architecture handbook

XR/station Project
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.

1.2. Starting at the beginning…

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.

1.3. Registers

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, zero, almost always reads as zero and ignores writes. This is a common RISC design tactic that simplifies the encoding of many instructions. A table of GPRs follows:

#	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.

1.4. Control Registers

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**. A table containing a summary of all defined control registers follows:

#	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 **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 **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.

Any absent CR numbers have undefined behavior if read or written.

See Section 4 for a more detailed description of each control register.

1.5. 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 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 2^{32} = 4GB virtual address space. For simplicity, the only supported page size on the XR/17032 processor is 4096 bytes, or 4KB. This means that the virtual address space is evenly tiled by 4GB / 4KB = 1048576 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 1048576 * 4 bytes = 4MB of memory (per process!), which is obviously unacceptable overhead.

In many architectures, such as fox32 and Intel 386, the virtual address space is therefore managed by a two-level page table. The indices into the two levels of the page table are usually extracted from bit fields of the 32-bit virtual address in the manner shown:

The two 10-bit fields from 22:31 and from 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 $2^{10} * 4 = 4096$ 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.

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 the aforementioned fox32 and Intel 386, 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.

This, still, has two major problems:

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

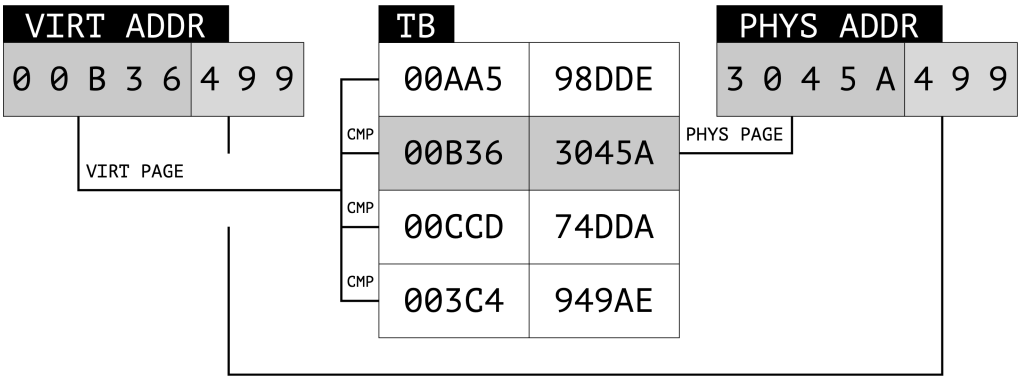
[1] Note that this is one reason for the usage of the 4KB page size: this is the page size for which the division of the virtual address into these three fields causes the tables to consume single page frames, which simplifies memory management.

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

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).



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 4.1.2), 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 4.1.11 for the format of the **TBTAG**, and Section 4.1.10 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 Intel 386: 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.

2.2.1. Address Space IDs

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 4.1.13 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 Section 4.1.4.1 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.

[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.

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.

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.

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.

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:

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

[4] As an example of the usage of wired entries, system software will typically map the exception block (see Section 4.1.4) 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.

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.

Pseudo-code for calculating the base address of the virtually linear page table is provided:

```
// 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
```

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:

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

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.

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.

3.2. Managing the Caches

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 4.1.14.

3.3. The Caches and XR/computer Systems

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

3.3. The Caches and XR/computer Systems

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”).

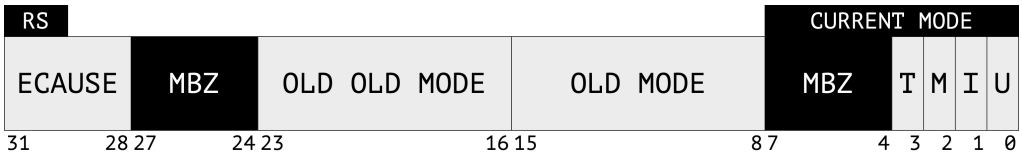
4. Processor Control

4.1.1. Introduction

The behavior of the processor is primarily controlled by a small set of control registers (CRs). They are summarized by a table in Section 1.4, and are explored in much more detail below.

4.1.2. 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. They are enumerated in Section 4.1.4.1.

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], this is accomplished by pushing the old state onto the stack.

However, because this is a design following RISCy philosophy, performing a memory access during exception dispatch is considered unacceptable complexity. Instead, 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, and so manually saving **RS** is required if it is desired to go more than three levels deep. 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.

[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.

