BONITO - PCI/SDRAM System Controller for Vr43xx
© 1998/1999 Algorithmics Ltd
Rev 2.7 of 1999/08/24

- **Direct connection to any 32-bit MIPS R4x00 CPU.**
- **Direct connection to 32-bit 33MHz PCI bus, conforming to Rev2.1. PCI arbiter and other "host" functions available, but can also operate as a peripheral. Includes PCI mailbox registers for intelligent peripheral communication.**
- **Independent CPU and PCI input clocks.**
- **High-performance SDRAM memory system using standard PC-100 parts in either a 32- or 64-bit array, including standard 100-, 144- and 168-pin DIMMs. Glueless for small systems, but by adding synchronous buffers it allows large systems which are only one clock period slower.**
- **Local ROM, I/O bus connects "dumb" components, isolated from high-speed signals. DMA support for faster devices on the local I/O bus, including "UDMA" transfers as defined in the ATA-4 standard for PC disk drives.**
- **Internal "cache" of local memory locations provides greatly enhanced PCI transfer performance for device controllers which are PCI bus initiators.**
- **PCI/local-memory copier for applications requiring bulk data transport.**
- **Configurable debug mode makes all cycles visible at a DIMM socket.**
- **Glueless support of CPU reset sequence.**
- **Includes useful generic interrupt controller.**
- **Can be configured from ROM, pins or PCI bus.**
- **Compact 352-pin 1.27mm pitch BGA package for reliable assembly.**
- **Bueno, bonito y barato!**

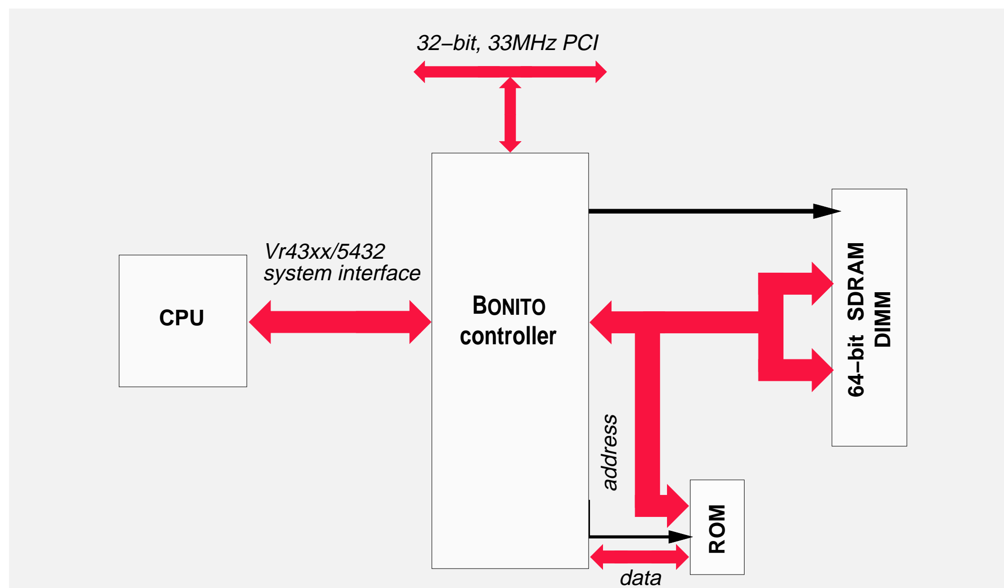


Figure 0.1 BONITO in a minimal system

1. Overview

This device has four ports: CPU, PCI, SDRAM and local ROM & I/O. Some SDRAM signals are used during some local ROM & I/O cycles, to provide more addresses. A minimal system block diagram is shown in Figure 0.1.

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 itself, with matched low-skew clock inputs.

BONITO can also be used in a larger system with external buffers to allow the connection of bigger local memory arrays and more local I/O devices, as shown in Figure 1.1.

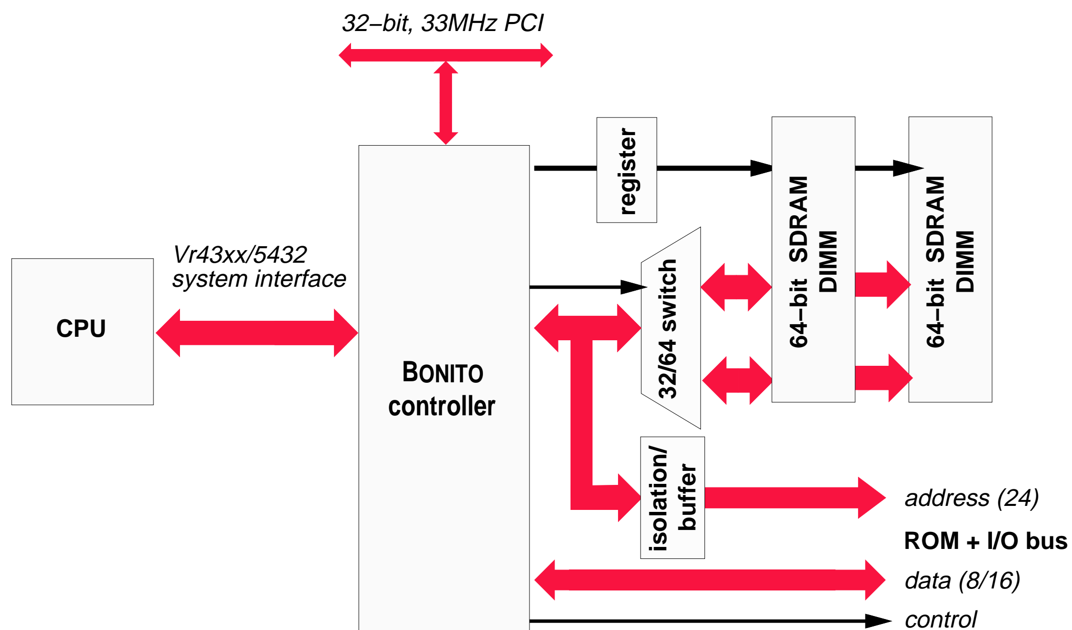


Figure 1.1 Buffered memory and expanded local I/O system

BONITO handles tasks which are common to many of those systems where a 32-bit MIPS CPU is used with a PCI bus and a local memory system. Such systems always (or almost always) need logic for some other functions, which are therefore also supported by BONITO:

- Managing the CPU reset sequence and supporting common configuration options.
- System bootstrap from local ROM or flash memory.
- High-speed transfer of data between local and PCI memory - the *PCI copier*, described in §3.2.
- A flexible and high-performance interrupt controller, see §5.14.
- Minimal DMA support for local bus devices, see §2.4.1.
- A few general-purpose I/O pins - required, for example, for access to the signature EEROM of a memory DIMM.

Within the confines of such a chip it does not make much sense to devote pins to specific I/O functions, since (unlike the above) these vary greatly from application to application.

Compared to earlier MIPS system controllers, BONITO offers a simple, fast, memory controller for PC-100 SDRAMs, with few options, and a high level of integration to minimise glue logic.

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

BONITO connects directly to any Vr43xx or Vr5432 CPU[1], and variants are flexible enough to handle any 32-bit descendant of the R4000 "SysAD" bus. It will run with CPUs to 83MHz. For CPUs which support it, parity is passed through to the local SDRAM system, and generated/checked on other cycles.

BONITO manages the CPU's reset sequence. A complete reset of BONITO from its *SysReset** pin - or the PCI *RESET** signal where BONITO is configured as a peripheral - causes the CPU to be driven through a cold-reset sequence. BONITO supports most MIPS CPU reset models, and can supply the "mode bits" for CPU configuration when required. 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 §4.3.1 for more information.

In some cases a system may be managed by a host across PCI; it's possible to configure BONITO to wait with the local CPU stalled while an external PCI host configures the chip and uploads software to local memory; see §4.3.2.

BONITO has an internal interrupt controller, and connects to two of the MIPS CPU interrupt inputs.

2.2. PCI port

BONITO 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 - it has a PCI arbiter onboard, can source the PCI reset signal, and can initiate configuration cycles.

BONITO supports CPU accesses to PCI space. CPU partial-word read or writes to PCI space are signalled 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 (cache refills and write-backs) are not implemented to PCI memory space; the different burst termination semantics of the buses make this a hazardous and troublesome feature.

Both type 1 and type 2 configuration cycles are available through a pair of registers (one sets up the PCI address bits, and the other is a data register for configuration cycles).

CPU writes to PCI are "posted"; there's a BONITO register `pciMstat` which software can read to check when all posted writes have been completed on PCI.

BONITO's PCI arbiter handles up to six external initiators 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 BONITO's own request/grant lines.

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

BONITO provides the standard configuration registers required by the PCI standard. Software running on the local MIPS CPU can modify some of these register contents, so that BONITO can take on different roles as part of an intelligent PCI controller; see §4.3.3.

[1] The choice of CPU interface type is reconfigurable at reset time between Vr4300-compatible and Vr5432-native-compatible.

BONITO can be configured to respond as a PCI target to four memory address regions. One is for the BONITO'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 influenced by some other registers. PCI access to the local I/O bus is expected to be used only for diagnostics and system initialisation; no guarantees are made about performance.

