecfg`** before the configuration host sees it; set the register bit **`bonponcfg.config_dis`** to hold off the host’s configuration cycles while you get in and change it; you can arrange to set that bit from system reset, as described in §5.2. (“Power-on settable configuration register - **`bonponcfg`**”) on page 23.

The **`pcibase2`** region provides access from PCI to all the programmable registers inside BONITO64 - all accessible either via the PCI bus or from the local CPU.

The **`pcibase0`** and **`pcibase1`** regions’ behaviour depends on the setting of the register **`pcimembasecfg`**, which is shown as Figure 5.1.

31 — 24	23 — 22	21 — 17	16 — 12	11 — 10	9 — 5	4 — 0
0	**`pcibase1`** options			**`pcibase0`** options		
	io	cached	trans	mask	io	cached	trans	mask

Figure 5.1 Fields in **`pcimembasecfg`**

Where the fields are as follows:

`io`: 0 to map this window into BONITO64 local SDRAM, 1 to select the upper part of the local map (which contains ROM and local I/O) - you can think of this as just setting bit 28 of the local address used for this access.

`cached`: 1 to make use of the IOBC for memory accesses - essential for good performance. You can set a memory region with this bit 0 when you want local memory transfers to happen synchronously with PCI transfers; sometimes useful for diagnostics or to find problems. See the IOBC section §5.10 for more about this subject.

`trans/mask`: two five-bit fields which determine bits 27-23 of the local address generated by accesses in this region. The PCI address is first masked to remove bits 31-28, and then PCI address bits 27-23 are

first ANDed with the complement of the **mask** field, then ORed with the **trans** field.

The **mask** field has another effect, in that it alters the subsequent behaviour of the corresponding configuration space register. If the local CPU programs the **mask** fields in **pcimembasecfg** before enabling incoming configuration cycles, the mask value will be fed back to the configuration host as the size of the available window.

5.6.4. Command Register Options

The PCI standard command register is used to negotiate some decisions about bus use to a PCI host, which might choose to take some actions in the light of them. A couple of these are writable. Note that the PCI standard "Status" register is the other half of this register, and is described in Table 5.10 below.

pcicmd register

Bit(s)	*Name*	*Value*	*Effect*
15-10		0	Unimplemented bits
9		0	Fast back-to-back transfer enable - always 0, never done
8	**serren**	0 1	Reporting of address-time parity errors - writable Do nothing Drive *SERR** signal when detected.
7	**astepen**	0	Address/data stepping - always 0, never done
6	**perrespen**	0 1	Response to parity errors on PCI bus - writable Ignore parity errors Respond to PCI parity errors. Note them in **pConfSC.Status**, and generate an interrupt if unmasked; drive the *PERR** signal.
5		0	"VGA graphics controller option" - PC specific, always zero.
4		0	Memory write and invalidate. Always 0, BONITO64 only does plain writes.
3		0	Special cycles - always 0, BONITO64 doesn't issue special cycles
2	**mstren**	0 1	Master enable - writable Don't initiate PCI bus cycles Initiator role enabled. BONITO64 almost certainly needs this bit set.
1	**memen**	0 1	Memory space enable - writable Disable target functions. Host should not leave this zero. Allow BONITO64 to respond on PCI
0		0	PCI I/O space enable - BONITO64 never responds to I/O space cycles.

Table 5.9: Fields in **pcicmd** *- PCI configuration space "Command" register*

5.6.5. Status/(Error Clear) Register

This is also a PCI-blessed standard register. It can be read as the high bits of the **pConfSC** register. You can write it to clear down any error flag or set of flags, by writing a 1 into the bitfield corresponding to the flag(s) you want to clear..

pcicmdStat register

Bit(s)	*Name*	*Value*	*Effect*
31	**perr_clr**	1	Detected a parity error - as initiator (reading) or as target receiving write data. Unaffected by the enable bit in **pConfSC(Command)**.
30	**serr_clr**	1	BONITO64 is driving *SERR**, having noticed an address parity error
29	**mabort_clr**	1	"Master-Abort" signaled BONITO64 initiated a read or write but no target responded with *DEVSEL**(*Master-Abort* in the PCI specification)
28	**mtabort_clr**	1	Target abort received BONITO64 initiated a read or write but the target couldn't do it and doesn't want the transaction retried (*STOP** asserted and *DEVSEL** deasserted).
27	**tabort_clr**	0	Target abort signaled. Always 0 - BONITO64 never responds with a target abort.
26-25		01	*DEVSEL** speed. Always this value - BONITO64 always responds as target on the second clock.
24	**mperr_clr**	1	Initiator-noted parity error. Set when there's a parity error (whether noticed by us or signaled by a receiver) on one of our read/write operations, but only where the enabling **pcicmd** bit is set.
23		0	Fast back-to-back transactions - not supported
22		0	"User-definable features" - not supported
21		0	66MHz operation - not supported
20-16		0	Reserved by PCI specification rev 2.1

Table 5.10: PCI configuration space "Status" register

5.6.6. Latency timer

This 8-bit field in **pciltimer** sets the maximum period (in PCI clock cycles) for which BONITO64 will occupy the PCI bus[13] when it's the initiator of a transfer. The counter runs down to zero before firing, so a value of 7 leads to an 8-clock time limit. BONITO64 doesn't trouble to implement the low 3 bits of the register field, always starting the count with all-ones in those positions - the time doesn't need to be defined so accurately. That means the latency timer can be set only to values 8, 24, 40 etc. In most systems it will make sense to set the register to 32, for a maximum burst length of 40 cycles.

[13] PCI rules only require this to be enforced when the PCI initiator can see that it will not gain the bus again for a subsequent transfer (that is, it's bus grant signal has been deasserted). But BONITO64 applies the programmed limit to all transfers.

5.7. CPU access to PCI

The MIPS CPU maps all PCI-accessible registers and memory into its own address space as shown in Table 4.1 on page 13.

5.7.1. PCI address regions

The CPU's windows onto PCI space are mapped using a "window" register called **pcimap**, shown in Table 5.11.

pcimap register

Bit(s)	*Name*	*Value*	*Effect*
18	**pcimap_2**	0/1	Controls the 2Gbyte top-half-of-memory PCI region; it allows you to choose only whether this 2Gbyte window is to the high (1) or low (0) half of PCI address space.
17-12	**pcimap_lo2**	base6	Map the 64Mbyte regions marked "PCI_Lo" in the CPU's memory map, each of which can be assigned to any 64Mbyte-aligned region of PCI memory. The address appearing on the PCI bus consists of the low 26 bits of the CPU physical address, with the high 6 bits coming from the appropriate base6 field. Each of the three regions is an independent window onto PCI memory, and can be positioned on any 64Mbyte boundary in PCI space.
11-6	**pcimap_lo1**		
5-0	**pcimap_lo0**		

Table 5.11: Fields in **pcimap**

Note that the PCI I/O region is hard-wired to access the lowest 1Mbyte of PCI I/O space, and the "PCI_1.5" region is not mapped at all.

5.7.2. PCI reads

BONITO64 doesn't try to overlap PCI reads; the PCI controller waits for the data to return. If your MIPS CPU supports multiple outstanding reads, it might be able to go on to perform other non-PCI accesses or to post PCI writes (which will not be carried out until after the blocked read). You can either use the data you just loaded, or use the MIPS sync instruction if your program must wait until the load is really completed and the data returned.

Don't assume that the address presented on the PCI bus is exactly the same as that produced by the CPU pins - and since it's a MIPS CPU, remember that the address on the CPU pins is already different from the software address.

The MIPS CPU can perform 32-bit, aligned 16-bit, 8-bit and "tri-byte" single accesses to PCI space. The PCI byte enables reflect the width of the transfer; note that the relationship between the CPU width code and address on the one hand, and the PCI byte enables and lanes used on the other, depends on the endianness configuration - see section 4.2.

64-bit CPU accesses to PCI memory work too. Some CPUs automatically break the access into two separate cycles; otherwise BONITO64 will convert it into a 2-word PCI burst. This supports device code which maps PCI locations as normal memory, then does doubleword accesses. Burst accesses to PCI longer than 2 words may work, but will behave peculiarly in the face of PCI cycles which are terminated early, and are not recommended.

5.7.3. PCI writes

PCI writes are always posted; the address and data for the transaction are stored inside BONITO64 and the CPU interface is then released for the next transaction.

There's a relatively short FIFO for posted PCI cycles; when it fills up, the CPU interface will be stalled until a PCI write can be completed.

The CPU may need to check that a particular write has actually been performed on the PCI bus; do that by reading the **`pcimstat`** register[14], see page 38.

5.7.4. PCI error conditions on CPU-initiated cycles

If the PCI cycle finishes with any kind of error:

- The address of the failing cycle is stashed in the **`pciBadAddr`** register.
- Status bits for conditions like target aborts or master aborts (that's PCI-speak for "nobody responded") get set in the PCI-standard configuration space register fields - part of the **`pcicmd`** register and shown in Table 5.10 above.
- If the CPU is stalled waiting for a read it will receive a MIPS "bus error".

  But in any case an interrupt condition is indicated by pulsing the internal signal *mastererr*. The pulse is caught by the interrupt controller's latch and may be detected and cleared there - see §5.15.

One possible outcome of a PCI cycle is that the target terminates the cycle with a retry request; the master is supposed to just keep retrying until the transfer goes through. But that means BONITO64 can be deadlocked (more or less) if for some reason a defective target responds with retry forever. BONITO64 is therefore equipped with a retry counter; after 256 attempts at a read, or a much larger number of attempts at a write, the transaction is abandoned. The PCI error interrupt is raised, but no bit is set in **`pcicmdStat`**.

It's legal - within the PCI specification - for a target to signal "retry" to 256 consecutive reads and still recover later; so software can catch the condition and do more retries if appropriate. BONITO64's write-retry limit is set large enough that it's occurrence should be seen as fatal.

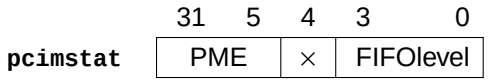


Figure 5.2 Fields in **`pcimstat`** *- PCI master status register*

The important field here is **`pcimstat.FIFOlevel`** which goes to zero when BONITO64 has no posted writes left queued. Software will want to read this field when it's vital to make sure that something out in PCI space has actually been written.

