architecture handbook

XR/17032
XR/17032
XR/17032
XR/17032
XR/17032
XR/17032
XR/17032
XR/17032

XR/17032 Architecture Handbook
Revision 1.0, October 26, 2021
Revision 2.0, December 10, 2023

Table of Contents

1. Overview

1.1. Introduction

XR/17032 is a 32-bit RISC architecture. This document describes it informally and is meant to be used as a reference handbook; it is intended for readers who are already somewhat familiar with computer architecture, or who are at least familiar with computer programming and have good independent research skills. This handbook need not be read in order; the reader is encouraged to skip around as unfamiliar terms appear. A brief description of the architecture follows.

The architecture's registers and the virtual address space are 32-bit, and 32-bit values are the widest that it can load or store. It has no floating-point operations, and 64-bit wide arithmetic must be synthesized from smaller operations. This architecture is intended to be relatively simple; it includes only 60 instructions. In the interest of supporting fancy operating system design, XR/17032 supports paged virtual addressing, as well as a distinction between user mode and kernel mode. Like most RISCs, memory accesses must be aligned to their size or else they will incur an exception, and, likewise, all instructions are 32 bits (4 bytes) wide and must be aligned to 32-bit boundaries. The architecture is little-endian.

The architecture defines 32 general purpose registers (GPRs), usable by any instruction that takes register operands. They are each 32 bits wide. The zeroth GPR almost always reads as zero and ignores writes. This is a common RISC design tactic that simplifies the encoding of many instructions.

The architecture defines 32 control registers (CRs). They are each 32 bits wide. As their name suggests, CRs are used to control the behavior of the processor, and are therefore only accessible via the privileged kernel mode instructions MTCR and MFCR. See Section 4 for a more detailed description of each control register.

#	Name	ABI Assignment
0	zero	Always reads as zero, ignores writes.
1-6	t0-5	6 temporary registers (caller-saved).
7-10	a0-3	First 4 arguments and return values (caller-saved).
11-28	s0-17	18 local variable registers (callee-saved).
29	tp	Thread-local storage pointer.
30	sp	Stack pointer.
31	lr	Link register.

All general purpose registers and their ABI assignments.

1.2. Reset

When the processor is reset, for instance during a reboot or a power-on, the RS control register is cleared to zero. The processor is thus forced to kernel mode, virtual address translation is disabled, exposing the physical address space. The program counter is set to the address `0xFFFE1000`, with the idea that a boot ROM is located at `0xFFFE0000` and is followed by an initial 4096 (0x1000) byte exception block.

2. Virtual Addressing

2.1. Introduction

For many reasons, it is useful to be able to dynamically re-map sections of the address space to other regions of physical memory, thereby creating a "virtual address space". A few such reasons are listed below:

1. Process isolation: Programs can be completely protected from eachother by giving them their own unique address spaces.
2. Demand zero: The allocation of memory can be delayed until it is actually needed.
3. Memory-mapped files: Files on disk can be mapped into the virtual address space, giving the illusion that they are a range of bytes that can be accessed like any other memory.
4. Shared memory: Popular physical memory (such as executable code) can be shared among multiple virtual addresses in multiple address spaces, thereby saving memory.
5. Virtual memory: Disk space can be transparently used as extra memory via swapping.

The mechanism that the XR/17032 architecture uses for this is *paged* virtual addressing, also known as "paging". In a paging scheme, the virtual address space is divided into evenly sized "pages" which can be individually re-mapped to arbitrary physical addresses. As this is a 32-bit architecture, the virtual addresses are 32 bits, leading to a 4GB virtual address space. The only supported page size on the processor is 4096 bytes, or 4KB. This means that the virtual address space is evenly tiled by 1,048,576 pages.

There is now a question of how to achieve this translation. If the translation of the virtual page to the physical page is performed by looking up a physically linear page table, with 32-bit table entries, it would therefore consume 4MB of memory per process, which is obviously unacceptable overhead.

In many architectures, such as fox32 and i386, the virtual address space is therefore managed by a two-level page table. The indices into the two levels of the page table are extracted from bit fields of the 32-bit virtual address.

The two 10-bit fields [22:31] and [12:21] contain the index into the level 2 table and the level 1 table, respectively.

As these indices are 10 bits, and the entries are 4 bytes wide, these tables are both 4096 bytes in size.[1]

To translate a virtual address, the level 2 table is indexed first, yielding a 32-bit entry that contains the physical address of the level 1 table. This level 1 table is indexed next, yielding the 20-bit page frame number to which this virtual page is mapped. The 12-bit byte offset is appended to this, yielding the final 32-bit physical address to which the memory access should be performed. Note that each address space needs its own level 2 page table, which may point to up to 1024 level 1 page tables, which each map 1024 virtual pages to physical pages.

[1] Note that this is one reason for the usage of the 4KB page size: this is the page size for which the division of the virtual address into these three fields causes the tables to consume single page frames, which simplifies memory management.

This scheme allows the omission of large sections of the page table that are not needed. In practice, virtual address spaces tend to be very sparse, so this usually reduces the original 4MB page table overhead to a mere handful of kilobytes per process.

However, there is one major problem: you now have to perform two extra memory accesses for each memory access! The solution to this is, as with many things in computer science, a cache: processors that employ this scheme contain a translation buffer, or TB,[2] which is a small memory typically containing 8 to 64 cached page table entries. The TB is usually fully associative, meaning that it can be indexed directly by virtual address; the virtual page number is compared simultaneously with all of the entries in the TB, and if any of them contain a matching entry, it is returned. This can easily be done within a single cycle, and a hit in the TB avoids the cost of looking up the page tables.

In the case that a needed virtual page translation is not cached in the TB, a "TB miss" occurs. On architectures like fox32 and i386, this results in a page table walk done automatically by the hardware, which then inserts the page table entry in the TB. The instruction is then transparently re-executed and hopefully succeeds this time.

Two major problems remain:

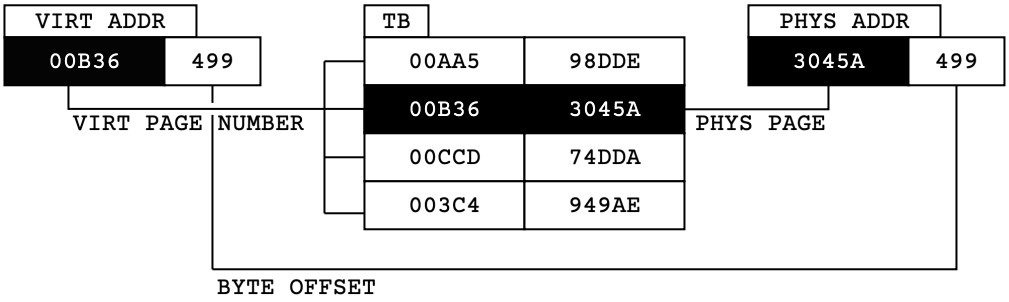
1. Complicated: The logic to perform a page table lookup in hardware is quite complex; it takes up many extra gates on the chip and can be difficult to debug during the development of prototype hardware.
2. Inflexible: If the system software wishes to manage its own custom paging structures, it is out of luck. It must use the hardware-enforced page tables.

For these reasons, XR/17032 instead uses a "software refill" design. In such a design, the management of the TB is exposed directly to software. When a TB miss occurs, an exception is taken, which redirects execution to a software routine which looks up the paging structure, loads the page table entry (PTE), and writes it into the TB. This exception handler returns, causing the re-execution of the original instruction, which hopefully succeeds this time.

In a software refill scheme, the system software has the ability to implement any paging structure it sees fit, and much complexity is removed from the hardware logic.

You can see this as the previously mentioned paging scheme "flipped on its head" – instead of a two-level page table being the primary governor of paging and the TB existing only as a nearly-transparent cache for it, the fixed-size TB is the first class citizen. The TB contains all of the currently valid mappings, and needs to be manually refilled from some other paging structure (such as a two-level page table).

[2] Also called a "translation lookaside buffer" or TLB.



In the example above, there is a 4-entry TB, containing entries for the virtual page numbers `00AA5`, `00B36`, `00CCD`, and `003C4`. The program references a virtual address `00B36499`, which is provided as the input to the TB. The page number, `00B36`, is compared with all entries in the TB simultaneously. Luckily, one of the entries matches, and produces the physical page number `3045A`. The byte offset from the original virtual address is appended to this physical page number, producing the final physical address with which the processor will perform the memory access.

Had there not been a matching entry in the TB, a TB miss exception would have occurred. The TB miss handler would have inserted the correct entry into the TB, and the original instruction would have re-executed; beginning this process again, but matching in the TB this time and succeeding.

2.2. The Translation Buffers

The XR/17032 architecture has two TBs. In fact, it could be seen as having two MMUs; an IMMU and a DMMU, providing translations for instruction fetch and data access respectively. This is to simplify pipelined implementations where a FETCH and MEMORY stage may want to access the TB to translate a virtual address simultaneously. The TB management scheme is architected such that the actual size of each TB is transparent to system software and may vary from processor to processor.