BONITO 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 sub-block order using PC-100 compliant memory devices. It runs in synchronisation with the CPU interface to minimise CPU access time. For small SDRAM systems - with a fan-out of 6 loads or less on all SDRAM control outputs - it is glueless. Larger systems require high-drive registered buffers (ALVCH374 or similar) for the multiplexed address lines and control signals which go to every DRAM device. When the registered buffers are used, a configuration register must be set to delay all data timings by one clock.

For pin-count reasons, the SDRAM data bus is used to carry addresses for I/O accesses requiring more than a small number of addresses - particularly for ROM cycles. I/O cycles using the SDRAM data bus cannot be run concurrently with an SDRAM access.

There's no support for SDRAM "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-word burst cycles (a "word" here and throughout this specification is 32 bits), but also allows word-sized reads and writes. Memory systems supporting parity generally don't directly support writing only some bytes of the SDRAM array, so BONITO implements partial-word writes with a read-merge-write sequence.

The SDRAM control signals are designed to attach to either a 32/36-bit or 64/72-bit array. Parity generation and checking are supported on 36-bit arrays, and are a diagnostics-only option on 72-bit modules.

A single 64/72-bit SDRAM DIMM may be attached by commoning up the halves of the data bus, controlling accesses using two data-mask *DQMBHi*/*DQMBLo* signals - for the high and low bank respectively. But where the number of connections to the data bus grows too large, a control signal is provided for a zero-delay FET switch bus multiplexer (Quality QS3390 or similar) which connects the chips SDRAM data bus to either the high or low half of each DIMM.

In either case each cache line burst uses a single bank, so the banks are not switched at burst speed.

PCI traffic has unconditional priority 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.

BONITO performs refresh cycles as required.

You can configure BONITO to support any SDRAM with PC-100 compatible timing, and with up to 13 row and/or column addresses. Configuration is software driven. System software can read an 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 on the SDRAM data bus. A single flash ROM device can be connected directly, but for larger systems 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.

BONITO generates Intel-style I/O device control signals, two ROM chip selects, and four I/O chip selects. ROMs must be 120ns or faster. Two fixed I/O timings are supported, corresponding to read/write pulse widths of approximately 200 and 800ns, respectively. There is no byte addressing for I/O devices; distinct I/O bus addresses are at 4-byte intervals in CPU or PCI space.

A limited I/O bus DMA facility is offered, which is particularly (but not uniquely) applicable to implementing a low-cost, high-performance IDE interface. It's described in §2.4.1.

BONITO 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 assembly of ROM data into 32-bit words on the CPU bus is done in a fixed order, regardless of programmed endianness; see §4.2 for why this is probably a good idea.

2.4.1. "DMA" for local I/O

BONITO'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 BONITO implements several up to and including "Ultra-DMA" with its 33Mbytes/s burst transfer rate.

BONITO'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

BONITO 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 GPIO pins are available to the interrupt controller too, as described in 5.14.

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.

BONITO 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 can slow down the system, so it is controlled by a configuration register 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 signalling is described in Appendix C.

3. Inside BONITO

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

- The I/O buffer cache, which is responsible for providing a temporary home for data in transit between local memory and an external PCI initiator.
- The PCI copier, used to transfer data between local SDRAM memory and PCI-accessible locations without (much) CPU intervention.

3.1. I/O buffer cache

The I/O buffer cache ("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 and write BONITO'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:

- The memory transfer unit is equal to the CPU's largest line size - 8×32-bit words.
- The cache-line-sized data stores are organised in pairs. Data from anywhere can use any of the four pairs, but consecutive data blocks are kept together in the same pair; each contiguous data stream from a PCI master device will tend to be buffered through one pair of entries.
- The cache is small, with just four pairs of entries. But in most applications 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 immediately forwarded to SDRAM), but has two particular differences from a CPU cache. The first is that the cache never reads data from memory in order to service a write, as is common for CPU caches. Instead, the cache keeps track of every byte which has been written by the I/O side; when the data is written back to memory, the IOBC performs a read-merge-write operation if any of the bytes in the line have *not* been written with I/O data.

The second difference relates to how lines are chosen for replacement. When a CPU misses in its cache and a new cache location is needed for the refill data, it's normal to make some attempt to employ the "least recently used" suitable line. By contrast the IOBC keeps its lines in pairs, and tries to use just one

pair of lines for an active data stream.

When a PCI master reads or writes data from local memory through the “IO cacheable” area, the cache automatically re-allocates the pair of the line involved in a transfer to the next sequential block of memory. If the pair line was “dirty”, the re-allocation causes it to be written back. If the PCI event was a read, the cache starts a memory operation to pre-fetch data into the newly allocated line.

I/O buffer cache coherence management

Because the IOBC is a cache, we need to consider the implications for coherence between the CPU's view of local memory and that of PCI bus initiators.

The IOBC hardware “snoops” CPU/memory traffic to ensure coherency for uncached accesses from the CPU, so device drivers which use uncached memory regions to share memory with PCI masters will continue to operate.

Snooping causes clean IOBC entries to be invalidated when the CPU writes a matching memory location; dirty entries will be written back before a matching CPU read is allowed to go ahead.

IOBC coherency with respect to cached (burst) CPU transfers can be turned on, as an option. This is a useful debug/diagnostic feature - making it easy to test for device drivers which fail to handle the cache correctly. But it is not recommended for real systems; it imposes a significant penalty on every CPU burst transfer, which is a very poor performance trade-off.

The CPU can control the state of lines in the I/O buffer cache. All I/O buffer cache lines can be written back or invalidated on CPU command. The operations might typically be invoked from within the CPU cache management functions of the board support package.

BONITO provides registers and operations which allow the CPU to inspect the state of each cache line pair, and to write-back and/or invalidate a specified line. See 5.9.1 for some programming guidelines.

3.2. PCI copier

Some applications require PCI memory-to-memory copy. These operations are not like conventional DMA, because they are not controlled by DMA request signals.

In an ideal world, it is good design practice to make sure that data is generated in the right place, not generated somewhere else and moved; but sometimes it can't be avoided. Such copies are problematic on the bus, because large memory-to-memory copies tend to absorb all available PCI bandwidth, increasing worst-case latency for all other bus users. BONITO provides a relatively dumb copy engine which can speed data between local SDRAM and another PCI-accessible memory.

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 BONITO, to gain the bus, thus reducing its impact on system-wide PCI transfer latency.

[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.

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. When a cycle has an error the copier always raises an interrupt and stops itself.

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. Programming BONITO

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 used by any BONITO 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/bonito.h`; it will save you hours of typing, reduce the risk of mistakes, and means that the worldwide family of BONITO 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

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`	32M	ROM	ROM (suitable for soldered flash) selected by *ROMCS1**.
`1e00 0000` `1f80 0000`	24M 4M		unused 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 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
		BONITO
		
		BONITO
			BONITO's own PCI configuration space registers available to other PCI bus masters
			BONITO'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.

[3] The great advantage of `bonito.h` is that we use it to build our software. But the file 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
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.

Table 4.1: CPU/local bus address map

Notes on CPU memory map and guidelines for PCI windows

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.

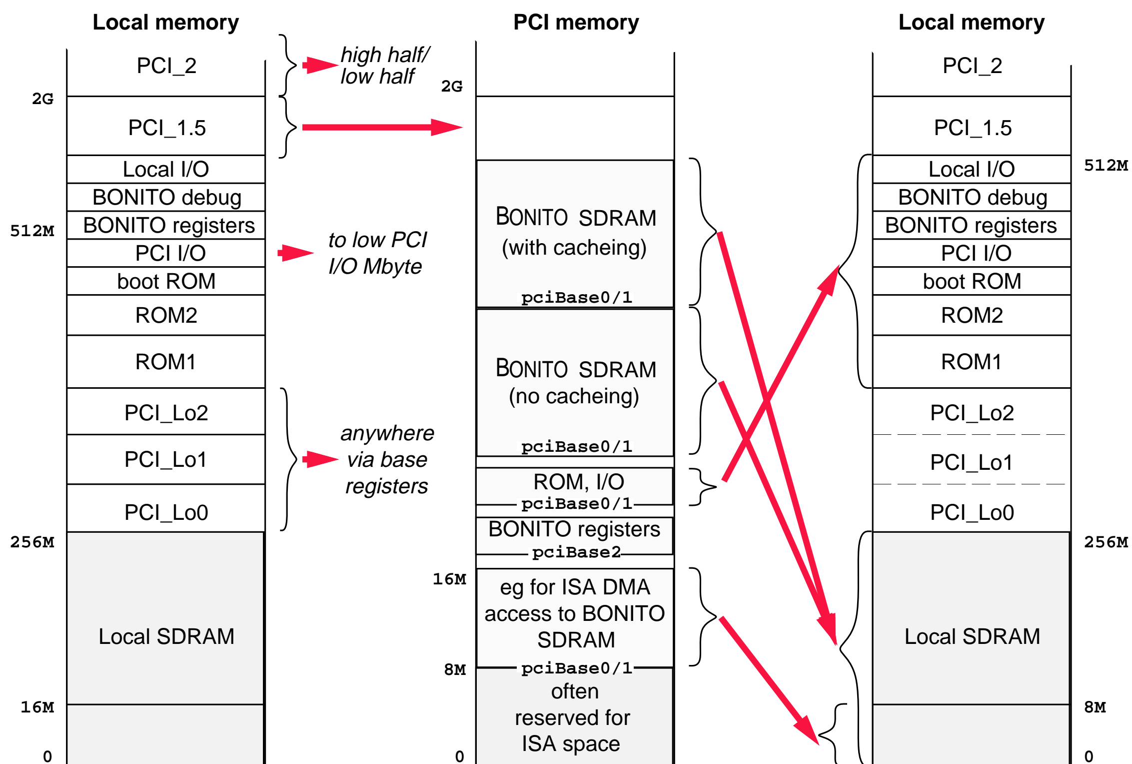


Figure 4.1 Memory regions and mappings between local and PCI space

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 BONITO 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.
- *BONITO registers*: here and subsequently, the name in bold (**`pciBase2`**) is a base register determined at configuration time.