`pcimstat.PME` is for diagnostic and test software only, and may not be present in production units.

[14] This area of operation is in need of a rethink. The existing scheme might take an arbitrarily long time to resolve if the PCI interface is very busy.

5.7.5. Accessing PCI configuration space

PCI configuration cycles require specially-formatted values to be driven on the bus at address time. Figure 5.3 shows what bits have to be set on the PCI *AD* bus:

AD31 24	23 16	15 11	10 8	7 2	1	*AD0*
AD31-11 value - system dependent - to create correct *IDSEL*			Function Number	Register Number	0	0

Fields for “Type 0” cycles

AD31-24 value as needed to create correct *IDSEL*	Bus Number	Device Number	Function Number	Register Number	0	1

Fields for “Type 1” cycles

Figure 5.3 PCI address-time values for configuration cycles

To make configuration cycles happen you need to setup the register **`pcimap_cfg`**, shown in Figure 5.4:

	31 17	16	15 0
`pcimap_cfg`	×	Type1	AD16UP

Figure 5.4 Fields in **`pcimap_cfg`**

The fields are as follows:

- *Type1* : should be set 0 for normal configuration cycles, and to 1 for “type 1” cycles (required when configuring something on the other side of a PCI-to-PCI bridge).
- *AD16UP* : defines the bits written to *AD31-16* during a configuration cycle.

The result is that a word read/write with MIPS physical address *addr* in the range `0x01FE.8000` to `0x01FE.FFFC` causes a configuration read/write on PCI, and the PCI *AD* bus will be driven to:

31 16	15 2	1 0
`pcimap_cfg.AD16UP`	*addr*15-2	0 **`pcimap_cfg.Type1`**

5.7.6. Accessing PCI I/O space

PCI I/O space is available through a special region of CPU space. Its use is deprecated for most purposes, and it is generally only used to make devices PC-compatible. PCI I/O space writes are posted exactly like any other PCI accesses. BONITO64’s window only gives access to the first 1Mbyte of I/O space - but since this is more than enough to handle ISA bus legacy devices, that should be OK.

5.8. The PCI copier

This is an autonomous engine which shuffles blocks of data between PCI-accessible memory and local DRAM. The (potentially very long) stream will be interleaved with any CPU or PCI master transfers.

Each command can copy up to 64K 32-byte chunks (2Mbyte) of data; to copy more data, issue multiple commands. The copier registers are “double-buffered” so that the CPU can start a chain of transfers by issuing two commands; as soon as the first copy is finished BONITO64 immediately begins the second, raising an interrupt so the CPU can go in and write another new entry. Repeat as necessary to keep data flowing as fast as the PCI environment will permit.

Copier interrupts/status

Three internal signals *copyrdy*, *copyempty* and *copyerr* are fed from the copier to the interrupt controller (see §5.15. (“Interrupt control”) on page 45) and may be read there or used to cause an interrupt.

copyempty is high/“1” whenever the copier has no work at all to do, and *copyrdy* is high/“1” whenever the copier can accept one more request.

copyerr is generated only when the transfer encounters a PCI bus error; under these circumstances the copier freezes. At that point software can find out how much of the transfer has happened, and should then reset the copier to clear the error.

Copier address registers

You submit a copier command by writing the write-only **`copdaddr`**/**`coppaddr`** registers, which define the local DRAM starting address and the PCI starting address respectively. The DRAM starting address must be on a 32-byte boundary - use the CPU to copy any unaligned odds and ends. The PCI address in the copier is not mapped like CPU→PCI cycles, but is a PCI physical address. The PCI transfer can start at any word-aligned address.

Copier control/status registers

The other registers are shown in Figure 5.5 below. Write the direction and the number of blocks to copy into **`copgo`**. Finally write **`copctrl`** to start the transfer. Both are write-only

The only readable register in the copier is **`copstat`**; you won’t normally refer to it unless there’s some kind of error.

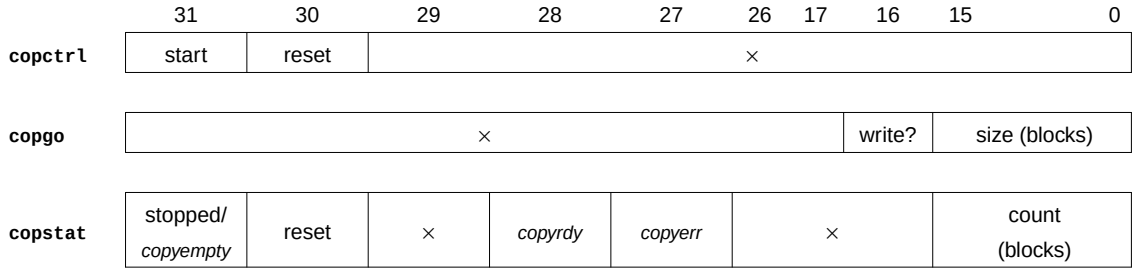


Figure 5.5 Fields in **`copctrl`**, **`copstat`**, **`copgo`**

Notes on fields shown in Figure 5.5:

- **`reset`**: set this bit to reset the copier - halt the current transfer and discard any pending entries (after completing any committed PCI transfer of not more than 8 words). This bit is set from system reset. You have to clear this bit before the copier will accept an entry.
- **`write`**: direction - “1” to transfer from DRAM to PCI.

- **`copgo.size`**: the number of 32-byte blocks to transfer.
- **`copstat.count`**: the number of 32-byte blocks remaining to be transferred in the request currently being processed. Note that this may not be related to the "**`size`**" field you just programmed, since your request may still be queued behind an earlier one. More precisely, only if *copyrdy* is active is **`copstat`** reporting on the transfer you last submitted.
- **`copctrl.start`**: set "1" when writing **`copctrl`** to submit an entry. In fact, there's not usually any reason to write **`copctrl`** except to submit an entry, so you will only ever write this bit zero when un-resetting the copier).
- **`copstat.stopped`**: is only interesting after you've started a transfer, when it should go to zero; it changes to a "1" when the copier stops - either because your transfer is complete, or because the copier encountered an error. It's in fact identical to the internal signal *copyempty* you can read through the interrupt controller.

5.9. PCI access to SDRAM and IOBC programming

PCI initiators can access local SDRAM through either of BONITO64's two programmable regions associated with the PCI base registers **`pcibase0-1`**. Either of these windows can be set to map the whole of the maximum possible SDRAM configuration of 256Mbytes[15].

PCI initiators access SDRAM through the *I/O Buffer Cache - IOBC* for short, described in 3.1 above. The IOBC keeps copies of local memory data in chunks which are the same size and alignment as CPU cache line blocks. All traffic between the local memory and the IOBC are cache-line-sized bursts at full memory speed.

You should make sure you've enabled the IOBC-related control bits **`bongencfg.cachestop`** (which must be zero); and **`bongencfg.uncached, bongencfg.wbehinden`** and **`bongencfg.prefetchen`** (which should all be 1).

As described in §5.6.3, an access window has associated flags and the. So if the first PCI access windows is to be used for bulk data transfer should have **`pcimembasecfg.membase0_cached`** set (see on p.34). You just might want to set up a window "uncached", if it's one where a PCI master device just reads and writes flags. Try it and see.

PCI initiator writes only result in a burst to memory after a whole "line" of data is provided (the writeback is usually triggered when the PCI transfer runs on to the succeeding line in memory).

5.10. Registers for IOBC management

Expect lots of changes to this section as coherent operation is prototyped and introduced.

Transfers happen in the basic 8-word storage unit, which is BONITO64's preferred transfer unit in and out of SDRAM. Each of the IOBC's four cache lines holds *two* 8-word chunks of data. See §3.1. ("The I/O buffer cache") on page 10 for an overall description of the cache.

The CPU may want to be able to either *invalidate* IOBC line (ensuring that any copies of memory data held in the line are discarded, and must be re-read from SDRAM if required), or *write-back* an IOBC line, causing any PCI-written data held in the line to be written back to SDRAM.

Typically, you want to writeback or invalidate any entry containing data in some particular range. While you could do that by blindly invalidating all the lines, that seems likely to be inefficient when there's a lot of PCI traffic - which is when it matters. So we recommend that you first look at the line to see whether it contains the data you're interested in, and only writeback/invalidate it if it does.

[15] Access to more than 256Mbytes SDRAM needs to be possible, but is not defined yet.

Commands on the IOBC are run by writing to **pcicachectrl**; poll the same register to check that the command has completed. Commands typically take less than a microsecond to run, so it's not worth waiting for an interrupt.

Where you need to read tag data from a cache entry the register **pcicachetag** is used as a staging point.

The control fields are shown in Table 5.12. The register fields are somewhat different when reading or writing it.

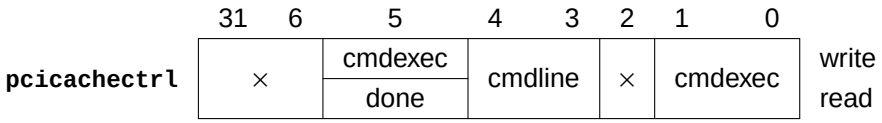


Table 5.12: PCI Cache control register

The fields are as follows:

- **cmdexec**: a value encoding what kind of action to carry out:

  0 Invalidate
  1 Write-back & invalidate
  2 Read tag
  3 No-op

  "No-op" is more useful than it looks; all actions except "invalidate" cause a command to be sent from the IOBC to the memory controller, and the IOBC to wait until it returns, so a "no-op" can be used to ensure that there are no older write-backs pending.

- **cmdline**: which of the four cache lines are being operated on.

After a "read tag" command for a particular line, the (read-only) **pcicachetag** register holds the line state formatted as in Table 5.13

pcicachetag register

Bit(s)	*Name*	*Meaning when "1"*
30	**wback**†	writeback pending
29	**pfpend**†	prefetch or write-behind pending
28	**pend**†	update pending.
27	**mod**	contains data written from PCI but not in memory yet
26	**pfdval**†	read pre-fetch data valid.
25	**dval**†	read data valid - this line has been read from local memory.
24	**aval**	address valid - this line is genuinely allocated to the block whose address follows.
23-0	**tagaddr28-5**	Local SDRAM address of data block being kept in this cache line.