When the M bit is set in the RS control register (see Section 4.2), instruction fetches are translated by the ITB, and data accesses are translated by the DTB. The TB entries are each 64 bits wide. The upper 32 bits contain the TBTAG, which is the 12-bit ASID (Address Space ID) and 20-bit VPN (Virtual Page Number) that will match the TB entry. The lower 32 bits contain the TBPTE, containing the 20-bit physical page number along with some flag bits. The TBPTE is the "preferred" format for a page table entry. See Section 4.13 for the format of the TBTAG, and Section 4.4 for the format of the TBPTE.

Something important to note is the difference between a TB miss exception, and a page fault exception. A TB miss exception occurs when a key consisting of a VPN and the current ASID fails to match in the TB. A page fault occurs when it *does* match, but matches to a PTE whose V (valid) bit is clear. This is behavior that differs significantly from other architectures like i386: it is possible to have a TB entry that matches a virtual page, but is invalid and causes a page fault.

This seemingly strange behavior makes more sense when you recall that we perform software TB miss handling. This behavior makes it possible to perform an optimization in which an invalid PTE can be "blindly" inserted into the TB, the faulting instruction can be re-executed, and a page fault then occurs. If this were not the case, the TB miss handler would need to have a branch to make sure that the PTE is valid before it inserts it into the TB, and branch to the page fault handler if it isn't.

Adding a branch to what may be the hottest codepath in the entire system is a bad plan, as opposed to allowing invalid PTEs to match in the TB.

Note that this has implications on TB management. A page must be flushed from the TB not only when it is transitioned from valid to invalid, but also when it is transitioned from invalid to valid. Otherwise a stale TB entry may continue to track the page as invalid, causing erroneous page fault exceptions when it is accessed.

2.2.1. Address Space IDs

To describe ASIDs, it is useful to describe the problem they solve. Most multitasking operating systems operate under a scheme where each process has its own isolated address space. As a result, when a context switch occurs, the address space is also switched. The contents of the TB are irrelevant to the new address space. In some architectures, this necessitates flushing the entire contents of the TB, which incurs many extra expensive TB misses and is therefore quite wasteful.

One trick that helps alleviate some of the burden is to add a G bit, or global bit, to the page table entry. This indicates that any entry for that page should be left in the TB upon address space switch, and is useful to set for globally shared mappings such as those in kernel space. However, this solution still isn't perfect, as you still lose many TB entries from other processes' userspace that may have been useful to have after switching back to that process.

One extra feature that you can place atop G bits is the concept of an address space ID, or ASID. Each TB entry has a 12-bit ASID associated with it, along with the virtual page number. If the G bit is clear in a TB entry, it will only match a virtual address if the current ASID stored in the ITBTAG or DTBTAG control register (depending on which TB it is[3]) is equivalent to the one stored in the TB entry. If the G bit is set, the TB entry will match the virtual address regardless of the current ASID; that is, it will match in all address spaces.

By assigning a different ASID to each process, you can now have the TB entries for multiple address spaces residing in the TB simultaneously without fearing virtual address collisions, and can completely avoid flushes on context switch. If it helps, you can logically think of the ASID as being an extra 12 bits on the virtual address, in order to differentiate identical virtual page numbers that belong to different address spaces. If this doesn't help, then forget that sentence and try to live the rest of your life in bliss.

A very important note is that setting an ASID of 0xFFF (4095) in ITBTAG or DTBTAG should never be done and will lead to unpredictable results, because this particular value is used internally to denote an "unused" TB entry. Using this ASID could cause spurious TB hits and bizarre behavior.

2.2.2. Translation Buffer Invalidation

See Section 4.5 for a thorough explanation on how to invalidate entries in the TB by writing to the ITBCTRL and DTBCTRL control registers.

2.3. Translation Buffer Miss

As explained earlier, the XR/17032 architecture invokes a software exception handler when a TB miss occurs. There are two TB miss exception vectors, one for ITB miss and one for DTB miss (see

[3]Note that in general, the ASID field should be the same in the ITBTAG and DTBTAG control registers; it's hard to imagine a situation where it would be useful for them to differ. However, this is not prohibited.

Section 4.3 for the exact offsets). This makes the miss handlers shorter as they do not need to figure out which TB to insert the entry into; there can simply be a distinct miss handler which deals only with that TB.

The behavior of the processor when a TB miss occurs is contingent on whether the T bit was set in the RS control register's current mode bits. This bit is also set by a TB miss, so in reality, it is contingent on whether the TB miss is "nested" within another TB miss or not. The reason you would want to take a TB miss within a TB miss handler will be elucidated later.

There are also some special cases for page faults that occur while the T bit is set. Note that this behavior essentially causes the new page fault to look like a page fault on the original virtual address that missed in the TB, instead of a page fault on the virtual address referenced by the TB miss handler.

Aside from the special cases in TB miss and page fault handling, there is another major effect of the T bit being set, which is that the ZERO register is no longer hardwired to read all zeroes. It can therefore be used freely as a scratch register by TB miss routines, without needing to be saved or restored.

In either case, the low 20 bits of the ITBTAG or DTBTAG control register are automatically filled with the virtual page number of the virtual address that failed to match in the TB. This also shortens the TB miss handler, because it doesn't need to assemble the upper 32 bits of the TB entry: the appropriate ASID for the mapping (the same as the current one) is already set, and now so is the appropriate VPN. It only needs to load the PTE for the mapping and insert it into the TB by writing it to either the ITBPTE or DTBPTE control register. The upper 32 bits of the resulting TB entry are taken from the current value of ITBTAG or DTBTAG, and the lower 32 bits are taken from the PTE written to the control register.

The index into the TB that is overwritten with the new entry is taken from the control register ITBINDEX or DTBINDEX, which is then automatically incremented, creating a FIFO behavior for TB replacement. When the replacement index reaches the end of the TB, it wraps back to 4 instead of 0. This means that entries [0-3] will never be replaced naturally, and are permanent or "wired" entries that can be used to create permanent virtual page mappings for any purpose.[4]

With all of this information, you can now imagine a TB miss handler which does a manual page table walk; it calculates an offset within the level 2 page table and loads the level 2 PTE, decodes this to get the address of the level 1 page table, and then loads the level 1 PTE. The level 1 PTE can be written to the appropriate ITBPTE or DTBPTE control register to write the TB entry, and then the miss routine can return.

This scheme would work, and is in fact used by the AISIX kernel, which runs with memory mapping disabled in kernel mode. However, it has several significant issues:

[4] As an example of the usage of wired entries, system software will typically map the exception block (see Section 4.3) permanently with one wired entry of the ITB to avoid taking an ITB miss on the ITB miss handler, which for obvious reasons is an unrecoverable situation.

1. It requires access to the physical address space. As memory mapping is not disabled when an exception is taken, you would have to ensure that the exception block is identity mapped. This would allow you to disable paging on the fly within the exception block, perform the miss handling, and then return.
2. Two memory loads must be done in all cases.
3. It is quite a lengthy codepath, and requires several branches.

It is possible to do much better. In fact, it is possible with a small amount of extra work in the architecture and system software to accomplish a two-level page table TB miss routine which looks like this, as taken from the MINTIA executive:

```
DtbMissRoutine:
  mfcr zero, dtbaddr
  mov  zero, long [zero]
  mtcr dtbpte, zero
  rfe
```

At a mere four instructions, with zero branches, this is fairly close to optimal. In essence, this routine works by running in the virtual address space and loading the PTE directly from a linear page table, and then writing it to the TB. Unfortunately, explaining how this works requires some labor.

To begin, we have to understand the concept of a "virtually linear page table". It turns out that placing the level 2 page table as an entry into itself creates a region of virtual address space which maps the two-level page tables as if they were a linear array indexed by virtual page number.[5] The reason for this is that accessing memory within this region causes the level 2 page table to be treated as a level 1 page table, and so all of its entries directly map the level 1 page tables. The level 2 page table itself can also be found within this region.

```
// Assume INDEX is a constant containing the index of the level 2 page
// table that has been set to create the virtually linear page table
// mapping. Any index can be chosen, to place the linear page table
// within the address space as desired.

// Since each level 1 page table maps 1024 pages of 4096 bytes each,
// the following formula can be used to find the base address. Note that
// it is always 4 megabyte aligned.

LinearPageTableBase := INDEX * 1024 * 4096

// Since each level 1 page table is mapped as a 4096 byte page within
// the virtually linear page table, the following formula can be used
// to find the address of the level 2 table itself.

Level2Table := LinearPageTableBase + INDEX * 4096
```

Pseudocode for calculating the base address of the virtually linear page table.

[5] This is also sometimes referred to as "recursive mapping" or "recursive paging".

The architectural support provided for loading the PTE out of a virtually linear page table comes in the form of the ITBADDR and DTBADDR control registers. When a TB miss occurs, the low 22 bits of this control register are filled with the virtual page number of the missed address, shifted left by 2. If the upper 10 bits of the control register was previously filled with the 4 megabyte aligned base address of the linear page table, then upon a TB miss, this control register will contain the address from which the PTE can be loaded. This saves several instructions that would otherwise be required to calculate this address.

There are now two cases that are concerning. The first is the case where the page table in which the PTE resides is not present in the DTB. A nested TB miss will occur upon an attempt to load the PTE. This sounds like it would always be a fatal condition, until three facts are recalled from earlier in this chapter:

1. The TB has support for 4 "wired" or permanent entries which are never replaced.
2. The PTE address for the miss on the page table page will always reside within the level 2 page table.
3. There is special cased behavior for TB misses, enabled by the T bit of RS which is set when a TB miss is taken.