  This region provides the PCI view of BONITO's internal registers. All registers are mapped at the same position relative to the PCI base as they are mapped in CPU space.
- *BONITO ROM, I/O (example)*: a view of the BONITO ROM and local I/O space. There's nothing to stop you trying to go through BONITO to that part of the PCI bus mapped into local memory - but it's bizarre and not much is guaranteed.
- *BONITO SDRAM (with and without IO caching) (examples)*: we show two large windows mapping all the SDRAM - once for accesses where IO caching is undesirable, and once for accesses where IO caching is a good idea. You can setup BONITO to provide a PCI window to any power-of-two sized aligned region of local memory.

If BONITO attempts a transfer which should decode as a self-access (to either BONITO registers or its attached local memory) then it will not respond on the PCI bus unless the access maps to local SDRAM (where such cycles are useful for IO buffer cache diagnostics). In all other cases the cycle will finish with a "Master Abort"[4].

4.2. Endianness

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 right, 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, and most PCI peripherals are little-endian in all their dealings over PCI[5]. A MIPS CPU may be configured to run with either endianness. Although the presence of the PCI bus is a good reason to make your CPU little-endian, sometimes an existing code base may sway the balance the other way.

So BONITO's "endianness" may be configured at reset time and changed at any time[6] by a software-writable 32-bit register. Endianness affects the interpretation of a MIPS CPU's uncached partial-word reads and writes, where it is necessary to steer read or write data to/from the correct byte lane of the system bus, so BONITO must agree with the CPU's configured endianness before any partial-word accesses can work.

---

[4] This is only to be relied on for configuration space cycles.

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

[6] In practice you'll change it (if at all) once very early in the bootstrap process.

Endianness and the SDRAM port

Data is conveyed between the CPU's 32-bit data path and the SDRAM's 32-bit data path through corresponding bit numbers. No attempt is made to do anything sensible with data in SDRAM across an endianness change; the results are undefined.

Endianness and ROM cycles

Local ROM data is always "little-endian" in that a byte whose address in the ROM device is 0 modulo 4 will always be presented to the CPU on *SysAD0-7*. This relationship is deliberately kept independent of configured endianness. We do that because some MIPS CPUs (notably NEC's Vr4300) can be set to either endianness purely by software.

The ROM's fixed bit-to-address mapping means that when you make sub-word accesses to ROM from a big-endian CPU the address at the ROM pins is actually flipped (at the byte-within-word level) relative to that coming out of the CPU.

Any read from ROM which can't be achieved with a single ROM cycle is (for simplicity) implemented by reading all four bytes and letting the CPU 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.

ROM write operations are only ever single-cycle.

Endianness and local I/O accesses

You are recommended to 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-indpendent software. Of course, only the parts of the CPU's data word which is actually connected to the device data bus are important. During I/O reads and writes the data bus *IOD0-15* feeds or is fed by the CPU data bus *SysAD0-15*.

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 I/O bus (IDE) DMA

The IDE disk bus - and any similar 16-bit data channel - has endianness; when two bytes of data are passed along the IDE cable the byte on *IOD7–0* is earlier in sequence than the byte on *IOD15–8*. To preserve this view of byte sequence for a big-endian CPU, there's a swapper built into the IDE DMA hardware; you can enable it at **`iodevCfg.wordswapbit_ide`**, described in Table 5.12.

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. In this case we recommend that all CPU reads and writes of PCI locations, and PCI master accesses to local SDRAM, should be routed through BONITO's byte-lane swapper. There are two configuration bits to set, described in Table 5.3: **`bonGenCfg.mstrbyteswap`** and **`bonGenCfg.byteswap`**.

With all PCI transfers swapped, your local CPU and PCI will share a common view of byte addressing, but means that 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 PCI accesses to internal BONITO registers; they're defined as 32-bit aligned objects and are generally OK. Local CPU software should avoid doing byte or other partial word transfers with BONITO registers.

Endianness and bootstrapping

When the system starts up BONITO is operating with undefined endianness. Since this may not match the CPU's ideas, you can not (at this stage) rely on BONITO correctly interpreting any partial-word read or write operation from the CPU. You must program BONITO's CPU configuration register really, really early!

4.3. Bootstrapping

From a cold start, some MIPS CPUs require a fairly complex reset sequence; BONITO takes care of it. Most MIPS CPUs require some reset-time configuration. Where this is static, your design needs to include the appropriate pull-ups/downs; but some MIPS CPUs rely on clocking in a bit-stream from a configuration ROM. The MIPS interface - the CPU drives the clock and looks for serial data - is incompatible with low-cost serial ROMs. But here again BONITO will handle it.

In addition BONITO can perform a software-initiated reboot of the MIPS CPU, as if from a cold reset. This can be valuable in that it permits a CPU to come up in some "default" configuration, set-up BONITO's internal registers with an alternate CPU configuration and then reset itself, ending up in a software-determined configuration. See §4.3.1 below.

In most BONITO 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. However, it's also possible for a host attached over the PCI bus to hold the local MIPS CPU while the remote host uploads software into local SDRAM memory; described in §4.3.2.

4.3.1. CPU-specific reset options and "mode bits"

All the MIPS CPUs compatible with BONITO are to some extent hardware-configured 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 BONITO 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 BONITO 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.

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. A CPU self-reset does not reset BONITO and in this case the mode bit stream is fed from the register **`intPol`**, which is borrowed for this 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 *intPol* and reconfigure the CPU accordingly.

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

4.3.2. Choice of boot memories and PCI-initiated bootstrap

BONITO 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 *ROMCS0** window is much smaller (4Mbytes maximum) and is intended for a first-time-only bootstrap for designs which have flash memory soldered using *ROMCS1**, and need some way to get that flash device programmed in production.

However, it's possible to build a BONITO 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. BONITO 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 BONITO comes up.

2. The PCI-located host can now program BONITO from the PCI side - remember, all BONITO registers can be reached from PCI. The PCI host must initialise much of BONITO - 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.

4.3.3. Software-determined PCI configuration characters

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.

BONITO can be used in two roles. If the MIPS CPU is the PCI bus host, then BONITO'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 BONITO 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 BONITO configuration registers to see a disk controller, not a "MIPS CPU bridge".

Three facilities in BONITO are available to help with this:

1. BONITO'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 MIPS 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.

   BONITO'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.

---

[8] Some BONITO-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`

So the apparent size of the regions mapped by BONITO's base registers can be changed by the MIPS CPU.

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 BONITO's configuration-space registers. So there's also an BONITO option - available as a reset-time configuration bit **`bonPonCfg.configdis`**- which causes BONITO to defer host processing, by responding with a PCI "retry" to any configuration-space access. Local software should hurry to fix up BONITO'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.

BONITO'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 BONITO internal registers are accessible to both the CPU and an external PCI master. 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

Register	*Page/figure*	*What is it*
`bonGenCfg`	19/Table 5.3	Early boot-time configuration, mostly PCI-related
`bonPonCfg`	18/Table 5.2	Configuration bits which can be set either way using *IOD* pullups
`copCtrl`	30/Figure 5.5	PCI copier registers
`copDAddr`	30	
`copGo`	30/Figure 5.5	
`copPAddr`	30	
`copStat`	30/Figure 5.5	
`gpioData`	36/Figure 5.8	GPIO level read/write
`gpioIE`	36/Figure 5.8	GPIO input/output setting
`intEdge`	36/Table 5.13	Interrupts selected as level/edge triggered
`intEn`	37/5.14	Separate interrupt enable bits
`intEnClr`	37/5.14	Interrupt enables - per-bit clear
`intEnSet`	37/5.14	Interrupt enables - per-bit set
`intISR`	37/5.14	Readable interrupt inputs
`intPol`	37/5.14	Interrupt polarity
`intSteer`	37/5.14	Which of two CPU interrupt pins gets raised by each interrupt condition
`iodevCfg`	34/Table 5.12	I/O bus cycle characteristics
`ldmaAddr`	35	I/O bus DMA, mostly for IDE
`ldmaCtrl`	35/Figure 5.7	
`ldmaGo`	35/Figure 5.7	
`ldmaStat`	35/Figure 5.7	
`pciBase0`	24/Table 5.5	Base registers in PCI configuration space, which define what regions BONITO makes available to other PCI initiators.
`pciBase1`	24/Table 5.5	
`pciBase2`	24/Table 5.5	
`pciCacheCtrl`	32/Table 5.10	Registers for the I/O buffer cache (also known as "PCI cache").
`pciCacheTag`	32/Table 5.11	

Register	Page/figure	What is it
`pciClass`	24/Table 5.5	Standard PCI configuration registers
`pciCmd`	26/Table 5.7	
`pciDid`	24/Table 5.5	
`pciExpRBase`	24/Table 5.5	
`pciInt`	24/Table 5.5	
`pciLTimer`	24/Table 5.5	
`pciMail0-3`	36/5.12	Mailbox registers
`pcimap`	28/Table 5.9	Register to fix the windows available to the local CPU to access PCI memory or devices.
`pcimap_cfg`	29/Figure 5.4	Used to complete the PCI address when the local CPU is using BONITO to perform PCI configuration cycles.
`pcimembaseCfg`	25/Figure 5.1	Used by local host to size and position the PCI-accessible windows into BONITO's local memory and local I/O.
`pciMStat`	29/Table 5.10	How many posted writes are still pending?
`sdCfg`	21/Table 5.4	Set up BONITO to match the SDRAM shape, size, speed etc

Table 5.1: All registers

5.2. General principles for BONITO registers (read this)

Don't type in 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/bonito.h`.