Table 5.13: Fields in the pcicachetag register

To check whether a cache line is used in some particular buffer you should:

1. Issue a "read tag" command by writing **pcicachectrl**.
2. Wait for the command to be finished, by polling until **pcicachectrl.done** is set.

† The meaning of these fields is obscure, mostly for diagnostics, and may change. You shouldn't use them except for diagnostics.

3. Check that **`pcicachetag.aval`** is set (otherwise there's nothing valid in this line).
4. Check whether the 32-byte memory chunk whose address (divided by 32) starts at **`pcicachectrl.tagaddr`** overlaps with the memory region you're interested in.

You can obtain sample IOBC management routines for drivers to use from Algorithmics Ltd.

5.11. PCI access to local I/O

PCI hosts can read and write addresses on BONITO64's I/O bus, and do absolutely anything. Either PCI access window can be programmed to a base address which maps to I/O. This is useful for diagnostics and bootstrapping, but probably a really bad idea for running software.

5.12. Local I/O DMA control

BONITO64 provides a DMA facility on its local I/O bus, strongly - if not quite solely - orientated to the needs of an attached "IDE" bus. BONITO64 can read or write a stream of (16-bit) half-words on the *IOD* bus, collecting them up for transfer in or out of local SDRAM. Devices taking advantage of this facility must be able to use the *DMARQ*/*DMACK** signals as prescribed by IDE bus specifications and folklore. No "terminal count" signal is provided.

DMA cycle signaling must be IDE-compatible in DMA, MDMA or UDMA modes. Timing is programmable too - IDE cycle options are configured in **`iodevcfg`** register defined on p.27.

DMA is under the control of a set of registers, whose names all start with **`ldma`**-. To start a transfer:

- Make sure the DMA controller is out of reset: that requires that **`ldmactrl.reset`** is zero.
- Write the starting address in local SDRAM into **`ldmaaddr`**. Starting addresses must be 2-byte-aligned.
- Write the transfer count and direction into **`ldmago`**. It counts half-words (16-bit data). You can only transfer an odd count of half-words if you know that the DMA transfer will finish there; that is, you're not expecting it to chain.
- Write **`ldmaCtrl.start`** to one. Even so, nothing happens until and unless the device asserts *DMARQ*.
- Program your IDE-compatible device to do its thing and transfer data.

Like the copier, the DMA controller allows two transfers to be outstanding at any one time, and when the hardware finishes one transfer it will automatically proceed with the queued one ("chaining"). You can track its progress, or arrange to get interrupts, from the internal signals *dmardy*/*dmaempty*, which are wired to the interrupt controller. *dmardy* is high when the DMA controller could accept another entry (even if one is already in progress), and *dmaempty* is high when the DMA controller has finished all outstanding transfers.

The register **`ldmactrl`** is write/read, though it's normal to call it **`ldmastat`** for reading. **`ldmago`** is write-only. They're all shown in Figure 5.6.

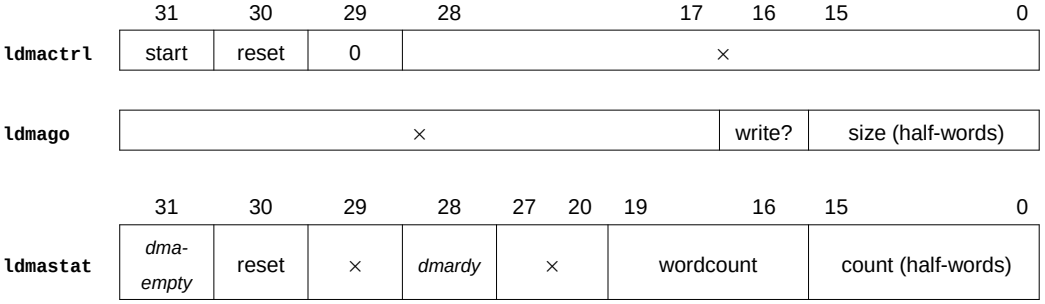


Figure 5.6 ldmastat and ldmaCtrl register layouts

The fields in the registers shown in Figure 5.6 are as follows:

- **`ldmactrl.reset`**: write a “1” here to reset the DMA subsystem, stop everything, and discard any queued entries. This bit is set “1” from system reset, and must be written zero to use DMA.
- **`ldmactrl.start`**: write a “1” to start a transfer - you must have set up the address first! In fact you only ever write a zero to this bit when resetting or un-resetting DMA.
- **`ldmastat.dmaempty`** reads “0” when you’ve started a DMA, but it hasn’t finished; and changes to “1” when the DMA completes.
- **`write`**: direction bit - set “1” for a transfer from SDRAM to the I/O bus.
- **`size`**: holds the transfer count as a number of half-words. When you read the **`ldmastat.size`** field it returns the current transfer count of the active entry; note that because you can queue a transfer request, this may not be the transfer you just programmed (it may be the one before).

5.13. PCI Mailbox registers

Four mailbox registers **`pcimail0-3`** may be read and written; 8 bits are implemented in each, and carry data between the parties. Any write to one of these registers pulses the corresponding internal ’signal’, which can be caught in the interrupt controller as an interrupt; you’ll need to program the interrupt controller to respond to a positive-going edge, and to clear down the stored interrupt when you’ve done with it. See §5.15 for details.

5.14. GPIO pins

Are programmed simply through a pair of bit-per-signal registers, one for read/writing each pin’s logic level, and one for controlling the direction of each signal[16].

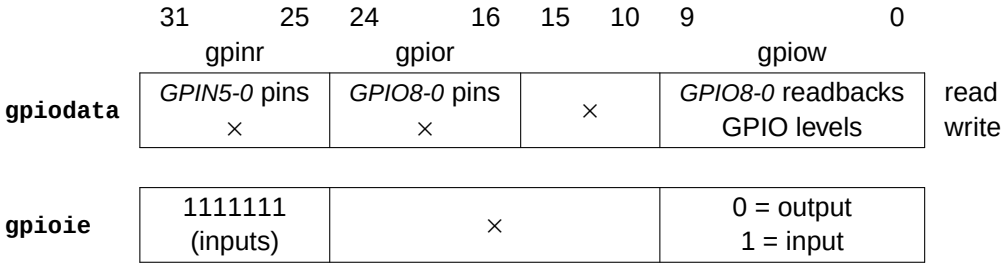


Figure 5.7 Fields in **`gpiodata`** *and* **`gpioie`**

The input-only *GPIN* pins are handled at the same time, though the corresponding **`gpioie`** bits (where writing a 1 makes the corresponding bit an input) are hard-wired to 1.

The *GPIO* pins appear twice in the read-data register; the high-order bits reflect the logic level at the pin, while the low-order bits is a readback from the *GPIO* register. The two will be different when the corresponding **`gpioie`** (input enable) bit is a 1, or may be different because of logic-level contention.

[16] The number of GPIO and GPIN pins, the split between them, and how many of them are available as interrupts is implementation-dependent. The numbers here are correct for both the Bonito32 ASIC and the FPGA BONITO64 at the time of writing.

5.15. Interrupt control

BONITO64 contains a simple, flexible interrupt controller. In addition to a number of input interrupts signaled through the *GPIn* and *GPIO* inputs, it also manages interrupts caused by internally-detected events.

All the interrupt registers have the same layout, in which each potential interrupt source has one bit:

`intXXX` register

Bit(s)	*Name*	*What are they?*
31	**`irqa`**	Only in **`inten`**, **`intenclr`** and **`intenset`**; doesn't really belong here; this controls the PCI interrupt *INTA** when it's configured as an output (see **`bongencfg`**). It's in the **`inten`** register because single bits can be changed by a single write.
30-25	**`gpins`**	The general-purpose input pins *GPIN5-0*.
24-20		Not connected
19-16	**`gpios`**	The general-purpose I/O pins *GPIO3-0*.
15		Not connected
14	**`IntTimer`**	A pulse produced by the internal reference timer while the timer value is equal to the "compare" value. It should be configured as "active high" and latched - see **`intpol`** and **`intedge`** respectively.
13	**`retryerr`**	A PCI cycle initiated by BONITO64 has been abandoned after too many retries. See §5.7.4. This field was called "**`pciMTimeout`**" in earlier versions of this manual.
12	**`dramperr`**	Parity error detected by SDRAM controller, when enabled.
11	**`systemerr`**	PCI error in cycle when BONITO64 is target.
10	**`mastererr`**	PCI error in cycle when BONITO64 is initiator.
9	**`pciirq`**	Active level on PCI interrupt pin *IRQA**.
8 7 6	**`copyerr`** **`copyempty`** **`copyrdy`**	Copier (see §5.8) internal signals, all reported as levels. *copyerr* indicates a PCI error on a copier transfer; the copier has stopped and and can only be restarted after a software reset. *copyempty* is asserted when the copier has finished all requested transfers. *copyrdy* is asserted as soon as it's possible to program another copier transfer.
5 4	**`dmaempty`** **`dmardy`**	DMA (see §5.12) internal signals, all levels. *dmaempty* means that all programmed DMA has finished, whereas *dmardy* just invites you to give it some more work.
3-0	**`mboxes`**	Pulsed when mailboxes are written (see §5.13. ("PCI Mailbox registers") on page 44). You need to configure these interrupts as active-high and edge-triggered.

Table 5.14: Fields in **`intXXX`** *registers*

The interrupts are configured for polarity and edge/level sensing with the following registers:

- **`intisr`** (read-only) has a bit set for any interrupt condition which is active - in the case of external interrupts configured as edge-sensitive, it returns the state of the internal latch.

When you're just using the signal as a program-readable input, you read its value here and leave the corresponding **inten** bit clear so it takes no further part in the interrupt system.

- **inten** (which may not be directly writable) has a bit set for any interrupt which is enabled. You most often manipulate the interrupt enables by writing either **intenset** which sets only **inten** bits corresponding to a "1" data bit, or **intenclr** which zeroes any bits corresponding to a "1" data bit.