If system software maps the level 2 table page with one wired entry of the DTB, this will provide an "anchor point" which will halt the chain of DTB misses. The nested DTB miss will load the level 2 page table entry for the page table and insert it into the DTB. Due to the special cased behavior of nested TB misses, the exception state of the original TB miss was left completely intact, and so the nested TB miss will return directly to the instruction that caused the original miss. This instruction re-executes, and misses again, as the original page it needed is still not in the TB. However, the TB miss handler will now succeed in loading the PTE from the virtually linear page table, as we inserted the page table page into the DTB during the nested TB miss earlier.

Careful readers will now understand how the four instruction TB miss routine from earlier works. You may also note that this scheme has an extra benefit, whereby only one memory access is needed to load the PTE from the two-level page table, as long as the containing level 1 page table is already present in the DTB. Note that the nested TB miss which loads the level 1 page table into the DTB does not require any special code, the processor merely (re-)executes the exact same normal DTB miss handler.

There is one small snag, which is the second concerning case from earlier. If the level 1 page table does not actually exist, then the nested DTB miss will load an invalid level 2 page table entry into the DTB. In this case, a page fault will occur in the TB miss handler when it attempts to load the PTE from the level 1 page table again. The special cased page fault behavior listed earlier addresses this case, by clearing the T bit and setting EBADADDR to the value of TBMISSADDR. It also keeps the exception state intact in a similar manner to the nested TB miss special case. The page fault exception handler is thereby "fooled" into thinking that the original instruction caused a page fault on the original missed address.[6]

[6]Note that a processor implementation must keep a latch somewhere that remembers whether the last non-nested TB miss (the last one that occurred while the T bit was clear) was caused by a read or write instruction, so that this page fault case will result in the appropriate page fault exception.

TB Miss Exception Behavior	
T=0	T=1
The TBMISSADDR control register is set to the missed virtual address.	The TBMISSADDR control register is left alone.
Normal exception logic occurs; the RS mode stack is pushed. However, the exception program counter is saved in the TBPC control register instead of the EPC control register.	None of the normal exception logic occurs except to redirect the program counter to the appropriate exception vector. The RS mode stack is not pushed.
The T bit is set.	The T bit remains set.

The effect that the T bit in the RS control register has on the behavior of a TB miss exception.

Page Fault Exception Behavior	
T=0	T=1
The EBADADDR control register is set to the faulting address.	The EBADADDR control register is set to the value of the TBMISSADDR control register.
A Page Fault Read exception is triggered if the access was a read, or Page Fault Write otherwise.	If the last TB miss exception that occurred while the T bit was clear was a read, a Page Fault Read exception is generated, otherwise Page Fault Write.
Normal exception logic occurs. The RS mode stack is pushed.	None of the normal exception logic occurs except to redirect the program counter to the appropriate exception vector. The RS mode stack is not pushed. The T bit is cleared. The EPC control register is set to the value of the TBPC control register.

The effect that the T bit in the RS control register has on the behavior of a page fault exception.

3. Memory Caching

3.1. Introduction

Modern computer systems contain memory subsystems that produce results in a time several factors slower than the processing unit can accept them, creating a substantial performance bottleneck. The general solution to this is to add fast "cache" memory close to the processor, in which frequently or recently used memory is kept, and waits upon the machine's memory subsystem to respond can be avoided.

The XR/17032 architecture contains a "split cache" scheme, where instruction bytes and data bytes are cached separately in an Icache and Dcache, respectively. The software-visible effects that these caches have are as follows:

1. The Icache is never automatically kept in sync with the Dcache or with the contents of memory. If the instruction stream is written into, for instance due to copying a program in memory, the Icache must be manually flushed. Otherwise, stale instruction bytes may be executed, causing problems that are hard to diagnose.
2. Depending on the platform, the Dcache may or may not be kept in sync with external device activity (i.e. a DMA transfer into memory from a disk controller). If it isn't, then manual Dcache flushes are required after these events, or stale data bytes might be seen.
3. This is an explicit statement of something that must *not* be visible to software: in a multiprocessor system, the Dcache of each processor *must* be kept in sync with the Dcache of all other processors in the system through some coherency protocol.

Due to these issues, among other reasons, the paging architecture includes an N bit in the PTE format which indicates that accesses to that page should bypass the Dcache. This bit should be used when mapping pages containing device registers for driver access.

While virtual address translation is disabled, for instance at system reset, the cache is bypassed for all accesses to physical addresses at or above `0xC0000000` (3GB). For this reason, it is advisable for a platform to place device registers in this region of the physical address space, to allow boot firmware to easily manipulate them. It is also advised to immediately copy the boot firmware from the ROM in high memory to RAM in low memory and execute it from there instead, or else it will execute noncached (that is, extremely slowly).

For detailed information on how to flush either a single page or the entirety of the Icache or Dcache, see Section 4.6.

3.2. The Caches and XR/computer Systems

This information is included here for quick reference, and strictly speaking, belongs in these systems' respective manuals.

Neither the XR/station desktop, the XR/MP deskside server, nor the XR/frame minicomputer keep Dcache coherency with device activity. When a DMA transfer completes, the system software must be sure to flush the Dcache appropriately.

On multiprocessor configurations such as the XR/MP and XR/frame systems, the processor that handles an I/O request completion must be sure to send an IPI (inter-processor interrupt) to the other processors to ensure that they flush their Dcache as well (a "Dcache shootdown").

4. Processor Control

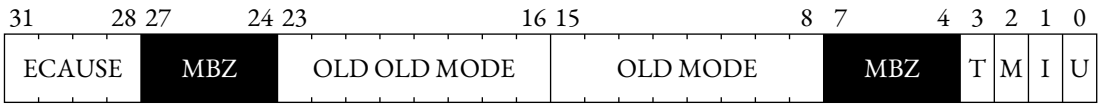
4.1. Introduction

The behavior of the processor is primarily controlled by a small set of control registers (CRs). They are explained below. Note that some CRs whose function is very similar but whose names differ by only a character are de-duplicated into a single section whose name omits that character. For example, DCACHECTRL and ICACHECTRL are both explained in the CACHECTRL section.

#	Name	Function
0	RS	Current and previous processor mode bits.
1	WHAMI	Unique ID for this processor in a multiprocessor system.
5	EB	Exception block base address.
6	EPC	Program counter before the last exception.
7	EBADADDR	Bad address that triggered the last exception (if relevant).
9	TBMISSADDR	Bad address that triggered the last TB miss exception.
10	TBPC	Program counter before the last TB miss exception.
11-15	SCRATCH0-4	Permanently reserved for arbitrary system software usage.
16	ITBPTE	Lower 32 bits of an entry to insert in the ITB. Causes ITB insertion when written.
17	ITBTAG	Upper 32 bits of an entry to insert in the ITB. Doubles as the current ASID, and the VPN of the last virtual address that missed in the ITB.
18	ITBINDEX	Next replacement index for the ITB.
19	ITBCTRL	Causes ITB invalidations when written.
20	ICACHECTRL	Yields Icache size parameters when read, causes Icache invalidations when written.
21	ITBADDR	Pre-calculated virtual PTE address for use upon ITB miss.
24	DTBPTE	Lower 32 bits of an entry to insert in the DTB. Causes DTB insertion when written.
25	DTBTAG	Upper 32 bits of an entry to insert in the DTB. Doubles as the current ASID, and the VPN of the last virtual address that missed in the DTB.
26	DTBINDEX	Next replacement index for the DTB.
27	DTBCTRL	Causes DTB invalidations when written.
28	DCACHECTRL	Yields Dcache size parameters when read, causes Dcache invalidations when written.
29	DTBADDR	Pre-calculated virtual PTE address for use upon DTB miss.

All defined CRs. Absent CR numbers have undefined behavior if read or written.

4.2. RS (Processor Status)



The RS control register contains a three-deep "stack" of mode bits (the top of which contains the primary mode bits that control the state of the processor). The four ECAUSE bits identify the cause of the last exception.

When an exception is taken, several of the mode bits are overwritten to place the processor into kernel mode with interrupts disabled. Because of the need to return from an exception with the state of the processor intact, there is a need to save the previous mode bits somewhere. In some CISC designs, like fox32[7], the old state is saved onto the stack.

Instead of performing memory access during exception dispatch, there is a small "stack" of mode bits within the RS control register. When an exception is taken, the current mode is shifted left by 8 into the "old mode", which itself is shifted left by 8 into the "old old mode", effectively performing a "push" onto a small stack. Any mode bits in the "old old mode" at this time are destroyed. Manually saving RS in the exception handler is required if more than three levels of nesting are needed. The RFE (Return From Exception) instruction atomically reverses this, popping the old mode bits into the current mode bits.

The reason that this stack is three-deep is to account for this case:[8]

1. Normal processing is occurring.
2. An exception occurs, pushing the original state into the old mode.
3. A TB miss exception is taken during the exception handler, before it can save RS, pushing the original state into the old old mode.

Modifying the current mode bits must be done with a read-modify-write procedure; that is, if one wished to enable interrupts, they would need to read RS into some register, set the I bit, and then write the contents of that register back into RS. The same principle applies to the other mode bits.