To avoid lengthening the whole manual with endless repetition, the following general rules apply to the use of BONITO'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 BONITO register.
- Some options affect BONITO's support for the CPU bootstrapping itself, so must be settable at reset time. Register fields representing these options take their values by sampling the signals *IOD0-15* (the IO data bus) while the system reset *SYSRESET** is still active. BONITO 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 §4.3.1 for where those bits come from.
- Many other register bit-fields are forced to a fixed level (most often zero, occasionally 1) following reset. However, this manual will document that only when it's 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.

5.3. Configuration register

The **bonPonCfg** register brings together BONITO 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, **bonPonCfg** is really 18 bits long, and the two highest bits show the reset-time value of *ROMCS0-1**; like other bits in the register which have no hardware effect, they could be used for software configuration information. Here are the fields which have a hardware effect:

bonPonCfg register

Bit(s)	*Name*	`Value` / *Effect*
14	**CPUbigend**	1 to support big-endian MIPS CPU; see §4.2 for a full description of the consequences. A wrongly-configured BONITO is still capable of being used by a MIPS CPU which avoids all sub-word accesses - so it's possible for your system to leave this to the default setting so long as early software changes it to match the CPU.
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 BONITO *always* passes SDRAM parity straight through, and generates parity for I/O and PCI data.
11	**romCs1fast**	Select "faster" operation on the ROM attached to these select signals; slow ROM has an access time of 12 CPU clocks, while fast ROM cycles in 9 CPU clock periods. 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**	read-only - where CPU boots from; picks which memory region will be selected for accesses in MIPS' traditional `0x1FC0.0000` start address. `00` from 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` from local SDRAM; but also 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, BONITO 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.
4	**is_arbiter**	when set 1, BONITO operates as PCI arbiter. When 0, the roles of the zero-th request and grant signals are reversed, and BONITO will use the services of an external arbiter.

bonPonCfg register

Bit(s)	Name	Value / Effect
3	**pcireset**	If BONITO 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 BONITO as PCI bus controller. But that's now done with the dedicated signal *SysController**.
2-0	**CPUtype**	What kind of CPU is attached? `000` NEC Vr4300 `001` NEC Vr5432 `100` Most other compatible CPUs with wide command bus, R4000-heritage reset, and "mode bits" configuration. See §4.3.1 for where the configuration mode bits come from.

Table 5.2: Fields in **bonPonCfg**

5.4. Mostly-PCI configuration - not affected by pullups

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
16	**noretrytimeout**	0/1	Normally (with this field 0), BONITO will not retry indefinitely on a locally-initiated PCI cycle where the target returns a "retry" response; eventually BONITO gives up and returns a PCI error. Sometimes the target may just be very slow, and software can catch the error condition and re-try the cycle manually. 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 BONITO 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. Only in even more bizarre circumstances should it ever be set differently from the **byteswap** bit described below, which controls transfers where BONITO 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 11-10	**ciqueue** **cachealg**		More IOBC diagnostic fields
9	**wbehinden**	 0 1	IOBC write-behind control Data written by a PCI initiator to BONITO's local memory is immediately forwarded to the memory controller. Write data is retained in the IOBC until the line is recycled; much more efficient for normal purposes.

`bonGenCfg` register

Bit(s)	Name	Value	Effect
8	`prefetchen`	0/1	IOBC read prefetch control. Set 1 to allow BONITO 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 BONITO 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	Should BONITO's interrupt controller be connected to the PCI interrupt signal *INTA**? No; *INTA** (if an output at all), will be explicitly controlled by the MIPS CPU. Yes; *INTA** (if set as output), will track the level from the interrupt controller (see §5.14) which is driven on the CPU's second interrupt signal, *Int1**.
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 §5.14.
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 BONITO 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`

5.5. SDRAM configuration

BONITO 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*[9]. Sides can be 64- or 32-bits wide; where we

[9] 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 the chunks "sides" if they share a chip select. Sorry; we have to call them something.

want to talk about one of the 32-bit halves of a 64-bit side we'll call it a *half-side*.

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.

BONITO can directly support two modules - 32- or 64-bits wide - and each may have two sides. BONITO needs to be set up so that the memory map extends from zero to 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?

And for each module:

- Is it 64- or 32-bits wide?
- 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 BONITO decode between 11 and 14.
- How many *MuxAD* addresses does the SDRAM decode in its second (*Cas**) phase? BONITO 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.

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 compact 2-wire interface. Software can drive BONITO's GPIO pins to access the data.

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 BONITO'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, BONITO's registers assume that the two sides within each module are always identically organised.

The first module's sides are selected by the signals *DCS0L**/*DCS0H** and the second module's sides by *DCS1L**/*DCS1H**.

Modules can be either 32/36- or 64/72-bits wide, but BONITO has only a 36-bit SDRAM data bus so wide modules are wired with the high and low data halves commoned together; transfers to wide modules are qualified by the assertion of only one of the two mask signals *DQMBLo*/*DQMBHi*.

BONITO always implements parity on the memory array - it passes parity through on CPU cycles, and generates/checks it on all other cycles. However, unless a 72-bit memory is set up so that half of the parity bits are always enabled with the appropriate half of the data bus (and 72-bit DIMMs don't do this) parity won't work correctly. With wide modules you can still use parity for diagnostic and test purposes - parity should be correctly stored and returned for either the lower or upper half of the normal range of this memory module, according to whether *DQMBLo* or *DQMBHi* enables the parity bits on your module.

You tell BONITO about the characteristics of each module through the register `sdCfg`, shown in Table 5.4.

---

† Some SDRAM manufacturers count an internal bank-select address - the same as the BONITO 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.

`sdCfg` register

Bit(s)	Name	Value	Effect
23 22	**`dramparity`** **`dramextregs`**	`0/1` `0/1`	Set 1 to enable parity generation/checking in the DRAM system. Set 1 if your system uses high-drive registers to boost multiplexed addresses and shared control signals to the SDRAM modules.
21	**`dramreset`**	`0`	TBD - leave 0 for now
20-19	**`extraswidth`**		Timing option‡
18	**`extprech`**		Timing option‡
17	**`extrascas`**		Timing option‡
16	**`extrddata`**		Timing option‡
			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 if no memory is fitted in the B module Not implemented yet - ignored!

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` `00` `00` `00`	no of row addresses at SDRAM† 11 12 13 14

Table 5.4: Fields in `sdCfg`

The whole register is cleared to zero on power-up.

Note that from system reset **`sdCfg`** is cleared. This is intended to leave the SDRAM configuration in a default state which will yield 4Mbytes of functioning memory, so long as any usable devices are connected to the chip-select *DCS0L**. This can be used for some bootstrap functions - but the contents of memory in this default configuration may be scattered all over the address map by the time you've installed the

‡ SDRAM timing is obscure. Early users should not change the values set up by Algorithmics' power-up code.

correct configuration.

5.6. BONITO registers available in PCI configuration space

These registers, shown in Table 5.5 conform to the PCI 2.1 standard[10]. BONITO 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 BONITO's internal register block. Some registers which are read-only from the PCI side are writable from the local side.

By reset-time option, BONITO 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.