  As a useful side effect, writing a "1" bit to **intenclr** also resets the edge-detecting latch of the corresponding interrupt.

- **intpol** sets the polarity of each interrupt source; a "1" bit for active-high, and a "0" bit for active-low. The inversion happens before the edge-detecting latch.

  Since **intpol** is forced to all-zeroes by a system reset, you need to program it to avoid getting spurious interrupts from active-high signals - and the "internal" interrupt signals are all active high.

- **intedge** can be used to program each external interrupt source from level (bit 0) to edge (bit 1). Effectively it does this by selecting either the direct pin input, or the state of the edge-detecting latch. The latch is still there, and its state is not affected by the programming of this register.

- **intsteer** causes an active, enabled interrupt condition to affect either of two outputs intended for MIPS CPU interrupt inputs: either *Int0** (corresponding bit 0), or *Int1** when the corresponding bit is set to 1. The interrupt output *Int1** can also appear on the PCI line *INTA** by setting the **bongencfg.irqa_from_int1** register bit, shown in Table 5.3 above.

Some BONITO64 implementations may not offer all options on all interrupts. As for the GPIO system, some register bits may become quietly read-only. For example, **intpol** bits corresponding to internal interrupt sources (whose activity level is always nominally high) will always read 1. Similarly, **intedge** bits corresponding to interrupt sources which must always be latched may always read 1.

Initialising the interrupt controller

To avoid spurious interrupts, you must initialise the ICU's **intpol** and **intedge** registers as part of system start-up. You should then disable all the interrupts by writing all-ones to **intenclr**, which has the side effect of making sure that all the edge-detecting latches are clear. Once that's done you can enable interrupts at the CPU, and each device driver can enable its own interrupt as required.

5.16. BONITO64 reference timer

This simple timer lacks the sophistication, ease of programming, range and precision of the on-CPU timer found on all compatible MIPS CPUs. So a running OS is unlikely to use this as a timer: but by connecting BONITO64's own timer to a fixed clock frequency, software can use it to calibrate the main system clock which is likely to change over product variants or lifetime - and that's very useful to some systems.

The timer itself can be fed by a clock from a number of different sources, to give you a choice of ways of wiring your system. You may be able to use your PCI bus clock if:

1. Your PCI bus clock is known to run always (the PCI spec permits it to be stopped);
2. Your PCI bus clock really runs at a constant and accurate rate. Note that popular PC-world clock synthesisers often provide different approximations to a 33MHz PCI bus clock at different settings.

More useful is probably an external clock, which can be wired to either of *GPIO7-8*; a good choice might be a clock from a network controller, or any fixed crystal-controlled frequency from your board.

When enabled, the timer counts up until it reaches the value programmed into the **timercfg.timercompare** field, from where it clears to zero on the next clock edge. While the counter value is equal to the compare value, an internal signal *IntTimer* is asserted and fed to the interrupt controller, where it can be seen, latched and used as an interrupt.

`iodevcfg` register

Bit(s)	*Name*	*Value*	*Effect*
31-25			Not used
24			Reserved
23	**`timerenable`**	`0/1`	Set “1” to allow timer to free-run - otherwise it runs to zero and stops.
22	**`timerrefclksrc`**	`0` `1`	External clock (if selected) is taken from *GPIO7*. External clock (if selected) is taken from *GPIO8*.

21-20	**`timerclocksrc`**	`1`
		`2`
		`3`
			Clock every second edge of *ClockIn*.
			Run timer from the PCI clock *CLK*.
			Run timer from one of *GPIO7-8*; which of them is used depends on the setting of **`timercfg.timerrefclksrc`**, described above.

19-0	**`timercompare`**	`0-1M`	Write the “compare” value here. When timer reaches this number, it wraps around to zero and generates a 1-clock pulse on the onchip signal *IntTimer* fed to the interrupt controller.

6. Hardware description

This section describes BONITO64 as implemented in a Xilinx “Virtex” FPGA - the XCV400E6FG676, and with the pinout used in Algorithmics’ P−6064 prototyping board.

This FPGA can be configured to match a range of I/O standards, but not old-fashioned 5V TTL - the high levels permitted on 5V TTL are liable to damage it if enough current flows. On the prototyping boards pins connected to 5V signals are defended either with voltage-limiting “Quick-Switch” gates or with series resistors.

6.1. Signals

The signals on this chip are as follows:

Signal Name	*Type*	*Description*
CPU interface signals		
SysAD0-63	Bi	MIPS multiplexed bus, parity check bits (*SysADC*), and transfer type code (*SysCmd*). Note that Vr4300 systems and others which make no use of parity won’t connect *SysADC*.
SysADC0-7		
SysCmd0-8		
*ValidIn**	Out	Pulsed when BONITO64 is driving the CPU bus.
*ValidOut**	In	Pulsed when CPU is driving the CPU bus
*WrRdy**	Out	CPU write cycle flow control. MIPS CPUs usually have a separate *RdRdy** pin, which should be strapped permanently active.
*Release**	In	Pulsed as CPU stops driving the bus in a read cycle
CPUClock	In	master clock for BONITO64 which must be precisely aligned with the CPU’s input clock. SysReset*:In:T{ Reset for BONITO64 and perhaps other circuits. Often connected to the active-high power-good signal from a power supply.
*CPUColdReset**	out	Controls for CPU’s two-stage reset sequence.
*CPUReset**		
VCCOk	Out	Some MIPS CPUs use this (active-high) signal in their reset sequence. For CPUs without such an input, it can be ignored. **Warning**: designated as an input in versions of this spec up to and including v2.1.
ModeClock	In	Clock from some CPUs, used to shift out “mode bits” to select reset options. The mode bits themselves are fetched from the boot ROM, and are presented in turn on *ModeIn*.
ModeIn	out	
*Int0-1**	Out	BONITO64’s interrupt lines to CPU
*NMI**	Out	MIPS non-maskable interrupt. For strange debug purposes only; please ignore.
*PAck**	Out	Extra CPU interface signals for RM70xx CPU - see CPU manual if you’re interested. May be recycled to other extra-pin sets for CPUs with slightly different buses.
*PReq**	In	
RdType	In	
RspSwap˜	In	
TcMatch	In	
TcTCE˜	In	
PCI interface signals		

Signal Name	*Type*	*Description*
CLK	In	PCI bus clock, 33MHz nominal
AD0-31	Bi	PCI address/data plus parity
PAR		
*CBE0-3**		PCI cycle type (address time) and active-low byte enables (data time)
*DEVSEL**	Bi	PCI cycle control signals
*FRAME**		
*IRDY**		
*STOP**		
*TRDY**		
*PERR**	Bi	PCI parity reporting signal. A parity error during one of our cycles causes an interrupt or bus error to be returned to the MIPS CPU.
*SERR**	Bi	PCI general error reporting signal. Can generate an interrupt to the MIPS CPU.
*RESET**	Bi	PCI bus reset. Can be configured as an output, normally when BONITO64 is responsible for host functions on the PCI bus; in this mode it is asserted from system reset and subsequently controlled by software. When active, the PCI bus interface is held in reset, and no PCI shared signals are driven.
IDSEL	In	Marks incoming configuration cycles
*IRQA**	Bi	Can either be configured as an input (to BONITO64's interrupt controller) or made an open-collector output, which is then controlled by writing the **`intenclr`**/**`intenset`** registers.
*REQ0-5**	In	When BONITO64 is serving as PCI bus arbiter, these are the PCI request/grant signals for use by other potential PCI initiators. When we're using an external arbiter, *GNT0** acts as BONITO64's PCI *request*, and *REQ0** acts as its *grant* (swapping roles means we can leave some signals as pure inputs). Note that BONITO64's arbiter is enabled/disabled by a software-writable register bit **`bonponcfg.is_arbiter`**, but it can be preset on as described in §5.2. ("Power-on settable configuration register - **`bonponcfg`**") on page 23.
*GNT0-5**	Out	
SDRAM interface signals		
MUXAD0-13	Out	Multiplexed addresses.
DBA0-2	Out	Bank select - up to three additional address lines into SDRAM components. Most use one or two of them.
DD0-63	Bi	SDRAM data bus. During ROM or I/O accesses these signals act as an address bus (but they don't count up during "burst" ROM cycles - you need to connect your ROM to *IOA0-4* for that.)
DDP0-7		Parity/check bits for data bus
DQS0-8	Bi	Source-provided strobes for DDR data transfer, multiplied up so as to provide an independent signal per byte lane - one per active chip if the 72-bit wide memory is built from ×8 components.

Signal Name	*Type*	*Description*
*DRAS**	Out	SDRAM cycle control signals: RAS, CAS, write-enable and clock enable. Typically common to all DRAMs. Clock-enable duplicated for fan-out reasons.
*DCAS**		
*DWE**		
DCKE0-1		
*DCS0-1H**	Out	DRAM chip selects for four “sides” (physical banks) of SDRAM.
*DCS0-1L**		
DQMBLo0-1	Out	Byte “masks” - that is, byte enables. Used in groups of four as word enables for the 32-bit halves of the SDRAM. Writes less than 32-bits to memory are implemented inside BONITO64 using a read-modify-write cycle. Each signal is attached to four of the byte masks going into the DIMM.
DQMBHi0-1		
I/O and ROM interface signals		
IOD0-15	Bi	Separate data bus enables some I/O transactions (particularly DMA) to be completed without using the SDRAM signals. Note these signals are inputs while *SysReset** is active, and are then used to make pre-reset chip configuration choices. In some configurations, I/O addresses (*IOA5-20*) are multiplexed on these pins.
IOD16-31	Bi	Data bus extension found on some parts configured to work with wide ROM system.
IOA0-4	Out	CPU address bits 0-4, valid during I/O (including ROM) cycles. *IOA0-4* count during ROM bursts, and are thus the only correct signals to use for low ROM address bits.
IOAHI_CLK	Out	Demultiplexing signal when obtaining I/O address from *IOD0-15*. It’s suitable for use either with a transparent latch (’373 or similar), or it can work as an active-high clock enable valid with reference to the CPU/SDRAM clocks.
Isolate	Out	High to isolate ROM signals from the high-speed SDRAM data bus. Suitable for use as input to a QS3245 or similar switch. Also usable as an enable for a buffer for address-time signals on the local I/O data bus.
*RomCS0-1**	Bi	Chip selects for 2 memory devices - often one ROM socket (for first-time bootstrap) and one flash ROM - and some I/O devices. If you need more ROM chip selects, you can generate them by qualifying *ROMCS1** with some high address bits - which during ROM cycles are available on *DD0-31*. The ROM timings are sloppy enough to give you 10-15ns to do this while still maintaining setup time before the *IORD**/*IOWR** strobe. *ROMCS1** is the “default” bootstrap region, and the natural place for your standard bootstrap memory. The levels on *ROMCS0-1** are sampled at reset-time into the **`bonponcfg`** register, and you could use a 10K pullup or its lack that for a software-readable link.
*IOCS0-3**	Out	
IODIR	Out	Direction control (high for write) suitable for a ’245 buffer used to buffer the data bus - most often used to buffer IDE disk connection.