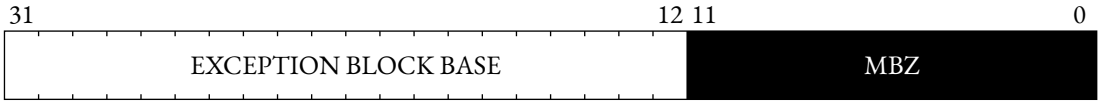
	Function
U	Usermode is active. Privileged instructions, which all have a major opcode of 101001, are forbidden and will produce a privilege violation exception if executed.
I	External device interrupts are enabled.
M	Paged virtual addressing is enabled. The ITB and DTB are looked up to translate instruction fetches and data accesses, respectively (see Section 2).
T	A TB miss is in progress. The ZERO register is usable as a GPR; it is not wired to read all zeroes while this bit is set. This frees it as a scratch register for TB miss routines. This bit also has effects on exception handling, which are enumerated in Section 2.3.

The function of the mode bits when set.

[7] fox32 is a trademark of Ryfox Computer Corp.

[8] The mode stack was once two-deep but was grown to three when software TB miss handling was introduced to the architecture, as this case would otherwise destroy the saved state in the old mode bits. Note also that the mode stack itself is an idea borrowed from the MIPS architecture's SR cop0 register.

4.3. EB (Exception Block Base)



The EB control register indicates the base address of the exception block.

When an exception is taken by the processor, the program counter must be redirected to an exception handler. Some architectures use a table of exception vectors, which is indexed and loaded by the processor in order to determine whether to jump. However, as this is a RISC architecture, memory accesses during exception dispatch are unacceptable complexity.

Instead, upon exception, the PC is redirected to an offset within the "exception block". The offset is calculated using the ECAUSE code of the exception, which is a 4-bit number between 0 and 15.

The exception block occupies exactly one page frame, which is 4096 bytes in size. As there are 16 possible exception codes, each vector has room for 256 bytes, or 64 instructions. This provides enough room to handle simple cases of exceptions, such as TB misses, without needing to branch outside of the exception block.

The new program counter is calculated by EB | ECAUSE << 8, and so the base address of the exception block must be page-aligned, that is, the low 12 bits must be zero, otherwise garbage may be loaded into PC.

Note that as the TB miss handlers themselves reside in the exception block, the system software must place the exception block page into a wired TB entry before virtual addressing is enabled, since the processor will not survive taking an ITB miss on the ITB handler. This also avoids costly TB misses when taking exceptions. See Section 4.14 or Section 2.2 for more details on wired TB entries.

#	EB	Name	Occurrence
1	+100	INT	An external interrupt has occurred.
2	+200	SYS	A SYS instruction has been executed.
4	+400	BUS	A bus error has occurred, usually caused by a non-existent physical address being accessed.
5	+500	NMI	Non-maskable interrupt.
6	+600	BRK	A BRK instruction has been executed.
7	+700	INV	An invalid instruction has been executed.
8	+800	PRV	A privileged instruction was executed in usermode.
9	+900	UNA	An unaligned address was accessed.
12	+C00	PGF	A virtual address matched to an unsuitable PTE in the TB during a read, causing a page fault.
13	+D00	PFW	A virtual address matched to an unsuitable PTE in the TB during a write, causing a page fault.
14	+E00	ITB	A virtual address failed to match in the ITB during instruction fetch.
15	+F00	DTB	A virtual address failed to match in the DTB during data access.

All defined ECAUSE codes. Absent codes are reserved.

4.4. TBPTE (Translation Buffer Page Table Entry)

When written, the ITBPTE and DTBPTE control registers will cause an entry to be written to the ITB or DTB, respectively. The upper 32 bits of the entry are taken from the current value of ITBTAG or DTBTAG, and the lower 32 bits are taken from the value written to this control register.

The low 32 bits of a TB entry are its "value", indicating the page frame that the virtual page maps to. The upper 32 bits, TBTAG, are its "key", containing the "matching" ASID and the virtual page number mapped by the entry. Note that the low 32 bits form a preferred format for page table entries, hence the name of this control register. See Section 2 for more information.

	Function
V	The translation is valid. If this bit is clear, accesses within the page will result in the appropriate page fault exception.
W	The page is writable. If this bit is clear, any write within the page will result in a PFW (Page Fault Write) exception.
K	The page may only be accessed while the processor is in kernel mode. Accesses from user mode will result in the appropriate page fault exception.
N	Accesses within this page should bypass the caches and go directly to the bus. This is most useful for memory-mapped IO.
G	This translation is global; the virtual page number will match this entry, regardless of the current ASID.

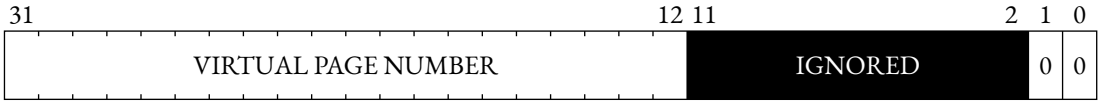
The meaning of the PTE bits when set.

4.5. TBCTRL (Translation Buffer Control)

TBCTRL WRITE: CLEAR ALL

TBCTRL WRITE: CLEAR PRIVATE

TBCTRL WRITE: CLEAR NON-WIRED



TBCTRL WRITE: INVALIDATE ONE

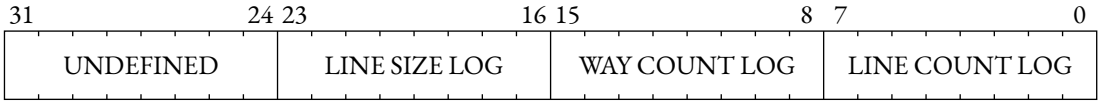
Writes to ITBCTRL and DTBCTRL can be used to invalidate entries in the ITB or DTB, respectively. The 32-bit value written to the control register should be in one of the three formats enumerated above, distinguished by the low two bits. Any other combination of low bits will yield unpredictable results.

Note that reads from these control registers yield unpredictable (non-useful!) results. If one wishes to determine the size of the ITB or DTB, they can set ITBINDEX or DTBINDEX to zero, and write values to ITBPTE or DTBPTE until they see the replacement index wrap. The last value of the replacement index before it wraps, plus one, is the size of that TB.

	Function
11	Every entry in the TB is cleared, including entries with the G bit set. Note that in general, this should not be done while virtual address translation is enabled, as this may clear wired entries like the exception block, and any TB miss taken will then be fatal.
10	Clear all non-wired private entries from the TB, i.e., non-wired entries with the G bit clear.
01	Clear all non-wired entries from the TB, including global entries.
00	Clear all TB entries that map the given virtual address. ASIDs are ignored; if there are multiple TB entries with the same virtual address but different ASIDs, they will all be cleared.

The function of each TBCTRL command.

4.6. CACHECTRL (Cache Control)



CACHECTRL READ

CACHECTRL WRITE: CLEAR ALL

CACHECTRL WRITE: CLEAR PAGE

Reads from the ICACHECTRL and DCACHECTRL control registers yield a 32-bit value whose bit fields indicate the parameters of the Icache and Dcache respectively; the number of lines in the cache, the number of ways (i.e. the set associativity) of the cache, and the size of a cache line are each given, in the form of a binary logarithm. I.e., if the line count field contains 8, then there are 256 lines in the cache.

Writes to the ICACHECTRL and DCACHECTRL control registers cause various invalidations to occur. The 32-bit value written to the control register should be in one of the two formats enumerated above, distinguished by the low two bits. Any other combination of low bits will yield unpredictable results.

	Function
11	Every line in the cache is invalidated. This is useful for, for example, keeping the Icache coherent after things like dynamic linking that modify the instruction stream in ways that are hard to predict.
10	Every line in the cache that is logically within the given page frame is invalidated. This is useful for, for example, keeping coherency within the Dcache after a DMA operation.

The action of each CACHECTRL command.

4.7. WHAMI (Who Am I)

In a multiprocessor system, WHAMI contains a numeric ID which is unique to each processor in the system. It should be in a range of [0, MAXPROC-1], where MAXPROC is the maximum number of processors supported by the platform. Therefore, on a uniprocessor system, it should always contain zero.

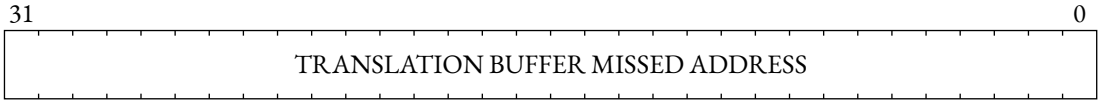
4.8. EPC (Exception Program Counter)

When an exception is taken, the current program counter is saved into EPC. The RFE instruction restores the program counter from this control register, atomically with restoring the mode bits (see Section 4.2).

4.9. EBADADDR (Exception Bad Address)

When a bus error or page fault exception is taken, EBADADDR is filled with the physical or virtual address, respectively, that caused the exception.

4.10. TBMISSADDR (Translation Buffer Missed Address)



When a TB miss exception is taken, and the T bit is not set in RS, TBMISSADDR is filled with the virtual address that failed to match in the TB. If the T bit is set, however (i.e., the processor is already handling a TB miss), this CR is left alone. This CR, therefore, is not affected upon a nested TB miss exception, and always contains the missed virtual address that caused the first one.