<table>
<tr><th></th><th colspan="2">31 … 16</th><th colspan="2">15 … 0</th><th></th></tr>
<tr><td>pciDiD</td><td colspan="2">Device ID</td><td colspan="2">Vendor ID</td><td>00h</td></tr>
<tr><td>pciCmd</td><td colspan="2">Status</td><td colspan="2">Command</td><td>04h</td></tr>
<tr><td>pciClass</td><td colspan="3">Class Code</td><td>Revision ID</td><td>08h</td></tr>
<tr><td>pciLTimer</td><td>0</td><td>Header Type</td><td>Latency Timer</td><td>0</td><td>0Ch</td></tr>
<tr><td></td><td colspan="4">Base address registers</td><td></td></tr>
<tr><td>pciBase0</td><td colspan="4">BONITO register bank, 64Kbyte</td><td>10h</td></tr>
<tr><td>pciBase1</td><td colspan="4">Local SDRAM memory (not IO-cached), 256Mbyte</td><td>14h</td></tr>
<tr><td>pciBase2</td><td colspan="4">Local SDRAM memory (IO-cached), 256Mbyte</td><td>18h</td></tr>
<tr><td></td><td colspan="4">unused</td><td>1Ch</td></tr>
<tr><td></td><td colspan="4">×</td><td>28h</td></tr>
<tr><td></td><td colspan="2">0</td><td colspan="2">0</td><td>2Ch</td></tr>
<tr><td>pciExpRBase</td><td colspan="4">"Expansion ROM" Base Address
(window into 64Kbytes of boot ROM)</td><td>30h</td></tr>
<tr><td></td><td colspan="4">Reserved</td><td>34h
38h</td></tr>
<tr><td>pciInt</td><td>Max_Lat</td><td>Min_Gnt</td><td>Interrupt Pin = 1</td><td>Interrupt Line</td><td>3Ch</td></tr>
</table>

Table 5.5: Standard PCI configuration space registers

5.6.1. BONITO device and vendor ID

Not yet settled. FPGA controllers have an unauthorised vendor ID of `0xDF53`, while we figure out something more sensible.

[10] Note that the FPGA version of BONITO 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, BONITO will not respond.

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 5.6: 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. BONITO base address registers

BONITO can map regions as large as all of its local memory - 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 BONITO’s memory regions each only offer a 16Mbyte window; the window can be repositioned within BONITO’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; the **`bonPonCfg.config_dis`** big (see Table 5.2 on page 18) may help here.

The **`pciBase2`** region provides access to all the programmable registers inside BONITO - 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. **`pcimembaseCfg`**

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 BONITO 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 §5.9.1 for more about this subject.

`trans/mask`: two six-bit fields which determine bits 27-22 of the local address used for accesses in this region. The PCI address is first masked to remove bits 31-28, and then PCI address bits 27-22 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 5.8 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		 0 1	Reporting of address-time parity errors - writable Do nothing Drive *SERR** signal when detected.
7		0	Address/data stepping - always 0, never done
6		 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, BONITO only does plain writes.
3		0	Special cycles - always 0, BONITO doesn't issue special cycles
2	**mstren**	 0 1	Master enable - writable Don't initiate PCI bus cycles Initiator role enabled. BONITO almost certainly needs this bit set.
1	**memen**	 0 1	Memory space enable - writable Disable target functions. Host should not leave this zero. Allow BONITO to respond on PCI
0		0	PCI I/O space enable - BONITO never responds to I/O space cycles.

Table 5.7: Fields in **pciCmd** *- PCI configuration space "Command" register*

5.6.5. Status Register

This is also a PCI-blessed standard register. It can be read as the high bits of the `pConfsc` register.

`pciCmd` register

Bit(s)	*Name*	*Value*	*Effect*
31		`1`	Detected a parity error - as initiator (reading) or as target receiving write data. Unaffected by the enable bit in `pConfSC(Command).`
30		`1`	BONITO is driving *SERR**, having noticed an address parity error
29		`1`	"Master-Abort" signalled BONITO initiated a read or write but no target responded with *DEVSEL**(*Master-Abort* in the PCI specification)
28		`1`	Target abort received BONITO 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		`0`	Target abort signalled. Always 0 - BONITO never responds with a target abort.
26-25		`01`	*DEVSEL** speed. Always this value - BONITO always responds as target on the second clock.
24		`1`	Initiator-noted parity error. Set when there's a parity error (whether noticed by us or signalled 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 5.8: PCI configuration space "Status" register

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 9. The PCI interface does not support burst cycles for CPU cache refill and write-back, so the CPU cannot access PCI locations through cached space - the results are undefined.

Caution: Vr4300 and Vr5432 CPUs use burst cycles for uncached accesses when executing a load/store instruction for an integer double (`long long`) or floating point double (`double`). Such load/stores to PCI-mapped locations will have undefined effects; so don't regard PCI locations as simple program memory.

5.7.1. PCI address regions

The CPU's windows onto PCI space are mapped using a "window" register called **pcimap**, shown in Table 5.9.

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 5.9: 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

Normal PCI reads are synchronous; the CPU interface is held in wait states until the data is 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.

5.7.3. PCI writes

PCI writes are always posted; the address and data for the transaction are stored inside BONITO 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, see page 29.

5.7.4. PCI error conditions on CPU-initiated cycles

If the PCI cycle finishes with any kind of error:

- 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 5.8 above.

- If the CPU is stalled waiting for a read it will receive a MIPS “bus error”.

  If the failing cycle was a write, 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 §5.14.

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 BONITO can be deadlocked (more or less) if for some reason a defective target responds with retry forever. BONITO 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. BONITO’s write-retry limit is set large enough that it’s occurrence should be seen as fatal.

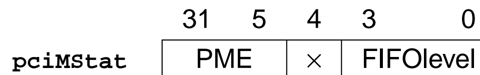


Figure 5.2 Fields in **`pciMStat`** *- PCI master status register*

The important field here is **`pciMStat.FIFOlevel`** which goes to zero when BONITO has no posted writes left queued. Software will wnt 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.

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 is needed:

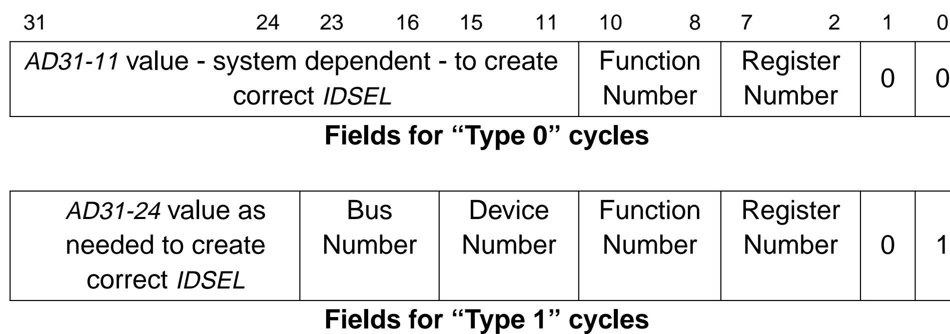


Figure 5.3 PCI address-time values for configuration cycles

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

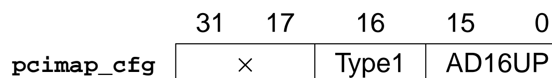


Figure 5.4 Fields in **`pcimap_cfg`**

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 (and reads may be posted) exactly like any other PCI accesses. BONITO'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. Copier transfers generally have lower priority than any other transfers, so the (potentially very long) stream defers to CPU or PCI master transfers.

The CPU can issue a second copier command at once, so that as soon as one copy is finished BONITO immediately begins the second transfer; software can use this to "chain" copier transfers to keep data flowing as fast as the PCI environment will permit. Two internal signals *copyrdy* and *copyempty* are fed through to the interrupt controller 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.

A third internal signal *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.

You submit a copier command by writing the **copDAddr**/**copPAddr** registers, which define the local DRAM starting address and the PCI starting address respectively. Write the direction and the number of blocks to copy into **copGo**. Finally write **copCtrl** to start the transfer.

The copier only deals in whole 32-byte blocks from local DRAM, aligned to a 32-byte memory boundary. The PCI transfer can start at any word-aligned address; note that the PCI address in the copier is not mapped like CPU→PCI cycles, but is a PCI physical address.

The **copCtrl** register is shown in Figure 5.5.

	31	30	29 … 17	16	15 … 0
copCtrl	start	reset	×		
copStat	stopped	reset	×		
copGo	×			write?	size (blocks)

Figure 5.5 Fields in **copCtrl**, **copStat**, **copGo**"

The register is usually called **copStat** when reading, to emphasise the fact that the fields don't just read back:

- **`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. Write this bit zero before trying to submit a copier entry.
- **`write`**: direction - "1" to transfer from DRAM to PCI.
- **`copCtrl.size`**: the number of 32-byte blocks to transfer.
- **`copStat.size`**: 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 (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.

5.9. PCI access to SDRAM

PCI initiators can access local SDRAM through either of BONITO'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.

PCI initiators access SDRAM through the *I/O Buffer Cache* - *IOBC* for short. You'll also sometimes see it called the "PCI cache". The IOBC keeps copies of local memory data in 8-word chunk, 8-word-aligned in local memory space; the chunks are the same size and alignment as CPU cache line blocks. All traffic between the local memory and the IOBC are 8-word bursts at full memory speed[11]. Each of the IOBC's four cache lines holds *two* 8-word chunks of data. One is "current" and the other may hold prefetched data (if the last PCI request in this memory region was a read) or data waiting to be written back to memory (if the last PCI request in this region was a write).

As described in §5.6.3, a window can be marked as either "I/O cacheable" or "I/O uncacheable". "Cacheable" accesses will trigger read-aheads and gather multiple PCI writes into a single memory write burst, which will considerably improve performance for accesses which are predominantly sequential. A window marked as "uncacheable" might be suitable for devices which are only accessing an odd word, but is probably suitable only for diagnostics.