Signal Name	*Type*	*Description*
*IODEN**	Out	Enable control for an *IOD* data buffer - most often used to buffer IDE disk connection.
*IORd**	Out	Intel-style read and write strobes for ROM and I/O. Addresses and chip selects have enough setup and hold time from the active strobes to allow external logic to generate more chip selects from the addresses.
*IOWr**	Out	
IORDY	In	These two signals are usually connected together; if you configure an I/O bus DMA mode to support UDMA IDE, this signal becomes a source-synchronous clock for DMA read data, and at high speed it's essential to wire them to one of the FPGA's "global clock" pins - called *UDMA_DSTROBE*. However "I/O channel ready" has another role in slower I/O cycles, where it's an active-low request for extra wait states. If you don't use it in this role, you must make sure this signal is always high during I/O cycles.
UDMA_DSTROBE		
I/O bus DMA and IDE support		
DMARQ	In	DMA peripheral is ready to transfer data.
*DMACK**	Out	DMA acknowledge "decode" distinguishing I/O bus reads or writes for DMA (with no chip select).
Programmable IO signals		
GPIO0-8	Bi	General purpose programmable I/O. Note that *GPIO0-3* (only) are usable as interrupt inputs, and *GPIO7-8* (only) may be used as clock inputs for the reference timer described in §5.16. *GPIn8* may also be used to supply a modebit stream to feed options into a MIPS CPU, when you need to have full control over more than 32 configuration bits.
GPIn0-5	In	Interrupt/General purpose input pins. These signals are weakly pulled down and with a link to *VDD* can be used to implement software-readable link or switch settings.
Test and dedicated configuration signals		
*SysController**	In	Dedicated configuration signal. If it's low, BONITO64 drives the PCI *Reset** signal; if it's high, PCI *Reset** becomes an input and itself resets all functions in BONITO64. Unlike all other BONITO64 configuration signals, it must be stable at all times.
*enable_outputs**	in	When inactive, float everything. Board test feature.
MOD_TYPE0-3	In	Available to allow the FPGA to be configured to support different CPU interfaces. On Algorithmics' evaluation board the levels on these signals are typically determined by presets on each variant CPU daughterboard.
FPGA_SPARE0-15	Bi	Signals unused in standard logic; on Algorithmics' prototyping board these are brought out to a header for use when experimenting with different FPGA logic programs.
JTCK	In	JTAG boundary test pins.
JTDI	In	
JTDO	Out	
JTMS	In	
Power and generic signals		

Signal Name	*Type*	*Description*
GND		System ground
V1V25		1.25V reference signal for DDR SDRAM (SSTL2) interface signals.
V1_8V		1.8V power, used for most of FPGA logic
VDD		3.3V power used for most I/Os. Note that the XCV400E part is not 5V tolerant, which makes the evaluation boards more complicated.
FPGA-specific signals		
FPGA_CCLK	In	Signals used to program the FPGA (which is a soft device which receives its logic “program” when the system is reset). Some of them can be reconfigured for I/O, but on Algorithmics’ evaluation boards these signals are not used for any other purpose.
FPGA_CS˜		
FPGA_D1-7		
FPGA_DIN_D0		
FPGA_DONE		
FPGA_DOUT_BUSY		
FPGA_DXN		
FPGA_DXP		
FPGA_INIT˜		
FPGA_M0-2		
FPGA_PROGRAM˜		
FPGA_WRITE˜		
HWDEBUG_CAP_CLK		Xilinx FPGA logic debugging interface. Available as a test header on Algorithmics’ evaluation board.
HWDEBUG_CAP_EN		
HWDEBUG_RST		
HWDEBUG_TRIG		
IOxx		You’ll see these names on the pin-out diagram, marking where the FPGA has unused I/O pins. Treat as NC.

6.2. Pinout

Currently the 676-pin FPGA is pinned out like this: this is the pinout as seen looking at the PCB pads. See below for signal name abbreviations - essential to get it on the page.