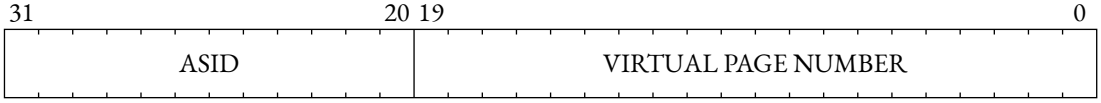
4.11. TBPC (Translation Buffer Miss Program Counter)

When a TB miss exception is taken, and the T bit is not set in RS, the current program counter is saved into TBPC. If the T bit is set, however (i.e., the processor is already handling a TB miss), this CR is left alone. This CR, therefore, is not affected upon a nested TB miss exception, and always contains the program counter that caused the first one. Additionally, if the T bit is set when the RFE instruction is executed, it will restore the program counter to the value of TBPC rather than that of EPC, allowing instant return to the original faulting instruction without having to potentially unwind several levels of nested TB misses. See Section 2.3 for more details.

4.12. SCRATCH0-4 (Arbitrary Scratch)

The system software can use the SCRATCH0 through SCRATCH4 control registers for anything. They are fully readable and writable and do not perform any action. The intended usage is to save general purpose registers to free them up as scratch within exception handlers, but other usages are also possible.

4.13. TBTAG (Translation Buffer Tag)

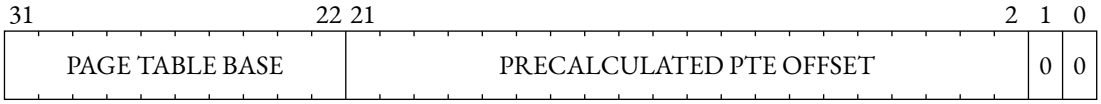


The ITBTAG and DTBTAG control registers contain the current ASID (Address Space ID), and the last virtual page number that incurred a TB miss. This control register also doubles as the uppermost 32 bits of the entry that is written to the TB when a write occurs to the ITBPTE or DTBPTE control register (see Section 4.4 and Section 2.3).

4.14. TBINDEX (Translation Buffer Index)

The ITBINDEX and DTBINDEX control registers contain the next replacement index for the ITB and DTB, respectively. See Section 2.3 for more information.

4.15. TBADDR (Translation Buffer Miss PTE Address)



The ITBADDR and DTBADDR control registers exist solely for the benefit of TB miss routines, and serve no other functional purpose. When a TB miss exception is taken, this control register is filled with the virtual address of the PTE to load from the virtually linear page table, saving a TB miss handler that implements this scheme from having to calculate this itself.

System software should write the upper 10 bits of this control register with the upper 10 bits of the virtual address of a virtually linear page table. As this scheme has no way to handle a page table base containing non-zero bits in the low 22 bits, the page table base should be naturally aligned to the size of the page table, i.e. 4MB-aligned.

When a TB miss exception occurs, the low 22 bits of this control register are filled by the processor with the index at which the relevant PTE can be found within the virtually linear page table. As the page table is a linear array, this index is trivial to calculate; it consists simply of the upper 20 bits of the missed virtual address.

The TB miss routine can load this control register into a general purpose register and use the contents as a virtual address with which to load the 32-bit PTE directly from the virtually linear page table. If the table page happens to be resident in the DTB already, this will succeed immediately. Otherwise, a nested DTB miss may be taken. See Section 2.3 for more details.

4.16. NMI Masking Events

A problem with non-maskable interrupts (NMIs) arises on RISC architectures. If an NMI is delivered while an exception handler is saving or restoring critical state, then this can be a fatal condition.

Therefore, in order to maximize the usefulness of NMIs, the XR/17032 architecture specifies several events which delay NMI delivery for at least 64 cycles. Each occurrence of one of these events resets an internal counter that decrements once per cycle, and NMIs can only be delivered while this counter is equivalent to zero.

The following events mask NMIs for a short period of at least 64 cycles:

1. An exception is taken.
2. The MTCR instruction is executed.
3. The MFCR instruction is executed.

5. Instructions

5.1. Instruction Set Summary

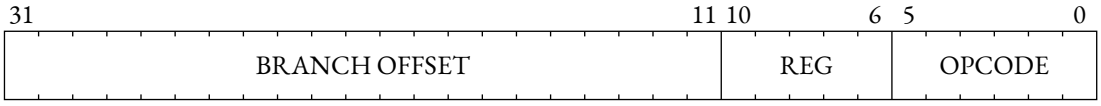
The XR/17032 architecture features only four instruction formats. All instruction formats are 32 bits wide, and there are a total of 60 instructions. This section contains a summary of all of the instructions defined by the XR/17032 architecture along with their encodings. The instructions are grouped first by format, and then by major opcode. The summary is followed by a much more comprehensive listing of the instructions and what they do.

Note that the assembly language also supports several "pseudo-instructions" for ease of assembly programming, which are not listed below, as they don't directly correspond to any particular hardware instruction, and are usually translated to a sequence of several hardware instructions. See Section 5.6 for a listing of pseudo-instructions.

Jump Format

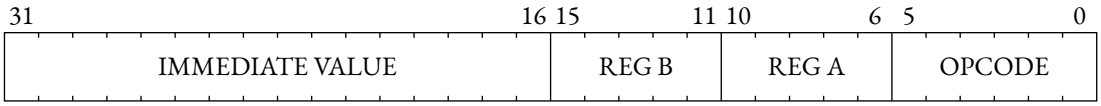
Opcode	Mnemonic	Name
0x6	J IMM29	Jump
0x7	JAL IMM29	Jump And Link

Branch Format



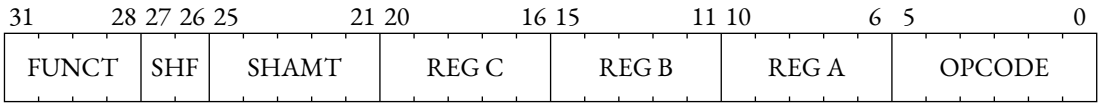
Opcode	Mnemonic	Name
0x3D	BEQ RA, IMM21	Branch Equal
0x35	BNE RA, IMM21	Branch Not Equal
0x2D	BLT RA, IMM21	Branch Less Than
0x25	BGT RA, IMM21	Branch Greater Than
0x1D	BLE RA, IMM21	Branch Less Than or Equal
0x15	BGE RA, IMM21	Branch Greater Than or Equal
0x0D	BPE RA, IMM21	Branch Parity Even
0x05	BPO RA, IMM21	Branch Parity Odd

Immediate Operate Format



Opcode	Mnemonic	Name
0x3C	ADDI RA, RB, IMM16	Add Immediate
0x34	SUBI RA, RB, IMM16	Subtract Immediate
0x2C	SLTI RA, RB, IMM16	Set Less Than Immediate
0x24	SLTI SIGNED RA, RB, IMM16	Set Less Than Immediate, Signed
0x1C	ANDI RA, RB, IMM16	And Immediate
0x14	XORI RA, RB, IMM16	Xor Immediate
0x0C	ORI RA, RB, IMM16	Or Immediate
0x04	LUI RA, RB, IMM16	Load Upper Immediate
0x3B	MOV RA, BYTE [RB + IMM16]	Load Byte, Immediate Offset
0x33	MOV RA, INT [RB + IMM16]	Load Int, Immediate Offset
0x2B	MOV RA, LONG [RB + IMM16]	Load Long, Immediate Offset
0x3A	MOV BYTE [RA + IMM16], RB	Store Byte, Immediate Offset
0x32	MOV INT [RA + IMM16], RB	Store Int, Immediate Offset
0x2A	MOV LONG [RA + IMM16], RB	Store Long, Immediate Offset
0x1A	MOV BYTE [RA + IMM16], IMM5	Store Byte, Small Immediate
0x12	MOV INT [RA + IMM16], IMM5	Store Int, Small Immediate
0x0A	MOV LONG [RA + IMM16], IMM5	Store Long, Small Immediate
0x38	JALR RA, RB, IMM16	Jump And Link, Register
0x30	ADR RA, IMM16	Compute Relative Address

Register Operate Format



Major Opcode 111001 (0x39)

Function	Mnemonic	Name
0xF	`MOV RA, BYTE [RB + RC xSH IMM5]`	Load Byte, Register Offset
0xE	`MOV RA, INT [RB + RC xSH IMM5]`	Load Int, Register Offset
0xD	`MOV RA, LONG [RB + RC xSH IMM5]`	Load Long, Register Offset
0xB	`MOV BYTE [RB + RC xSH IMM5], RA`	Store Byte, Register Offset
0xA	`MOV INT [RB + RC xSH IMM5], RA`	Store Int, Register Offset
0x9	`MOV LONG [RB + RC xSH IMM5], RA`	Store Long, Register Offset
0x8	`LSH RA, RC, RB`	Left Shift By Register
0x8	`RSH RA, RC, RB`	Logical Right Shift By Register
0x8	`ASH RA, RC, RB`	Arithmetic Right Shift By Register
0x8	`ROR RA, RC, RB`	Rotate Right By Register
0x7	`ADD RA, RB, RC xSH IMM5`	Add Register
0x6	`SUB RA, RB, RC xSH IMM5`	Subtract Register
0x5	`SLT RA, RB, RC xSH IMM5`	Set Less Than Register
0x4	`SLT SIGNED RA, RB, RC xSH IMM5`	Set Less Than Register, Signed
0x3	`AND RA, RB, RC xSH IMM5`	And Register
0x2	`XOR RA, RB, RC xSH IMM5`	Xor Register
0x1	`OR RA, RB, RC xSH IMM5`	Or Register
0x0	`NOR RA, RB, RC xSH IMM5`	Nor Register