5.9.1. IOBC management

The operation of the IOBC is not software-transparent. Software intervention is required:

1. To ensure the IOBC is "clean" when the system starts up (invalidate all line);
2. To ensure that "stale" memory data is not retained in the IOBC after it has been overwritten by the CPU; any line matching memory locations which may be read by a PCI bus initiator should be invalidated before the initiator starts;
3. To ensure that the tail end of data written by a PCI initiator does not lurk invisibly inside BONITO, but is pushed out into memory; any line matching locations which may have been written by a PCI bus initiator should be written back before the CPU reads that data.

While this discipline is similar to that already required for the MIPS CPU's own caches, it's not the same; there are times when the IOBC needs attention when the MIPS caches are safe. See §5.9.2 below for

[11] Actually, it sometimes happens that IOBC lines must be written back to local memory even though not all the words have been written from PCI; this then produces a somewhat slower read-merge-write sequence.

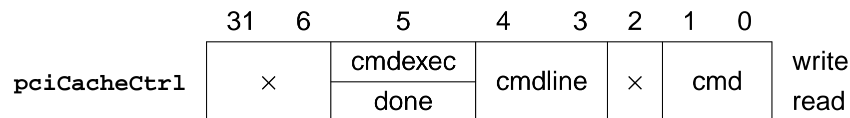
some suggestions as to how and when to program devices for correct IOBC operation.

The IOBC needs to be initialised by software before it can be used. Transfers happen in the basic 8-word storage unit, which is BONITO's preferred transfer unit in and out of SDRAM. Each of the IOBC's four cache lines holds *two* 8-word chunks of data.

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.

Management is done with the register **`pciCacheCtrl`**, which is written to make something happen; and the register **`pciCacheTag`**, used to read or write tag fields.

The control fields are shown in Table 5.10. The register fields are somewhat different when reading or writing it.



	31 6	5	4 3	2	1 0	
`pciCacheCtrl`	×	cmdexec	cmdline	×	cmd	write
		done				read

Table 5.10: PCI Cache control register

The fields are as follows:

- **`cmd`**: 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.

For diagnostic purposes you might want to root around in the IOBC to see what data has got stashed there. You must first issue a "read tag" command for a particular line, and then read the (read-only) **`pciCacheTag`** register, whose fields are shown in Table 5.11.

`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`**	line modified, either buffer
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 5.11: PCI Cache Tag register

You probably won't understand all of the fields in `pciCacheTag`; we'll only explain them in this manual as they turn out to be necessary.

5.9.2. Porting drivers to use the IOBC

Drivers for any PCI device which reads or writes the local memory may need some adjustment to work correctly.

5.10. Local I/O configuration

BONITO 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 with word-orientated addresses (the I/O bus does not see CPU address bits 0-1 - ROM cycles excepted - thus evading any endianness questions). Accesses to it should always be word-wide.

The bus uses a simple Intel-style signalling system, where chip selects qualify addresses but devices respond only when they see either of the separate read and write strobes. The timing diagram Figure 5.6 shows the basic attributes of the cycle.

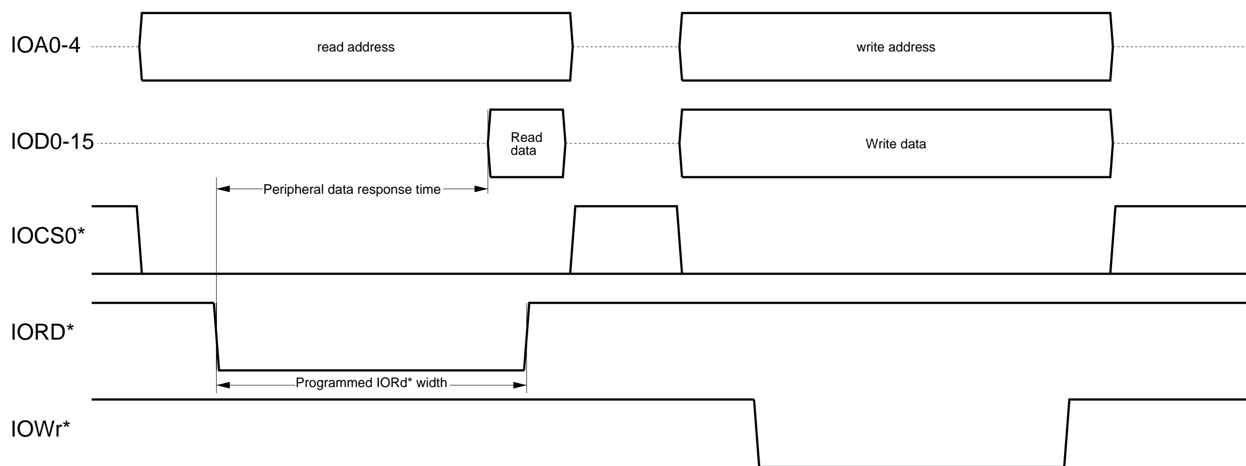


Figure 5.6 Local I/O bus read and write cycles

You can see that the address and chip select signal - *IOCS0** in this case - have a comfortable setup, nominally two CPU clock periods, to the read or write strobe *IORD**/*IOWR**. 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 BONITO at the rising edge of the *IORD** signal. The chip select and address lines are held for about one CPU clock time after the rising edge.

Write data is driven by BONITO 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 800ns, corresponding to "fast" and "slow" devices.

In addition, the read/write pulse can be extended by using the *IORDY* signal. It should be taken 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 BONITO 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 5.12:

iodevCfg register

Bit(s)	*Name*	*Value*	*Effect*
24-21	**dmaoff_ide**	0-2	Another UDMA transfer rate; set same as **dmaon_ide**.
20-16	**dmaon_ide**	0-2 0 1 2	UDMA "mode" - transfer rate from BONITO→disk, defined by the half-clock time (since UDMA uses both edges of the clock): 120ns (mode 0, 16Mbytes/s) 80ns (mode 1, 25Mbytes/s) 60ns (mode 2, 33Mbytes/s)
15	**modebit_ide**	0/1	Set 1 to enable "UDMA" transfers to an IDE disk. Set 0 to use conventional DMA. The disk may be programmed for either word-at-a-time or 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.
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; set 0 to rely on the addresses on *IOA4-0*.
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 '245-type bidirectional buffer controlled by *IODIR* and *IODEN**; 0 otherwise.
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 800ns).

Table 5.12: Fields in **iodevCfg**

ROM cycles are different. The *IORD** pulse width for ROM is nominally 120ns, since flash memories are faster than most peripherals. Secondly, the address bits *IOA0-4* are not only valid for 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.

5.11. Local I/O DMA control

BONITO provides a DMA facility on its local I/O bus, strongly - if not quite solely - orientated to the needs of an attached "IDE" bus. BONITO 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 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 - see **`ldmaCtrl`**;
- Write the starting address in local SDRAM into **`ldmaAddr`**;
- Write the transfer count and direction into **`ldmaGo`**.
- Write **`ldmaCtrl`** to set the **`start`** bit. But nothing happens yet unless the device has already asserted *DMARQ*.
- Program your mysterious 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. 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 registers **`ldmaCtrl`** and its alias (for reading) **`ldmaStat`** are shown in Figure 5.7.

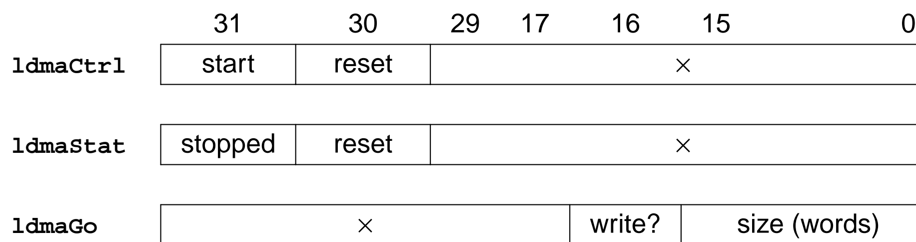


Figure 5.7 ldmaStat and ldmaCtrl register layouts

The fields in both registers as shown in Figure 5.7 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.stopped`**: 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.

5.12. PCI Mailbox registers

Four mailbox registers **`pciMail0-3`** may be read and written; only the lowest 3 bits are implemented, 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 §5.14 for details.

5.13. 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.

	31 25	24 16	15 10	9 0	
	gpinr	gpior		gpiow	
`gpioData`	*GPIN5-0* pins ×	*GPIO8-0* pins ×	×	*GPIO8-0* readbacks GPIO levels	read write
`gpioIE`	1111111 (inputs)	×		0 = output 1 = input	

Figure 5.8 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.

5.14. Interrupt control

BONITO contains a simple, flexible interrupt controller. In addition to a number of input interrupts signalled 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?*
30-25	**`gpins`**	The general-purpose input pins *GPIN5-0*.
24-20		Not connected
29-16	**`gpios`**	The general-purpose I/O pins *GPIO3-0*.
15-14		Not connected
13	**`pciMTimeout`**	A PCI cycle initiated by BONITO has been abandoned after too many retries. See §5.7.4.
12	**`dramperr`**	Parity error detected by SDRAM controller, when enabled.
11	**`systemerr`**	PCI error in cycle when BONITO is target.
10	**`mastererr`**	PCI error in cycle when BONITO is initiator.
9	**`pciirq`**	Active level on PCI interrupt pin *IRQA**.

`intXXX` register

Bit(s)	Name	What are they?
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 §5.11) 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`**	Interrupts caused by non-zero data in one of the four PCI mailbox registers

Table 5.13: Fields in **`intXXX`** *registers*

The interrupts are configured for polarity and edge/level sensing as follows.