A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	A16	A17	A18	A19	A20	A21	A22	A23	A24	A25	A26
GND	NC	NC	MUXAD2	DQMBL0	DWE~	DCS0~h	IO34	NC	NC	MDTYP1	V1V25	NC	XSPR12	NC	NC	DMARQ	GPIO6	GPIN1	IORDY	IOD6	IOD8	IOD13	NC	NC	GND
B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15	B16	B17	B18	B19	B20	B21	B22	B23	B24	B25	B26
NC	GND	MUXAD9	NC	DBA1	NC	NC	DQMBL1	GND	IO39	NC	NC	IO60	GND	NC	NC	NC	GND	GPIN3	IOD1	NC	IOD11	IOD15	NC	GND	NC
C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	C18	C19	C20	C21	C22	C23	C24	C25	C26
NC	NC	GND	MUXAD8	MUXAD4	MUXAD0	DCS0~L	NMI~	DBA2	IO42	V1V25	SRest~	CLK	XSPR11	XSPAR8	XSPAR5	GPIO0	GPIO5	GPIO8	IOD3	IOD10	XWRIT~	JTDO	GND	NC	NC
D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15	D16	D17	D18	D19	D20	D21	D22	D23	D24	D25	D26
NC	DQMBH1	MUXD11	GND	MUXAD6	V1V25	DBA0	DCAS~	DCS1~h	IO38	IO46	MDTYP3	NC	XSPR13	XSPAR6	IO79	GPIO3	GPIN2	GPIN4	IOD7	IOD14	JTDI	GND	XCCLK	NC	NC
E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12	E13	E14	E15	E16	E17	E18	E19	E20	E21	E22	E23	E24	E25	E26
DQS0	DD1	MUXD13	MUXD12	GND	JTCK	MUXAD3	V1V25	DCS1~L	IO35	IO43	MDTYP2	GCLK3	XSPR15	XSPAR7	MdClck	GPIO1	GPIN5	IOD5	IOD12	XCS~	GND	XDTBSY	AD31	FRAME~	AD25
F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18	F19	F20	F21	F22	F23	F24	F25	F26
DD5	NC	DD3	V1V25	JTMS	NC	MUXAD7	MUXAD1	DRAS~	IO32	IO40	MDTYP0	XSPR14	IO61	XSPAR4	GPIO2	GPIO4	IOD0	IOD9	NC	NC	XDIND0	AD27	AD28	NC	CBE1
G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	G11	G12	G13	G14	G15	G16	G17	G18	G19	G20	G21	G22	G23	G24	G25	G26
DD7	DD8	DD10	V1V25	DQMBH0	DCKE0	V1_8V	MUXAD5	MUXD10	V1V25	IO37	IO45	EnOut~	XSPAR9	XSPAR3	GPIO7	GPIN0	IOD4	IOD2	V1_8V	AD30	AD29	AD26	CBE3	AD23	AD3
H1	H2	H3	H4	H5	H6	H7	H8	H9	H10	H11	H12	H13	H14	H15	H16	H17	H18	H19	H20	H21	H22	H23	H24	H25	H26
NC	DD12	DD13	DD4	DD2	DD0	DCKE1	V1_8V	Vbnk2	Vbnk2	Vbnk2	Vbnk2	SCtrl~	XSPR10	VDD	VDD	VDD	VDD	V1_8V	PAR	IRDY~	CBE2	AD24	AD21	AD17	NC
J1	J2	J3	J4	J5	J6	J7	J8	J9	J10	J11	J12	J13	J14	J15	J16	J17	J18	J19	J20	J21	J22	J23	J24	J25	J26
NC	GND	DQS2	DD14	DD9	DQS1	DD6	Vbnk2	V1_8V	V1_8V	V1_8V	Vbnk2	Vbnk2	VDD	VDD	V1_8V	V1_8V	V1_8V	VDD	AD4	CBE0	AD22	AD20	AD19	GND	STOP~
K1	K2	K3	K4	K5	K6	K7	K8	K9	K10	K11	K12	K13	K14	K15	K16	K17	K18	K19	K20	K21	K22	K23	K24	K25	K26
DD19	DD21	DD18	DD15	DD16	V1V25	DD11	Vbnk2	V1_8V	GND	GND	GND	GND	GND	GND	GND	GND	V1_8V	VDD	AD18	AD2	XD2	AD16	XD1	AD1	NC
L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L11	L12	L13	L14	L15	L16	L17	L18	L19	L20	L21	L22	L23	L24	L25	L26
NC	NC	DQS3	DD20	DD17	DD22	DD23	Vbnk2	V1_8V	GND	GND	GND	GND	GND	GND	GND	GND	V1_8V	VDD	TRDY~	DEVSL~	IDSEL	AD15	AD13	NC	AD12
M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14	M15	M16	M17	M18	M19	M20	M21	M22	M23	M24	M25	M26
NC	V1V25	DD25	DD24	V1V25	DD26	DD27	Vbnk2	Vbnk2	GND	GND	GND	GND	GND	GND	GND	GND	VDD	VDD	AD14	AD11	XD3	AD0	REQ4	NC	REQ3
N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12	N13	N14	N15	N16	N17	N18	N19	N20	N21	N22	N23	N24	N25	N26
NC	GND	DD30	DD29	IO42!	DD31	DD28	DQS8	Vbnk2	GND	GND	GND	GND	GND	GND	GND	GND	VDD	REQ5	REQ2	REQ0	IO180	GNT5	REQ1	NC	NC
P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19	P20	P21	P22	P23	P24	P25	P26
NC	NC	IO420	DDP0	DDP1	DDP2	DDP3	DDP4	Vbnk2	GND	GND	GND	GND	GND	GND	GND	GND	VDD	IOCS1	IOCS0	IOWr~	IOCS3	IORd~	IO181	GND	NC
R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24	R25	R26
V1V25	NC	DDP6	V1V25	DDP5	DDP7	DD32	Vbnk2	Vbnk2	GND	GND	GND	GND	GND	GND	GND	GND	VDD	VDD	IOA2	ROMCS1	ROMCS0	IOA4	XD4	IOCS2	NC
T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18	T19	T20	T21	T22	T23	T24	T25	T26
NC	NC	DQS4	DD33	DD40	DD35	DD38	Vbnk2	V1_8V	GND	GND	GND	GND	GND	GND	GND	GND	V1_8V	VDD	TcMtch	IODEN~	IOA1	Isolat	IOA3	NC	NC
U1	U2	U3	U4	U5	U6	U7	U8	U9	U10	U11	U12	U13	U14	U15	U16	U17	U18	U19	U20	U21	U22	U23	U24	U25	U26
DD34	DD36	DD37	DD39	DQS5	V1V25	DD46	Vbnk2	V1_8V	GND	GND	GND	GND	GND	GND	GND	GND	V1_8V	VDD	DMACK~	GNT4	ModeIn	XD5	IOA0	IODIR	NC
V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16	V17	V18	V19	V20	V21	V22	V23	V24	V25	V26
NC	GND	DD41	DD44	V1V25	DD49	DD55	Vbnk2	V1_8V	V1_8V	V1_8V	Vbnk1	Vbnk1	Vbnk1	Vbnk1	V1_8V	V1_8V	V1_8V	VDD	AD8	AD10	GNT0	GNT2	XD6	GND	IOHCLK
W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15	W16	W17	W18	W19	W20	W21	W22	W23	W24	W25	W26
DD45	DD43	DD42	DQS6	DD51	DD58	XM1	V1_8V	Vbnk1	Vbnk1	Vbnk1	Vbnk1	SADC1	SAD35	Vbnk1	Vbnk1	Vbnk1	Vbnk1	V1_8V	IO236	XSPAR1	DbgCPC	AD5	GNT1	GNT3	PERR~
Y1	Y2	Y3	Y4	Y5	Y6	Y7	Y8	Y9	Y10	Y11	Y12	Y13	Y14	Y15	Y16	Y17	Y18	Y19	Y20	Y21	Y22	Y23	Y24	Y25	Y26
NC	DD48	DD47	DD53	DD56	DD59	V1_8V	XDXP	SAD1	SAD13	SAD20	SAD27	NC	SAD32	SAD38	SAD47	SAD55	Valdt~	Int1	V1_8V	XINIT~	DbgCPN	IRQA~	AD6	SERR~	AD9
AA1	AA2	AA3	AA4	AA5	AA6	AA7	AA8	AA9	AA10	AA11	AA12	AA13	AA14	AA15	AA16	AA17	AA18	AA19	AA20	AA21	AA22	AA23	AA24	AA25	AA26
DD50	NC	DD52	DD57	DD63	NC	SCmd5	SCmd0	SAD6	SAD15	SAD21	SADC0	SADC4	UDStb	SAD36	SAD44	SAD50	SAD62	RSwap~	CdRst~	NC	XPrg~	DbgTRG	RESET~	NC	AD7
AB1	AB2	AB3	AB4	AB5	AB6	AB7	AB8	AB9	AB10	AB11	AB12	AB13	AB14	AB15	AB16	AB17	AB18	AB19	AB20	AB21	AB22	AB23	AB24	AB25	AB26
DD54	DQS7	V1V25	DD60	GND	XM2	XDXN	SCmd3	SAD8	SAD18	SAD26	SAD29	CClock	SADC6	SAD34	SAD42	SAD48	SAD57	WrRdy~	RdType	XDONE	GND	XD7	DbgRST	XSPAR0	XSPAR2
AC1	AC2	AC3	AC4	AC5	AC6	AC7	AC8	AC9	AC10	AC11	AC12	AC13	AC14	AC15	AC16	AC17	AC18	AC19	AC20	AC21	AC22	AC23	AC24	AC25	AC26
NC	DD61	DD62	GND	SCmd7	SCmd2	SAD3	SAD4	SAD11	SAD22	SAD24	SAD30	SADC2	IO300	SAD39	SAD41	SAD49	SAD52	SAD60	SAD63	Int0	IO241	GND	IO237	NC	IO235
AD1	AD2	AD3	AD4	AD5	AD6	AD7	AD8	AD9	AD10	AD11	AD12	AD13	AD14	AD15	AD16	AD17	AD18	AD19	AD20	AD21	AD22	AD23	AD24	AD25	AD26
NC	NC	GND	XM0	SCmd4	SAD2	SAD12	SAD16	SAD19	SAD25	SAD28	SAD31	SADC5	SADC7	SAD37	SAD40	SAD43	SAD53	SAD56	SAD59	Valdn~	TcTCE~	CRest~	GND	NC	IO242
AE1	AE2	AE3	AE4	AE5	AE6	AE7	AE8	AE9	AE10	AE11	AE12	AE13	AE14	AE15	AE16	AE17	AE18	AE19	AE20	AE21	AE22	AE23	AE24	AE25	AE26
NC	GND	NC	SCmd8	SAD0	NC	SAD7	SAD14	GND	SAD23	NC	NC	GND	NC	NC	NC	SAD46	GND	SAD54	SAD61	NC	PReq~	VCCOk	NC	GND	NC
AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15	AF16	AF17	AF18	AF19	AF20	AF21	AF22	AF23	AF24	AF25	AF26
GND	NC	NC	SCmd6	SCmd1	SAD5	SAD9	SAD10	SAD17	NC	NC	SADC3	IO301	NC	SAD33	NC	SAD45	NC	SAD51	SAD58	PAck~	Reles~	NC	NC	NC	GND

Figure 6.1 676-pin FPGA pinout for BONITO64

Signal name abbreviations used in pinout table

Originally	*Abbrev*	*Originally*	*Abbrev*	*Originally*	*Abbrev*
CPUClock	CClock	CPUColdReset˜	CdRst˜	CPUReset˜	CRest˜
DEVSEL˜	DEVSL˜	DQMBHi0-1	DQMBH0-1	DQMBLo0-1	DQMBL0-1
FPGA_CCLK	XCCLK	FPGA_CS˜	XCS˜	FPGA_D1-7	XD1-7
FPGA_DIN_D0	XDIND0	FPGA_DONE	XDONE	FPGA_DOUT_BUSY	XDTBSY
FPGA_DXN	XDXN	FPGA_DXP	XDXP	FPGA_INIT˜	XINIT˜
FPGA_M0-2	XM0-2	FPGA_PROGRAM˜	XPrg˜˜	FPGA_SPARE0-15	XSPAR0-9,XSPR10-15
FPGA_WRITE˜	XWRIT˜	HWDEBUG_CAP_CLK	DbgCPC	HWDEBUG_CAP_EN	DbgCPN
HWDEBUG_RST	DbgRST	HWDEBUG_TRIG	DbgTRG	IOAHI_CLK	IOHCLK
Isolate	Isolat	MOD_TYPE0-3	MDTYP0-3	MUXAD10-13	MUXD10-13
ModeClock	MdClck	Release˜	Reles˜	RspSwap˜	RSwap˜
SysAD0-63	SAD0-63	SysADC0-7	SADC0-7	SysCmd0-8	SCmd0-8
SysController˜	SCtrl˜	SysReset˜	SRest˜	TcMatch	TcMtch
UDMA_DSTROBE	UDStb	VDD_FPGA1-2	Vbnk1-2	ValidIn˜	Valdn˜
ValidOut˜	Valdt˜	enable_outputs˜	EnOut˜		

6.3. Package information

BONITO64 is implemented with a Xilinx “Virtex-E” part - the XCV400E6FG676, and you can get physical design information from `http://www.xilinx.com/partinfo/pkgs_pdf/fg676.pdf`. Xilinx’ drawing shows the bottom of the chip (caution: Algorithmics’ pin-layout drawing in Figure 6.1 above shows the PCB pad layout, as if looking “through” the mounted chip.

It’s an 1mm pitch BGA, a fully-populated 26×26 square (the inner pads are all power and ground, and are mostly provided for heat dissipation). Fine pitch BGAs can present a challenge to PCB layout and routing, and Xilinx have an application note on their web page about this: see `http://www.xilinx.com/xapp/xapp157.pdf` or browse the application notes master page `http://www.xilinx.com/apps/virtexapp.htm`.