Major Opcode 110001 (0x31)

Function	Mnemonic	Name
0xF	MUL RA, RB, RC	Multiply
0xD	DIV RA, RB, RC	Divide
0xC	DIV SIGNED RA, RB, RC	Divide, Signed
0xB	MOD RA, RB, RC	Modulo
0x9	LL RA, RB	Load Locked
0x8	SC RA, RB, RC	Store Conditional
0x6	PAUSE	Pause
0x3	MB	Memory Barrier
0x2	WMB	Write Memory Barrier
0x1	BRK	Breakpoint
0x0	SYS	System Service

Major Opcode 101001 (0x29, Privileged)

Function	Mnemonic	Name
0xF	MFCR RA, CR	Move From Control Register
0xE	MTCR CR, RA	Move To Control Register
0xC	HLT	Halt Until Next Interrupt
0xB	RFE	Return From Exception

5.2. Jump Listing

The format for the absolute jump instructions consists of a 3-bit opcode and a 29-bit jump target. The two possible opcodes for jump instructions are 111 and 110.

Note that this opcode field is unique; all other formats have a 6-bit opcode field. This small opcode is to allow the jump target to cover a 2GB range. This is accomplished by shifting the jump target left by 2, which produces a 31-bit address, and then taking the uppermost bit from that of the current program counter. This allows jumping anywhere within a 2GB userspace or kernel space in a single instruction.

Opcode	Mnemonic	Name
`111/0x07`	`JAL IMM29`	Jump And Link

```
Reg[31] = PC + 4
PC = (IMM29 << 2) | (PC & 0x80000000)
```

Description

The `JAL` instruction provides a lightweight means of calling a function. The next program counter (PC + 4) is saved in the link register (31) and then the PC is set to the target address.

Note that if the called function needs to call another function, it must be sure to save the link register first and then restore it.

Exceptions

None.

Opcode	Mnemonic	Name
`110/0x06`	`J IMM29`	Jump

```
PC = (IMM29 << 2) | (PC & 0x80000000)
```

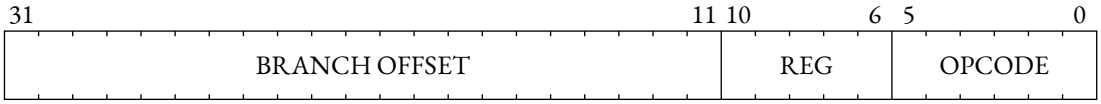
Description

The `J` instruction provides a way to do a long-distance absolute jump to another location, without destroying the contents of the link register.

Exceptions

None.

5.3. Branch Listing



The format for the branch instructions consists of a 6-bit opcode, a 5-bit register number, and a 21-bit branch offset. Every branch instruction has 101 as the low 3 bits of the opcode.

There is only one register field in order to maximize the size of the branch offset. This register is compared against zero in various ways. If the branch is taken, then the branch offset is shifted left by two, sign extended, and added to the current program counter. This gives a range of ±1M instructions, or ±4MB. As this covers the entire text section of most programs, and certainly covers any individual routine you're likely to find, this alleviates some burden that afflicts most RISC toolchains, as cross-procedure jumps will usually be done with absolute jumps anyway.

Opcode	Mnemonic	Name
`111101/0x3D`	`BEQ RA, IMM21`	Branch Equal
`110101/0x35`	`BNE RA, IMM21`	Branch Not Equal
`101101/0x2D`	`BLT RA, IMM21`	Branch Less Than
`100101/0x25`	`BGT RA, IMM21`	Branch Greater Than
`011101/0x1D`	`BLE RA, IMM21`	Branch Less Than Or Equal
`010101/0x15`	`BGE RA, IMM21`	Branch Greater Than Or Equal

```
IF Reg[RA] COND 0 THEN
  PC += SignExtend(IMM21) << 2
END
```

Description

The conditional branch instructions perform a relative jump if the given signed comparison between the contents of Register A and zero evaluates to true.

Exceptions

None.

Opcode	Mnemonic	Name
`001101/0x0D`	`BPE RA, IMM21`	Branch Parity Even
`000101/0x05`	`BPO RA, IMM21`	Branch Parity Odd

```
IF Reg[RA] & 0x1 == ? THEN
  PC += SignExtend(IMM21) << 2
END
```

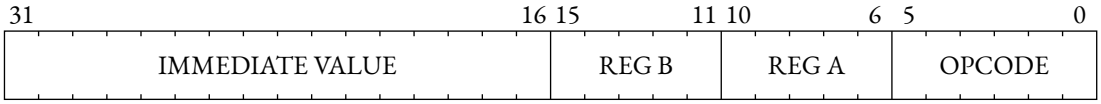
Description

The branch on parity instructions perform a relative jump if Register A has the given parity. That is, for BPE, a test is made whether the low bit is clear. For BPO, a test is made whether the low bit is set.

Exceptions

None.

5.4. Immediate Operate Listing



The format for the immediate operate instructions consists of a 6-bit opcode, two 5-bit register numbers, and a 16-bit immediate value. Every immediate operate instruction has either 100, 011, 010, or 000 as the low 3 bits of the opcode.

Note that the immediate value may or may not be sign extended, depending on the instruction.

Opcode	Mnemonic	Name
`111100/0x3C`	`ADDI RA, RB, IMM16`	Add Immediate
`110100/0x34`	`SUBI RA, RB, IMM16`	Subtract Immediate
`101100/0x2C`	`SLTI RA, RB, IMM16`	Set Less Than Immediate
`100100/0x24`	`SLTI SIGNED RA, RB, IMM16`	Set Less Than Immediate, Signed
`011100/0x1C`	`ANDI RA, RB, IMM16`	And Immediate
`010100/0x14`	`XORI RA, RB, IMM16`	Xor Immediate
`001100/0x0C`	`ORI RA, RB, IMM16`	Or Immediate
`000100/0x04`	`LUI RA, RB, IMM16`	Load Upper Immediate

```
Reg[RA] = Reg[RB] OP IMM16
```

Description

The immediate operate instructions perform the given operation between the contents of Register B and a 16-bit immediate value, which is zero-extended, except for `SLTI SIGNED` where it is sign-extended. The result is stored in Register A.

In the case of the comparison instructions, a 1 is stored if the comparison is true, and a 0 otherwise.

In the case of LUI, the operation performed is a bitwise OR. However, the immediate is first shifted 16 bits to the left, enabling the loading of large constants whose low 16 bits are clear. LUI also enables the synthesis of pseudo-instructions for loading arbitrary large constants.

Exceptions

None.

Opcode	Mnemonic	Name
`111011/0x3B`	`MOV RA, BYTE [RB + IMM16]`	Load Byte, Immediate Offset
`110011/0x33`	`MOV RA, INT [RB + IMM16]`	Load Int, Immediate Offset
`101011/0x2B`	`MOV RA, LONG [RB + IMM16]`	Load Long, Immediate Offset

```
Reg[RA] = Load(Reg[RB] + (IMM16 * width), width)
```

Description

The load instructions read a value into Register A from the address stored within Register B plus a zero-extended 16-bit immediate offset. To extend the range, the processor multiplies the immediate offset by the width of the access (in bytes) before it is added to the base register. The address must be naturally aligned to the width.

Exceptions

- DTB (DTB miss) if paging is enabled and the referenced page mapping is not in the DTB.
- PGF (Read Page Fault) if paging is enabled and the referenced page matches a DTB entry with a clear valid bit.
- BUS (Bus Error) if the memory access causes a timeout on the system bus.
- UNA (Unaligned Access) if the referenced address is not naturally aligned.

Opcode	Mnemonic	Name
`111010/0x3A`	`MOV BYTE [RA + IMM16], RB`	Store Byte, Immediate Offset
`110010/0x32`	`MOV INT [RA + IMM16], RB`	Store Int, Immediate Offset
`101010/0x2A`	`MOV LONG [RA + IMM16], RB`	Store Long, Immediate Offset

```
Store(Reg[RA] + (IMM16 * width), Reg[RB], width)
```

Description

The store register instructions store the value contained with Register B to the address stored within Register A plus a zero-extended 16-bit immediate offset. To extend the range, the processor multiplies the immediate offset by the width of the access (in bytes) before it is added to the base register. The address must be naturally aligned to the width.

Exceptions

- DTB (DTB miss) if paging is enabled and the referenced page mapping is not in the DTB.
- PGW (Write Page Fault) if paging is enabled and the referenced page matches a DTB entry with a clear valid bit, or a DTB entry with a clear writable bit.
- BUS (Bus Error) if the memory access causes a timeout on the system bus.
- UNA (Unaligned Access) if the referenced address is not naturally aligned.