- **`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 **`intEnClr`** also resets the edge-detecting latch of any external interrupt.

- **`intPol`** can be used to invert selected external interrupts. The inversion happens before the edge-detecting latch.

- **`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. Note that the interrupt output can also appear on the PCI line *INTA** by setting the **`bonGenCfg(irqa_from_int1)`** register bit, shown in Table 5.3 above.

Some BONITO 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 0. Similarly, **`intEdge`** bits corresponding to interrupt sources which must always be latched may always read 1.

6. Hardware description

6.1. Signals

The controller has about 235 active pins, and is accommodated in a 352-pin BGA package[12]. The signals on this chip are as follows:

Signal Name	*Type*	*Description*
Clock buffer signals		
Clockin	In	Raw clock input when BONITO is acting as clock buffer. Not connected to anything else, so an external clock buffer could be used instead.
ClockOut0-5	Out	Six identical buffered versions of *Clockin*; they should inherently have low skew between themselves, but have no defined relationship to *Clockin*. They should each have only one load, and should be arranged to be the same length of track. One of them should be connected (only) back into BONITO's own *MasterClock* input, one to the CPU's clock input; the others are available for SDRAM banks.
CPU interface signals		
SysAD0-31	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*.
SysADC0-3		
SysCmd0-8		
EValid/ ValidIn**	Out	Pulsed when BONITO is driving the CPU bus.
PValid/ ValidOut**	In	Pulsed when CPU is driving the CPU bus
*EOK**	Out	CPU cycle flow control. For CPU with separate *RdRdy** and *WrRdy** pins, *RdRdy** should be held permanently active and this signal connected to *WrRdy**.
PMaster/ Release**	In	Shows when CPU stops driving the bus in a read cycle
CPUClock	In	Identical to CPU's *MasterClock*.
*SysReset**	In	Reset for BONITO and perhaps other circuits. Often connected to the active-high power-good signal from a power supply.
*SysController**	In	Dedicated configuration signal. If it's low, BONITO drives the PCI *Reset** signal; if it's high, PCI *Reset** becomes an input and itself resets all functions in BONITO. Unlike all other BONITO configuration signals, it must be stable at all times.
*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	BONITO's interrupt lines to CPU
PCI interface signals		
CLK	In	PCI bus clock, 33MHz nominal

[12] FPGA prototypes are in a larger package with a completely different pin-out.

Signal Name	*Type*	*Description*
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**		
*LOCK**	In	PCI exclusive-access control. Not provided as an output - MIPS CPUs have no notion of an atomic read-modify-write sequence. Nor is anything done with it as an input.
*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 BONITO 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	May be driven by CPU command. Useful to peripherals wishing to generate an attention interrupt to a remote CPU.
*REQ0-5**	In	When BONITO 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 BONITO's PCI *request*, and *REQ0** acts as its *grant* (swapping roles means we can leave some signals as pure inputs).
*GNT0-5**	Out	
SDRAM interface signals		
MUX0-13	Out	Multiplexed addresses. If you configure BONITO to use an external FCT374 high-drive register (or similar component) to drive more SDRAM loads you should pass all the shared signals through the register - that's *MUX0-13*, *DBA0-1*, *DRAS**, *DCAS**, *DWE**, *DCKE*, *DCS0-1H**and *DCS0-1L*.
DBA0-1	Out	Bank select - an additional address line into SDRAM components
DD0-31	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.)
DDP0-3		Parity/check bits for data bus
*DRAS**	Out	SDRAM cycle control signals: RAS, CAS, write-enable and clock enable. Typically common to all DRAMs.
*DCAS**		
*DWE**		
DCKE		
*DCS0-1H**	Out	DRAM chip selects, typically driven in pairs
*DCS0-1L**		

Signal Name	Type	Description
DQMBLo *DQMBHi*	Out	Byte “masks” - that is, byte enables. Used to select one or other 32-bit half of the SDRAM. Partial-word writes to memory are actually implemented inside BONITO using a read-modify-write cycle. Each signal is attached to four of the byte masks going into the DIMM.
*DDMuxHi** *DDMuxLo**	Out	Output signal to enable DRAM data to come from the high/low half of a 64-bit DIMM’s data bus respectively. Wire to the select pins of a QS3390 or compatible component.
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.
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.
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** *IOCS0-3**	Bi Out	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.
*IORd** *IOWr**	Out 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.
IORDY	In	“I/O channel ready” - perhaps more clearly seen as an active-low request for extra wait states. If you don’t want this facility, this signal must be pulled up or connected to 3.3V. In a DMA mode intended for support of Ultra DMA IDE, this signal becomes a source-synchronous clock for DMA read data.
I/O bus DMA and IDE support		
DMARQ	In	DMA peripheral is ready to transfer data.
*DMACK**	Out	Identifies a cycle as being DMA with a particular peripheral; operates much like an extra chip select.
IODIR	Out	Direction control (high for write) suitable for a ’245 buffer used to buffer the data bus.
*IODEN**	Out	Enable control for an *IOD* data buffer.
Programmable IO signals		
GPIO0-8	Bi	General purpose programmable I/O
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. *GPIn5* is reserved in anticipation of being connected to internal logic to implement a reference clock, for systems which have significant configurability of the master clock rate.

Signal Name	*Type*	*Description*
JTCK	In	JTAG test pins. In FPGA version, double up as device programming inputs.
JTDI	In	
JTDO	Out	
JTMS	In	
Power and generic signals		
GND		System ground
VDD		Power signal - this is a pure 3.3V part
VD5		Separate power for PC bus I/Os. Can be either 5V or 3.3V, to match your PCI signalling environment.

Table 6.1: Signals definitions

6.2. Pinout

Figure 6.1 shows the connections to BONITO. The layout corresponds to a PCB pad layout, looking at it from the BGA's point of view. The table is (inevitably) too big to print comfortably, even with very small print; so we've shortened many signal names, as shown in Table 6.2.