Appendix A: Register Addresses

Registers in address order

register	*address*
pcidid	1FE00000
pcicmd	1FE00004
pciclass	1FE00008
pciltimer	1FE0000C
pcibase0	1FE00010
pcibase1	1FE00014
pcibase2	1FE00018
pciexprbase	1FE00030
pciint	1FE0003C
bonponcfg	1FE00100
bongencfg	1FE00104
iodevcfg	1FE00108
sdcfg	1FE0010C
pcimap	1FE00110
pcimembasecfg	1FE00114
pcimap_cfg	1FE00118
gpiodata	1FE0011C
gpioie	1FE00120
intedge	1FE00124
intsteer	1FE00128
intpol	1FE0012C
intenset	1FE00130
intenclr	1FE00134
inten	1FE00138
intisr	1FE0013C
pcimail0	1FE00140
pcimail1	1FE00144
pcimail2	1FE00148
pcimail3	1FE0014C
pcicachectrl	1FE00150
pcicachetag	1FE00154
pcibadaddr	1FE00158
pcimstat	1FE0015C
timercfg	1FE00160
ldmactrl	1FE00200
ldmastat	1FE00200
ldmaaddr	1FE00204
ldmago	1FE00208
copctrl	1FE00300
copstat	1FE00300
coppaddr	1FE00304
copdaddr	1FE00308
copgo	1FE0030C

Registers in name order

register	*address*
bongencfg	1FE00104
bonponcfg	1FE00100
copctrl	1FE00300
copdaddr	1FE00308
copgo	1FE0030C
coppaddr	1FE00304
copstat	1FE00300
gpiodata	1FE0011C
gpioie	1FE00120
intedge	1FE00124
inten	1FE00138
intenclr	1FE00134
intenset	1FE00130
intisr	1FE0013C
intpol	1FE0012C
intsteer	1FE00128
iodevcfg	1FE00108
ldmaaddr	1FE00204
ldmactrl	1FE00200
ldmago	1FE00208
ldmastat	1FE00200
pcibadaddr	1FE00158
pcibase0	1FE00010
pcibase1	1FE00014
pcibase2	1FE00018
pcicachectrl	1FE00150
pcicachetag	1FE00154
pciclass	1FE00008
pcicmd	1FE00004
pcidid	1FE00000
pciexprbase	1FE00030
pciint	1FE0003C
pciltimer	1FE0000C
pcimail0	1FE00140
pcimail1	1FE00144
pcimail2	1FE00148
pcimail3	1FE0014C
pcimap	1FE00110
pcimap_cfg	1FE00118
pcimembasecfg	1FE00114
pcimstat	1FE0015C
sdcfg	1FE0010C
timercfg	1FE00160

Appendix B: BONITO64's debug interface

When you set debug mode in BONITO64, controlled by **`bongencfg.debugmode`**

- The address for all CPU cycles, and all PCI cycles accessing BONITO64 local memory or registers, are driven out on the SDRAM data bus at the start of what looks like a SDRAM read/write.
- Cycle data is presented on the memory data bus at what looks like SDRAM data time.
- A few other SDRAM control signals are taken over for other purposes.
- When the cycle being reported is not really an SDRAM cycle no DRAM module/side select is active.

  A part of the memory map which would have selected the SDRAM bank which the debug board replaces - decoded by one of the top chip selects *DCS1H**/*DCS1L** - is available to access 8-bit devices on the debug board. However, Algorithmics standard 64-bit DIMM debug board doesn't provide any debug board I/O, and customers should definitely ask before relying on the existence of such a feature.

Debug mode is somewhat intrusive, and can alter system timings because I/O and PCI register accesses are reported on the debug interface, and that can hold up DRAM accesses. More precisely:

- DRAM cycles run the same way they always did;
- any IO cycle that needs to take some addresses from the DRAM data bus is unaffected (effectively, debug mode is always on for these cycles).
- With debug mode on all other IO cycles will use the SDRAM bus to report themselves. In particular this means accesses to Bonito's internal registers and CPU PCI reads/writes become visible.

How debug cycles work

The debug protocol provides address information (mostly) on the SDRAM data bus in the first cycle when *Ras** is asserted - which is well before any data is transferred on a real SDRAM read or write. (Remember, the debug protocol is active on SDRAM cycles as well as all others). The debug board sees data too; real SDRAM cycles have data, of course, but debug-only cycles deliver data using the *Cas** strobe.

The debug board has a set of registers which capture the SDRAM bus in two stages - typically the first-captured value is the address, and the last-captured the data. However, because the debug board uses programmable logic devices you can't see the registers...

Algorithmics' DIMM debug board

The standard 64-bit BONITO64 debug board is designed to plug into a 168-pin DIMM socket (as configured for standard PC synchronous memory: 3.3V un-buffered type). The debug board provides you with logic analyser connectors which present all the cycles which pass through BONITO64. The debug board buffers and re-registers the buses, so that even a relatively slow analyser will have no trouble following BONITO64's buses up to 83MHz. The connectors are pinned out to allow HP logic analyser "mass terminator pods" to be plugged straight in, but can be wired pin-by-pin to any kind of analyser.

Figure B.1 shows the layout of the board.

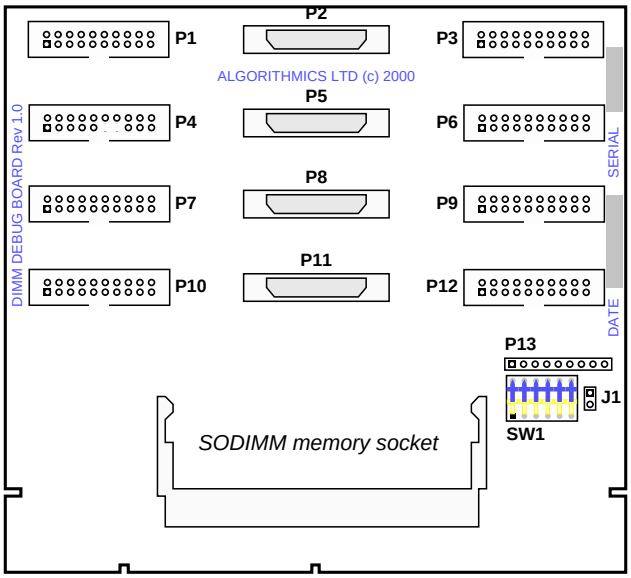


Figure B.1

6.3.1. Debug board analyser connectors

The signals available on the standard analyser connectors of the debug board are summarised in Table B.1.

Signal	*Description*
A31-0	Address of cycle
AM13-12	Raw value of SDRAM multiplexed addresses
ATRIG	Rising-edge trigger to capture cycle
BE7-0˜	independent active-low byte enables - *BE0˜* selects *D7-0* etc.
D63-0 DP7-0	Data bus and byte-wide parity
SIZE2-0	Encodes size of current transfer - a value of n means n+1 bytes
SRC1-0	tells you who is initiating the transfer: 0 → CPU initiated DRAM cycle. 1 → IDE DMA or PCI copier initiated DRAM cycle. 2 → PCI initiated DRAM cycle. 3 → IO or CPU -> PCI cycle
WR	High if this transfer is a write.

Table B.1: Signals available from debug board

Connector pin-outs

The DIMM debug board can have high-density “Mictor” connectors as used by newer HP analysers - but they’re hard to buy in the US, so are not always fitted. Let us know if they’re important to you.

Most of you will therefore connect into the 20-pin 0.1” dual headers as used on older and cheaper analysers - and they’re also usable if your test equipment needs to be attached pin-by-pin.

If you are using an HP or compatible analyser with mass terminator probes which plug right into the

connectors,then the signals get laid out for your convenience[17] as shown in Figure B.2.

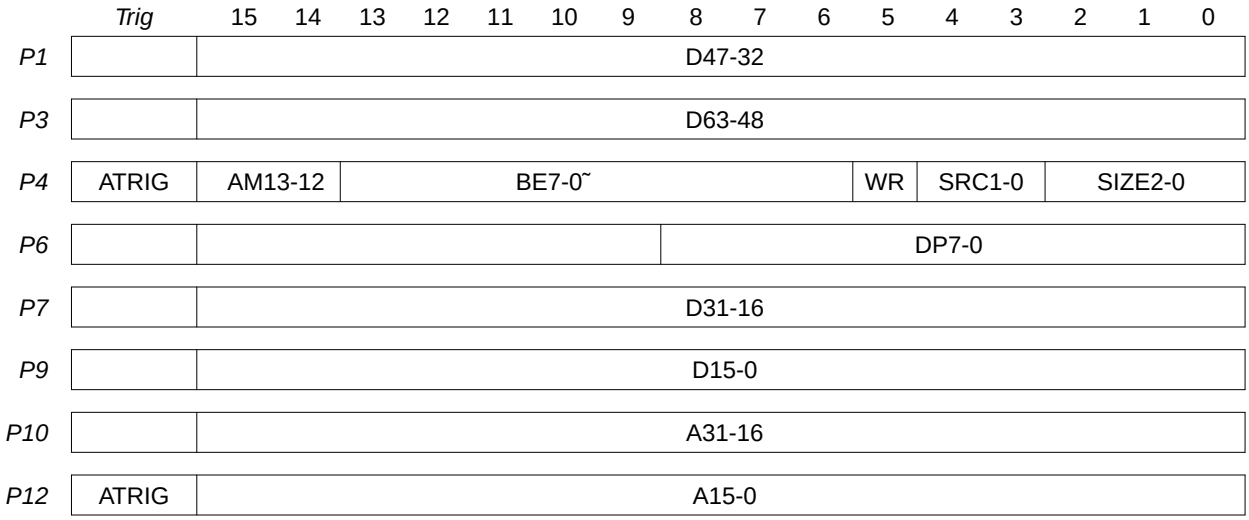


Figure B.2 Connecting an HP or compatible analyser to debug board

Those of you who haven't got an HP or compatible analyser can still use the connectors, of course; but you'll probably find Figure B.3 more helpful.

[17] **Caution:** some designs may swap memory data lines around for cleaner routing (for BONITO64, memory data connections can be arbitrarily swizzled within the low and high words). That might be fixable by a CPLD change on the debug board; so some debug boards somewhere may be different from this.