Opcode	Mnemonic	Name
`011010/0x1A`	`MOV BYTE [RA + IMM16], IMM5`	Store Byte, Small Immediate
`010010/0x12`	`MOV INT [RA + IMM16], IMM5`	Store Int, Small Immediate
`001010/0x0A`	`MOV LONG [RA + IMM16], IMM5`	Store Long, Small Immediate

```
Store(Reg[RA] + (IMM16 * width), SignExt5(IMM5), width)
```

Description

The store small immediate instructions store a sign-extended 5-bit immediate to the address stored within Register A plus a zero-extended 16-bit immediate offset. To extend the range, the processor multiplies the immediate offset by the width of the access (in bytes) before it is added to the base register. The address must be naturally aligned to the width. The 5-bit immediate is encoded in the Register B field of the instruction word.

Exceptions

- DTB (DTB miss) if paging is enabled and the referenced page mapping is not in the DTB.
- PGW (Write Page Fault) if paging is enabled and the referenced page matches a DTB entry with a clear valid bit, or a DTB entry with a clear writable bit.
- BUS (Bus Error) if the memory access causes a timeout on the system bus.
- UNA (Unaligned Access) if the referenced address is not naturally aligned.

Opcode	Mnemonic	Name
`111000/0x38`	`JALR RA, RB, IMM16`	Jump And Link, Register

```
Reg[RA] = PC + 4
PC = Reg[RB] + (IMM16 << 2)
```

Description

The JALR instruction provides a lightweight means of calling through a function pointer. The next program counter (PC + 4) is saved in Register A, and then the PC is set to the contents of Register B plus a 16-bit zero-extended immediate value shifted left by two.

This instruction can also be used as an indirect jump to an address stored in a register, by setting the destination register to the zero register, thereby discarding the return value.

Exceptions

None.

Opcode	Mnemonic	Name
`110000/0x30`	`ADR RA, IMM16`	Compute Relative Address

```
Reg[RA] = PC + (IMM16 << 16)
```

Description

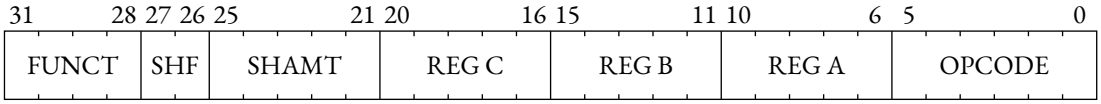
The ADR instruction adds the current program counter and a 16-bit immediate value shifted left by 16. The result is stored in Register A. This instruction is useful for PC-relative addressing modes.

Unlike other immediate operate format instructions, this instruction's Register B field must be zero.

Exceptions

None.

5.5. Register Operate Listing



The format for the register operate instructions consists of a 6-bit opcode, three 5-bit register numbers, a 5-bit shift amount, a 2-bit shift type, and a 4-bit function code (which acts as an extended opcode). Every register operate instruction has 001 as the low 3 bits of the opcode, and there are three such opcodes; 111001, 110001, and 101001.

All privileged instructions are in this format and are function codes of the last opcode mentioned, 101001. These instructions will produce a privilege violation exception if executed while usermode is enabled in the RS control register (see Section 4.2).

The value of Register C is shifted in the manner specified by the shift type, by the amount specified by the shift amount.

00	**LSH** Left shift.
01	**RSH** Logical right shift.
10	**ASH** Arithmetic right shift.
11	**ROR** Rotate right.

The 2-bit shift type codes (SHF field).

5.5.1. Register Operate, Opcode 111001

Function	Mnemonic	Name
`1111/0xF`	`MOV RA, BYTE [RB + RC xSH IMM5]`	Load Byte, Register Offset
`1110/0xE`	`MOV RA, INT [RB + RC xSH IMM5]`	Load Int, Register Offset
`1101/0xD`	`MOV RA, LONG [RB + RC xSH IMM5]`	Load Long, Register Offset

```
Reg[RA] = Load(Reg[RB] + (Reg[RC] xSH IMM5), width)
```

Description

These instructions perform a load with an offset contained within a register. A zero-extended value is loaded into Register A from the address computed by adding the value of Register C, shifted in the manner specified, to the value of Register B.

Exceptions

- DTB (DTB miss) if paging is enabled and the referenced page mapping is not in the DTB.
- PGF (Read Page Fault) if paging is enabled and the referenced page matches a DTB entry with a clear valid bit.
- BUS (Bus Error) if the memory access causes a timeout on the system bus.
- UNA (Unaligned Access) if the referenced address is not naturally aligned.

Function	Mnemonic	Name
1011/0xB	MOV BYTE [RB + RC xSH IMM5], RA	Store Byte, Register Offset
1010/0xA	MOV INT [RB + RC xSH IMM5], RA	Store Int, Register Offset
1001/0x9	MOV LONG [RB + RC xSH IMM5], RA	Store Long, Register Offset

```
Store(Reg[RB] + (Reg[RC] xSH IMM5), Reg[RA], width)
```

Description

These instructions perform a store with an offset contained within a register. The value in Register A is stored to the address computed by adding the value of Register C, shifted in the manner specified, to the value of Register B.

Exceptions

- DTB (DTB miss) if paging is enabled and the referenced page mapping is not in the DTB.
- PGW (Write Page Fault) if paging is enabled and the referenced page matches a DTB entry with a clear valid bit, or a DTB entry with a clear writable bit.
- BUS (Bus Error) if the memory access causes a timeout on the system bus.
- UNA (Unaligned Access) if the referenced address is not naturally aligned.

Function	Mnemonic	Name
1000 (0x8)	`LSH/RSH/ASH/ROR RA, RC, RB`	Various Shift By Register

```
Reg[RA] = Reg[RC] xSH (Reg[RB] & 31)
```

Description

This instruction shifts the contents of Register C by the contents of Register B and places the result in Register A. It is technically a single function code, but is split into several mnemonics for convenience. The IMM5 shift value encoded in the instruction is ignored, because the shift value is taken from Register B instead.

Exceptions

None.

Function	Mnemonic	Name
0111/0x7	`ADD RA, RB, RC xSH IMM5`	Add Register
0110/0x6	`SUB RA, RB, RC xSH IMM5`	Subtract Register
0101/0x5	`SLT RA, RB, RC xSH IMM5`	Set Less Than Register
0100/0x4	`SLT SIGNED RA, RB, RC xSH IMM5`	Set Less Than Register, Signed
0011/0x3	`AND RA, RB, RC xSH IMM5`	And Register
0010/0x2	`XOR RA, RB, RC xSH IMM5`	Xor Register
0001/0x1	`OR RA, RB, RC xSH IMM5`	Or Register
0000/0x0	`NOR RA, RB, RC xSH IMM5`	Nor Register

```
Reg[RA] = Reg[RB] OP (Reg[RC] xSH IMM5)
```

Description

These instructions perform the specified operation between the contents of Register B and the contents of Register C, and stores the result into Register A. The contents of Register C are first shifted in the manner specified.

In the case of the comparison instructions, a 1 is stored if the comparison is true, and a 0 otherwise.

Exceptions

None.

5.5.2. Register Operate, Opcode 110001

Function	Mnemonic	Name
`1111/0xF`	`MUL RA, RB, RC xSH IMM5`	Multiply
`1101/0xD`	`DIV RA, RB, RC xSH IMM5`	Divide
`1100/0xC`	`DIV SIGNED RA, RB, RC xSH IMM5`	Divide, Signed
`1011/0xB`	`MOD RA, RB, RC xSH IMM5`	Modulo

```
Reg[RA] = Reg[RB] OP (Reg[RC] xSH IMM5)
```

Description

These instructions perform the specified operation between the contents of Register B and the contents of Register C, and stores the result into Register A. The contents of Register C are first shifted in the manner specified.

The result of division is rounded toward zero, to a whole integer.

Exceptions

None.

Function	Mnemonic	Name
1001/0x9	LL RA, RB	Load Locked

```
Locked = TRUE
LockedAddress = Translate(Reg[RB])
Reg[RA] = Load32(Reg[RB])
```

Description

This instruction is used to implement atomic sequences. It loads the 32-bit contents of a naturally-aligned memory address contained within Register B into Register A. The "locked" flag is set to TRUE, and the "locked address" is set to the physical address being accessed.

If an RFE (Return From Exception) instruction is executed, the "locked" flag is cleared, causing a future SC instruction on the same processor to fail. This can be used to implement atomic sequences in non-privileged code.

In a multiprocessor system, if any other processor performs a store to this processor's "locked address", this processor's "locked" flag is cleared.

Exceptions

- DTB (DTB miss) if paging is enabled and the referenced page mapping is not in the DTB.
- PGF (Read Page Fault) if paging is enabled and the referenced page matches a DTB entry with a clear valid bit.
- BUS (Bus Error) if the memory access causes a timeout on the system bus.
- UNA (Unaligned Access) if the referenced address is not naturally aligned.

Function	Mnemonic	Name
1000/0x8	SC RA, RB, RC	Store Conditional

```
IF Locked THEN
  Store32(Reg[RB], Reg[RC])
END
Reg[RA] = Locked
```

Description

This instruction stores the current value of the processor's "locked" flag to Register A. If the "locked" flag is set, it stores the contents of Register C to the address contained within Register B.

Exceptions

- DTB (DTB miss) if paging is enabled and the referenced page mapping is not in the DTB.
- PGW (Write Page Fault) if paging is enabled and the referenced page matches a DTB entry with a clear valid bit, or a DTB entry with a clear writable bit.
- BUS (Bus Error) if the memory access causes a timeout on the system bus.
- UNA (Unaligned Access) if the referenced address is not naturally aligned.