A1 gnd	A2 gnd	A3 pclk	A4 vdd	A5 vd5	A6 ad19	A7 ad16	A8 trdy˜	A9 devsl˜	A10 perr˜	A11 cbe1˜	A12 gnd	A13 ad11	A14 gnd	A15 vd5	A16 ad5	A17 ad2	A18 gnd	A19 iod3	A20 iod6	A21 iod9	A22 iod12	A23 iod14	A24 iod15	A25 gpio0	A26 gnd
B1 cbe3˜	B2 gnd	B3 gnd	B4 ad23	B5 ad20	B6 ad18	B7 cbe2˜	B8 gnd	B9 stop˜	B10 serr˜	B11 ad15	B12 ad14	B13 ad10	B14 vdd	B15 cbe0˜	B16 ad4	B17 vdd	B18 iod0	B19 iod2	B20 iod5	B21 iod8	B22 iod11	B23 gpio1	B24 gpio2	B25 gnd	B26 gnd
C1 ad24	C2 vd5	C3 gnd	C4 ad22	C5 ad21	C6 ad17	C7 frame˜	C8 vdd	C9 vd5	C10 par	C11 vdd	C12 ad13	C13 ad9	C14 vd5	C15 ad7	C16 ad3	C17 ad1	C18 vd5	C19 iod1	C20 vdd	C21 iod7	C22 iod10	C23 iod13	C24 gnd	C25 gpio3	C26 gpio4
D1 ad26	D2 ad25	D3 gnd	D4 gnd	D5 gnd	D6 vdd	D7 irdy˜	D8 gnd	D9 gnd	D10 vd5	D11 vdd	D12 ad12	D13 ad8	D14 gnd	D15 ad6	D16 vdd	D17 gnd	D18 ad0	D19 gnd	D20 iod4	D21 vdd	D22 gnd	D23 gnd	D24 gpio5	D25 gpin0	D26 gpio6
E1 ad29	E2 ad28	E3 ad27	E4 vd5																			E23 gpin1	E24 gpio7	E25 gpin2	E26 gpin3
F1 ad31	F2 ad30	F3 vdd	F4 vdd																			F23 vdd	F24 gpio8	F25 tck	F26 smc
G1 req2˜	G2 req1˜	G3 req0˜	G4 gnd																			G23 gpin4	G24 gpin5	G25 trst	G26 vdd
H1 req4˜	H2 vd5	H3 req3˜	H4 gnd																			H23 iocs0˜	H24 tms	H25 isolat	H26 tdo
J1 gnt1˜	J2 vdd	J3 gnt0˜	J4 req5˜																			J23 gnd	J24 tdi	J25 iocs1˜	J26 iodir
K1 gnt3˜	K2 gnd	K3 gnt2˜	K4 gnd																			K23 gnd	K24 ioden˜	K25 rmcs1˜	K26 iocs2˜
L1 gnt5˜	L2 vd5	L3 gnt4˜	L4 vdd																			L23 vdd	L24 iocs3˜	L25 iord˜	L26 rmcs0˜
M1 irqa˜	M2 reset˜	M3 lock˜	M4 vdd																			M23 ioa1	M24 dmack˜	M25 ioa0	M26 iowr˜
N1 nc	N2 vd5	N3 ma13	N4 gnd																			N23 clckt1	N24 ioa3	N25 clckt0	N26 ioa2
P1	P2 idsel	P3 ma12	P4 vd5																			P23 gnd	P24 ioa4	P25 clckt3	P26 clckt2
R1 ma11	R2 ma10	R3 ma9	R4 ma8																			R23 gnd	R24 clckt4	R25 clockn	R26 vdd
T1 ma7	T2 ma6	T3 ma5	T4 vdd																			T23 vdd	T24 clckt5	T25 dmarq	T26 mdclck
U1 ma4	U2 gnd	U3 ma3	U4 ma2																			U23 iordy	U24 ssctl˜	U25 modein	U26 test1
V1 dcsh0˜	V2 ma1	V3 dcsh1˜	V4 gnd																			V23 test2	V24 test0	V25 evald˜	V26 vccok
W1 dcsl0˜	W2 dcsl1˜	W3 dba1	W4 ma0																			W23 gnd	W24 cprst˜	W25 eok˜	W26 cdrst˜
Y1 vdd	Y2 dwe˜	Y3 dqmbhi	Y4 dras˜																			Y23 int0˜	Y24 srest˜	Y25 vdd	Y26 pvald˜
AA1 dqmblo	AA2 dba0	AA3 ddmxl˜	AA4 vdd																			AA23 vdd	AA24 int1˜	AA25 scmd2	AA26 pmstr˜
AB1 dcas˜	AB2 dcke	AB3 ddmxh˜	AB4 vdd																			AB23 sadc1	AB24 scmd6	AB25 scmd1	AB26 scmd0
AC1 dd31	AC2 gnd	AC3 dd27	AC4 gnd	AC5 dd21	AC6 vdd	AC7 dd14	AC8 gnd	AC9 dd7	AC10 dd3	AC11 vdd	AC12 gnd	AC13 gnd	AC14 sad27	AC15 sad23	AC16 vdd	AC17 sad16	AC18 gnd	AC19 sad10	AC20 sad6	AC21 vdd	AC22 gnd	AC23 gnd	AC24 scmd5	AC25 tmd2	AC26 gnd
AD1 dd30	AD2 dd29	AD3 gnd	AD4 dd24	AD5 dd20	AD6 dd17	AD7 dd13	AD8 dd10	AD9 dd6	AD10 dd2	AD11 vdd	AD12 sad31	AD13 sad30	AD14 sad26	AD15 sad22	AD16 sad19	AD17 sad15	AD18 sad13	AD19 sad9	AD20 sad5	AD21 sad3	AD22 sad0	AD23 sadc0	AD24 gnd	AD25 cpuclk	AD26 tmd1
AE1 gnd	AE2 gnd	AE3 dd26	AE4 dd23	AE5 dd19	AE6 dd16	AE7 dd12	AE8 dd9	AE9 dd5	AE10 dd1	AE11 ddp3	AE12 ddp1	AE13 sad29	AE14 sad25	AE15 sad21	AE16 sad18	AE17 gnd	AE18 sad12	AE19 sad8	AE20 sad4	AE21 sad2	AE22 sadc3	AE23 scmd8	AE24 scmd4	AE25 gnd	AE26 tmd0
AF1 gnd	AF2 dd28	AF3 dd25	AF4 dd22	AF5 dd18	AF6 dd15	AF7 dd11	AF8 dd8	AF9 dd4	AF10 dd0	AF11 ddp2	AF12 ddp0	AF13 sad28	AF14 sad24	AF15 sad20	AF16 sad17	AF17 sad14	AF18 sad11	AF19 sad7	AF20 vdd	AF21 sad1	AF22 sadc2	AF23 scmd7	AF24 scmd3	AF25 gnd	AF26 gnd

Figure 6.1 BONITO (352-pin BGA) pinout

Originally	Abbrev	Originally	Abbrev	Originally	Abbrev
CLOCKIN	clockn	CLOCKOUT0-5	clckt0-5	CPUCOLDRESET~	cdrst~
CPURESET~	cprst~	DDMUXHI~	ddmxh~	DDMUXLO~	ddmxl~
EVALID~	evald~	SysController~	ssctl~	ISOLATE	isolat
MODECLOCK	mdclck	MUXAD0-13	ma0-13	PCIAD0-31	ad0-31
PCICBE0-3~	cbe0-3~	PCICLK	pclk	PCIDEVSEL~	devsl~
PCIFRAME~	frame~	PCIGNT0-5~	gnt0-5~	PCIIDSEL	idsel
PCIIRDY~	irdy~	PCIIRQA~	irqa~	PCILOCK~	lock~
PCIPAR	par	PCIPERR~	perr~	PCIREQ0-5~	req0-5~
PCIRESET~	reset~	PCISERR~	serr~	PCISTOP~	stop~
PCITRDY~	trdy~	PMASTER~	pmstr~	PVALID~	pvald~
ROMCS0-1~	rmcs0-1~	SYSAD0-31	sad0-31	SYSADC0-3	sadc0-3
SYSCMD0-8	scmd0-8	SYSRESET~	srest~		

Table 6.2: Abbreviated names used in pinout diagram Figure 6.1

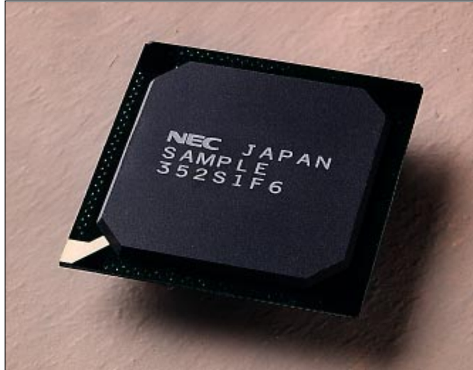
6.3. Package information

The following page - Figure 6.2 - shows the 352-pin BGA package and is borrowed with acknowledgements from NEC's ASIC data book.

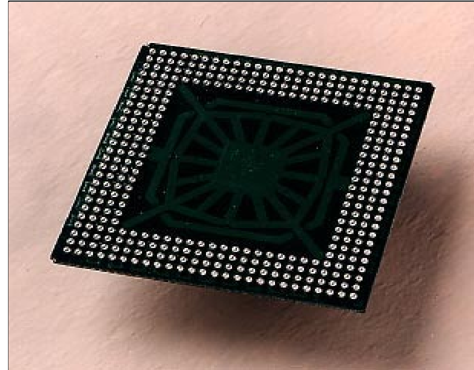
Plastic BGA

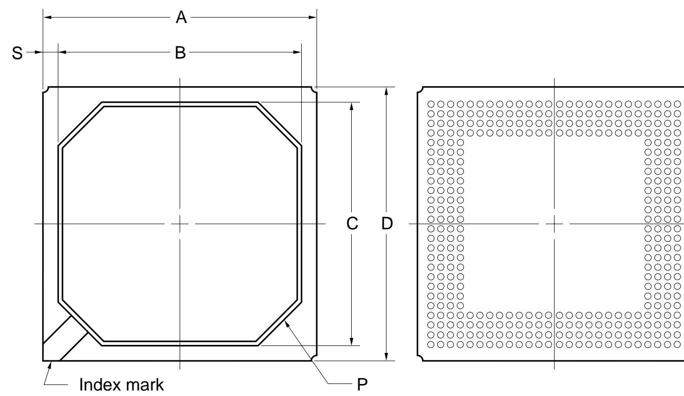
Package Drawing 352-Pin Plastic BGA (35x35mm)

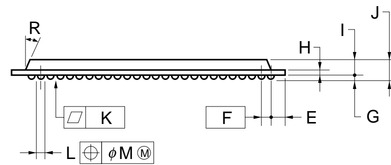
Top View



Bottom View







NOTE

Each lead centerline is located within φ0.3 mm (φ0.012 inch) of its true position (T.P.) at maximum material condition.

ITEM	MILLIMETERS	INCHES
A	35.0±0.2	1.378±0.008
B	30.0	1.181
C	30.0	1.181
D	35.0±0.2	1.378±0.008
E	1.62	0.064
F	1.27 (T.P.)	0.050 (T.P.)
G	0.6±0.1	0.024 $^{+0.004}_{-0.005}$
H	0.56	0.022
I	1.73±0.15	0.068±0.006
J	2.33±0.25	0.092 $^{+0.010}_{-0.011}$
K	0.15	0.006
L	φ0.75±0.15	φ0.03 $^{+0.006}_{-0.007}$
M	0.3	0.012
P	C 4.0	C 0.157
R	30°	30°
S	2.5	0.098

10

S352S1-F6-1

Figure 6.2 352-pin BGA informatino for BONITO

Index

TBA.

Appendix A: Register Addresses

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

Appendix B: FPGA prototype of BONITO

The FPGA prototype is built in a Xilinx “Vertex” device. It has some big differences:

- It’s in a bigger package - a 432-pin BGA instead of a 352-pin. The pinout is completely different.
- It implements none of BONITO’s boundary and other signal test features, since it can rely on those built into the FPGA.
- It is a soft-loaded part, requiring a logic program exceeding 2Mbits in size. This is loaded at about 60MHz, and is completed quite rapidly - but from power-up it is in the Xilinx-defined reset condition with most I/Os floating.

  We’ve avoided re-using any signals used in the loading process.

Appendix C: BONITO's debug interface