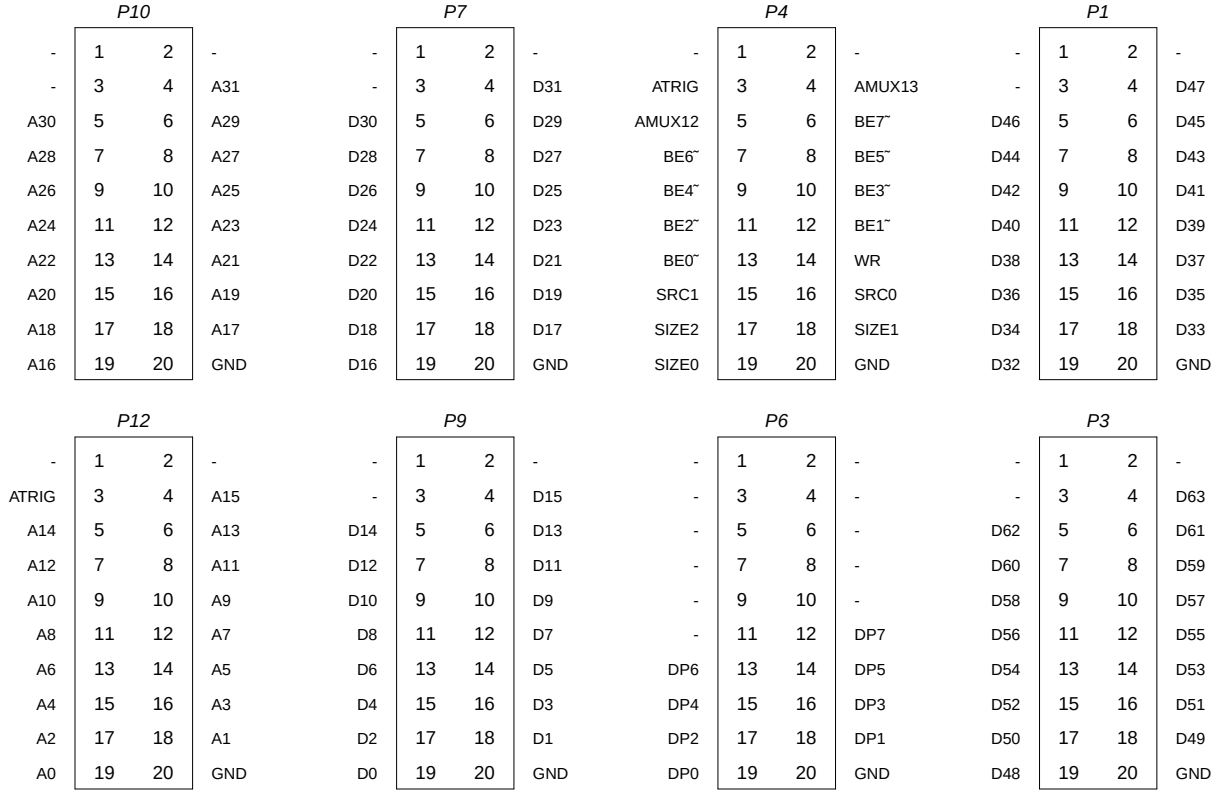


Figure B.3 Pin-by-pin analyser connection to debug board

Appendix C: - IDE interface and UDMA transfers

The UDMA hardware definitions appear in the “ATA-4” specification. Formally, this is an industry standard and full copies are available only at a price. However, free versions of drafts of the standard - adequate for many purposes - are available on the web from some of the companies who participated in the standards activity.

Under the standard, “UDMA” is a high-speed block data transfer protocol which may be negotiated for use in conjunction with the “ATA”, SCSI-like, high-level command set. But many (most?) ATA-4 compatible disk drives are also willing to use UDMA bursts for normal disk data DMA cycles, which may be much easier to program.

An introduction to UDMA

The IDE disk interface started life as really just an extension of the ISA bus, buffered so it could be shared down a ribbon cable. It’s accreted more and more options down the years as disk drive performance increased, and as the interface was opportunistically used to connect other kinds of device.

To make high speed transfer work on IDE is difficult, because the cable layout is not very good, it’s hard to control cable quality, termination is uncertain and who knows what might be attached. UDMA uses several tricks to try to improve data transfer:

- *Source synchronous transfers*: synchronous transfers go faster. But with a single clock, skew and uncertainty about clock transitions means that data being received by the clock driver is always more marginal than that being sent. UDMA always defines bursts with a clock supplied by the data provider.

  It’s not clear whether this is necessary or sensible at UDMA’s modest speeds, but it was a very popular scheme in the late 90s.
- *Clocking data on both edges*: halves the highest frequency on the cable, which reduces EMC emission problems.
- *Per-burst CRC check*: ensures that if we overreach the capacity of the cable, we get to find out about it. Note that the CRC check is always performed by the disk drive, so that it can be reported through the same register-orientated mechanism as an on-drive data error; bizarre, but might reduce the effort of porting device driver software.

In practice, if anything goes wrong on UDMA it will be down to data hold time; it seems that disk drives do not always provide the required 6ns data hold time from the strobe. You should be careful not to delay the strobe by more than the data - so, for example, we don’t recommend putting an extra buffer component between the cable *IORDY* signal and BONITO64.

UDMA cable termination guidelines

Where BONITO64’s IDE interface is used, we recommend that you use a 245-type buffer between the IDE cable and the *IOD0-15* data bus, controlled by BONITO64’s *IODIR* and *IODEN** signals.

To try to keep the cable quiet, you’re recommended to provide resistive termination on the IDE signals. Series terminators should be close to the buffer or the BONITO64 pin; pullups and pulldowns can be anywhere (within reason).

Signal	*Value (Ω)*	*Description*
IORDY	1K	Pull-up to +5V.
DMARQ *INTRQ*	5K6 10K	Pull down to ground, to ensure that the signal is “inactive” when no disk is attached.
DD7	10K	Make sure bit 7 reads 1 on register access to non-fitted disk drive. Can be used for auto-sense software.
DD0-16 *DA0-3* *CS0-1˜* other outputs	33 ^ 22	series resistors
DMARQ *INTRQ* *IORDY*	82	series resistors for inputs

Table C.1: UDMA signal termination recommendations

Appendix D: - Hardware timing and the programmer

In the past few years a gap has opened up between the performance of CPUs and their attached I/O systems. A MIPS CPU might be able to execute 50-100 instructions (from its cache) in the time that it takes to complete a single-word transfer over PCI.

In an attempt to lessen the impact of this performance difference, BONITO64 does a number of things:

- Writes are “posted”, everywhere. The CPU runs on leaving its writes queued somewhere in BONITO64 - or, if not still there, they may be queued in a PCI peripheral controller or bridge. Everything uses write posting...
- Transfers to different memory regions (local SDRAM, local I/O and ROM, PCI, internal registers) use separate queues.
- Many newer MIPS CPUs will continue to run with a read cycle outstanding. The CPU stops only when it tries to use the data obtained from the read.

At worse, external cycles can occur in a different sequence from that programmed. Even when that doesn’t happen, the interval between cycles (seen externally) may bear little relationship to the interval between the execution of the load/store instructions which they are servicing.

BONITO64 makes some promises about this cycle re-organising.

There are three different memory regions in BONITO64 for this purpose:

- Local SDRAM.
- Local I/O, ROM and BONITO64’s internal registers.
- PCI locations.

The golden rule: reads and writes to any one region are always carried out in the order in which they show up on the CPU interface.

Is that enough to make the hardware sequence invisible to programmers? No, it isn’t: here are some examples where you can get caught out, with our recommended solutions:

- *Clearing a PCI interrupt*: typically, you clear an interrupt by writing some control register on your PCI controller. It’s very often the last register access the driver makes. By the time the write hits the device and causes it to de-assert the interrupt signal, the CPU will have executed 50-100 instructions - which may well be enough for it to emerge from the interrupt routine, implicitly re-enabling the device interrupt.

  The result can be a second bogus interrupt, wasting time and confusing your device driver.

  The best solution to this is to do device interrupt signaling through mailbox registers. But not many controllers can be persuaded to do that.

  You can make sure your driver will shrug off bogus interrupts, and just let them happen. You system will spin taking interrupts until the write finally gets through and the interrupt is deasserted. This has the advantage of providing a window of opportunity for a higher-priority interrupt, if any such is active.

  You can read a device register (after the interrupt-clearing write); that will stall the CPU until the read chases the write through the PCI queues, and returns a value.

- *Local register write which affects PCI transfers*: most BONITO64 functions are configured by software-accessible registers. Cycles aimed at the registers themselves are kept in sequence, but if a write to a register affects cycles on other ports (like PCI or SDRAM) you can get trouble.

  Solution: if you’re writing a local register which has side-effects on other ports:

  1. Issue a dummy read to the affected port and wait for its value. That will ensure that no CPU transfers to the port are still queued.

2. Write your BONITO64 register.
3. Read your BONITO64 register back again and wait for the value. That way, your program won't continue until the write has really happened.

You'll need to be even more careful if you reconfigure local registers in a way which might affect DMA or PCI-master cycles, since those can happen at any time.

Appendix E: - software-visible changes from the BONITO ASIC

Most of these changes are backward compatible.

SDRAM Configuration

You can now set SDRAM options (eg the “cache latency” which the ASIC always set to “2”. It’s mostly done by changes in **`sdcfg`** (see Table 5.6 on page 30), but one of the register bits is located in the power-on-settable **`bonponcfg`** register.

The changes are:

- The field in **`sdcfg`** which used to be called “dramreset” is now **`sdcfg.drammodeset`**. The mode-set command will be issued at the next convenient SDRAM refresh time.
- Adding a flag **`sdcfg.drammodeset_done`**. so the CPU can wait for the SDRAM mode-set command to finish. Refresh time might not come around for 30μs or so.
- adding a **`bonponcfg.burstorder`** field to indicate that the CPU requires cache bursts to be provided as sequential with wrap-around, not in sub-block order. The SDRAMs are capable of supporting either.

  This flag affects the order ROM burst data is provided, too†.
- adding a **`sdcfg.dramburstlen`** field to tell the SDRAMs how long the bursts are; when working in sequential order the SDRAM has to know when to wrap-around.
- a new **`sdcfg.dramrfhmult`** field controls the refresh rate. New bigger SDRAMs need more frequent refreshes.

CPU clock rate independence

ROM/IO cycle times and DRAM refresh intervals are now scaled according to the CPU clock rate, so they don’t get too short for fast CPUs or too slow for slower ones.

To make this work you have to program **`iodevcfg.cpuclockperiod`** to the CPU clock cycle time (so 83MHz = 12, 100MHz = 10, 125MHz = 8).

If you don’t program it it will power up to zero, which is treated as a special case - I/O bus cycle times will then be very slow, and DRAM refreshes very frequent.

See Table 5.4 on 27.

Timer

There’s a new timer controlled by register ’timercfg’ whose main intended use is to accept a fixed-rate clock input and allow you to figure out how fast the CPU timer is running. The clock can be the PCI clock or a special clock fed in through *GPIO8*. See See Table 5.15 on 47.

---

† I’m not sure why this bit is in bonponcfg. That’s usually for things which must be set before the CPU can boot, but on the face of it it’s only required once the CPU is running cached, so could have been s/w set.