Function	Mnemonic	Name
0110/0x6	PAUSE	Pause

```
// Possible implementation.
IF PauseCount++ & 255 == 0 THEN
  Yield()
END
```

Description

On multiprocessor systems, this instruction should be executed on each iteration of a spin-wait for another processor to do something (e.g. release a spinlock, respond to IPI). It is a hint that the processor isn't doing useful work.

Exceptions

None.

Function	Mnemonic	Name
0011/0x3	MB	Memory Barrier

```
// Possible implementation.
FlushWriteBuffer()
SynchronizeLoads()
```

Description

This instruction ensures that, from the perspective of all processors and I/O devices in the system, no reads or writes performed by this processor are reordered across the MB instruction in either direction. This is sometimes necessary for proper multiprocessor memory ordering.

Exceptions

None.

Function	Mnemonic	Name
0010/0x2	WMB	Write Memory Barrier

```
// Possible implementation.
FlushWriteBuffer()
```

Description

This instruction ensures that, from the perspective of all processors and I/O devices in the system, no writes performed by this processor before the WMB instruction are reordered after the WMB instruction. One example of this instruction on a uniprocessor system is to ensure that a device has seen a sequence of writes to its registers before asking it to perform a command. An example on a multiprocessor system is to ensure data coherency before releasing a spinlock.

Exceptions

None.

Function	Mnemonic	Name
0001/0x1	BRK	Breakpoint

```
Exception(BRK)
```

Description

This instruction causes a breakpoint exception. Its intended use is for debugging purposes. See Section 4.3 for more information on exceptions.

Exceptions

- BRK (Breakpoint) is always triggered by this instruction.

Function	Mnemonic	Name
0000/0x0	SYS	System Service

```
Exception(SYS)
```

Description

This instruction causes a system service exception. It is useful for usermode to make a call into the system software to request a service (also called a system service or “syscall”). See Section 4.3 for more information on exceptions.

Exceptions

- SYS (System Service) is always triggered by this instruction.

5.5.3. Register Operate, Opcode 101001 (Privileged)

Function	Mnemonic	Name
1111/0xF	MFCR RA, CR	Move From Control Register

```
Reg[RA] = ControlReg[CR]
```

Description

This instruction moves the contents of the specified control register into Register A. The 5-bit control register number is encoded in the place of Register C.

Note that not all control registers are simple memory-like repositories of bits, and may perform special actions when read. See Section 4.1 for a full listing of control registers and their behaviors.

Exceptions

- PRV (Privilege Violation) if executed while usermode is active.

Function	Mnemonic	Name
1110/0xE	MTCR CR, RB	Move To Control Register

```
ControlReg[CR] = Reg[RB]
```

Description

This instruction moves the contents of Register B into the specified control register. The 5-bit control register number is encoded in the place of Register C. Register A is ignored but should be encoded as zero.

Note that not all control registers are simple memory-like repositories of bits, and may perform special actions when written. See Section 4.1 for a full listing of control registers and their behaviors.

Exceptions

- PRV (Privilege Violation) if executed while usermode is active.

Function	Mnemonic	Name
1100/0xC	HLT	Halt Until Next Interrupt

```
HaltUntilInterrupt()
```

Description

This instruction pauses execution of the processor until the next external interrupt is received. This can be used as a power-saving measure; for instance, executing HLT in a loop in the low priority idle thread of a multitasking kernel could greatly reduce the idle power consumption of the system. If external interrupts are disabled, this instruction causes the processor to halt forever, that is, until it is physically reset.

Exceptions

- PRV (Privilege Violation) if executed while usermode is active.

Function	Mnemonic	Name
1011/0xB	RFE	Return From Exception

```
Locked = FALSE
IF ControlReg[RS].ModeStack & T THEN
  PC = ControlReg[TBPC]
ELSE
  PC = ControlReg[EPC]
END
ControlReg[RS].ModeStack = ControlReg[RS].ModeStack >> 8
```

Description

This instruction “pops” the “mode stack” of the RS control register (see Section 4.2), and returns execution to the program counter saved in either the TBPC or EPC control register, depending on if the T bit of RS was set or not, respectively (i.e., whether a TB miss handler was active or not; see Section 2.3). It also clears the “locked” flag, causing the next SC (Store Conditional) instruction to fail.

Exceptions

- PRV (Privilege Violation) if executed while usermode is active.

5.6. Pseudo-Instruction Listing

Some operations are synthesized out of simpler instructions, but are common or inconvenient enough to warrant a "pseudo-instruction", a fake instruction that the assembler converts into a corresponding hardware instruction sequence. The following is a (not necessarily exhaustive, depending on the assembler) list of common pseudo-instructions.

Mnemonic	Name
`B IMM21`	Unconditional Relative Branch

```
BEQ ZERO, IMM21
```

Description

This pseudo-instruction performs an unconditional relative branch. This is synthesized from the BEQ (Branch Equal) instruction, used here to compare the contents of the register ZERO with the number zero; by definition, this will always be true, thereby synthesizing an unconditional branch.

Exceptions

None.

Mnemonic	Name
`RET`	Return

```
JALR ZERO, LR, 0
```

Description

This pseudo-instruction performs a common return-from-subroutine operation. This is synthesized from the JALR (Jump And Link, Register) instruction, by performing a jump-and-link to the contents of the link register LR, and saving the result in ZERO (thereby discarding it).

Exceptions

None.

Mnemonic	Name
`JR RA`	Jump to Register

```
JALR ZERO, RA, 0
```

Description

This pseudo-instruction performs a jump to the contents of Register A. This is synthesized from the JALR (Jump And Link, Register) instruction, by performing a jump-and-link to the contents of Register A, and saving the result in ZERO (thereby discarding it).

Exceptions

None.

Mnemonic	Name
`MOV RA, RB`	Move Register

```
ADD RA, RB, ZERO LSH 0
```

Description

This pseudo-instruction copies the contents of Register B into Register A. It is synthesized from the ADD (Add Register) instruction, by adding the contents of the ZERO register to the contents of Register B (which is a no-op), and saving the results in Register A.

Exceptions

None.

Mnemonic	Name
`LI RA, IMM16`	Load 16-bit Immediate

```
ADDI RA, ZERO, IMM16
```

Description

This pseudo-instruction loads a 16-bit immediate into Register A. It is synthesized from the ADDI (Add Immediate) instruction, by adding the immediate to the contents of the ZERO register and saving the results in Register A.

Exceptions

None.

Mnemonic	Name
`LA RA, IMM32`	Load 32-bit Immediate

```
LUI RA, ZERO, (IMM32 >> 16)
ORI RA, RA, (IMM32 & 0xFFFF)
```

Description

This pseudo-instruction loads a 32-bit immediate into Register A. It is synthesized from the LUI (Load Upper Immediate) and ORI (Or Immediate) instructions, by loading the upper 16 bits of the immediate into the register with LUI, and then bitwise OR-ing the lower 16 bits in with ORI.

Exceptions

None.

Mnemonic	Name
`NOP`	No Operation

```
ADDI ZERO, ZERO, 0
```

Description

This pseudo-instruction does nothing, by adding the contents of the ZERO register with the number zero and saving the result in the ZERO register.

Note that the instruction of all zeroes is *not* a no-op. The instruction set was designed to ensure that the instruction of all zeroes is an invalid instruction, so that an invalid instruction exception will tend to occur more immediately if a mistaken jump occurs.

Exceptions

None.

Mnemonic	Name
`LSHI/RSHI/ASHI/RORI RA, RB, IMM5`	Various Shift By Immediate

```
ADD RA, ZERO, RB xSH IMM5
```

Description

These pseudo-instructions shift the contents of Register B by the 5-bit immediate, and save the result in Register A. They are synthesized from the ADD (Add Register) instruction, by adding the contents of Register B with the contents of the ZERO register, and shifting it in the specified manner.

Exceptions

None.

Mnemonic	Name
`MOV RA, BYTE/INT/LONG [IMM32]`	Load from 32-bit Address

```
LUI RA, ZERO, (IMM32 >> 16)
MOV RA, BYTE/INT/LONG [RA + (IMM32 & 0xFFFF)]
```

Description

These pseudo-instructions load a value into Register A from a full 32-bit address. They are synthesized from LUI (Load Upper Immediate) and the appropriate offsetted load instruction. The upper 16 bits of the address are loaded into the register with LUI, and then a load is done into the register, with an offset of the low 16 bits of the address.

Exceptions

Any exception that can be caused by the load instruction.

Mnemonic	Name
`MOV BYTE/INT/LONG [IMM32], RA, TMP=RB`	Store to 32-bit Address

```
LUI RB, ZERO, (IMM32 >> 16)
MOV BYTE/INT/LONG [RB + (IMM32 & 0xFFFF)], RA
```

Description

These pseudo-instructions store a value into a full 32-bit address. They are synthesized with LUI (Load Upper Immediate) and the appropriate offsetted store instruction. The upper 16 bits of the address are loaded into a user-supplied temporary register with LUI, and then a store is done with an offset of the low 16 bits of the address.

Exceptions

Any exception that can be caused by the store instruction.