4.1.2. RS

The mode bits are defined as follows when set:

	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. This causes the **zero** register to be usable as a full register; i.e., it is not wired to read all zeroes while this bit is set. This is intended to free it as a scratch register for TB miss routines, to avoid having to save any registers. This bit also has special effects on exception handling, which are enumerated in Section 2.3.

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.

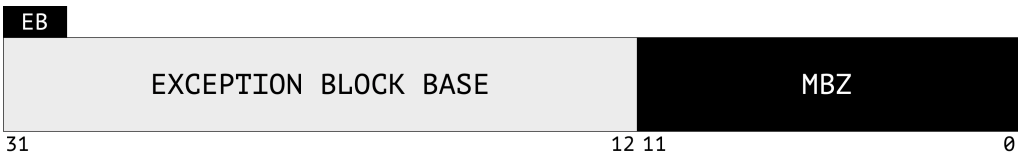
4.1.3. WHAMI

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.

4.1.4. 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 $\frac{4096}{16} = 256$ bytes, or $\frac{256}{4} = 64$ 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 4.1.12 or Section 2.2 for more details on wired TB entries.

4.1.4.1. ECAUSE Codes

A table of all defined exception causes follows:

#	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.

Any absent ECAUSE codes are reserved for future use.

4.1.5. 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 4.1.2).

4.1.6. 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.

4.1.7. 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.

4.1.8. 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.

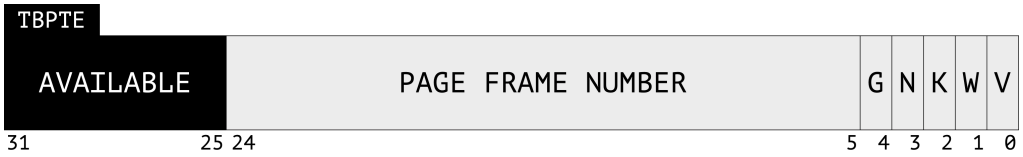
4.1.9. SCRATCH0-4

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.

4.1.10. ITBPTE/DTBPTE

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.

The PTE bits are defined as follows when set:

	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**.

4.1.11. ITBTAG/DTBTAG

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 4.1.10 and Section 2.3).

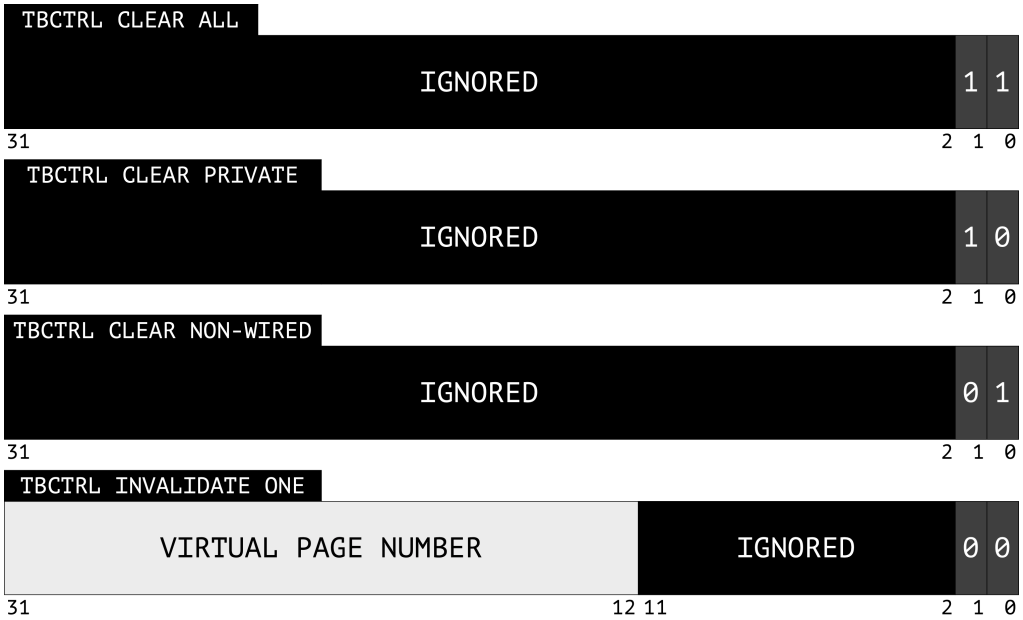
4.1.12. ITBINDEX/DTBINDEX

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.

4.1.13. ITBCTRL/DTBCTRL

Translation Buffer Control



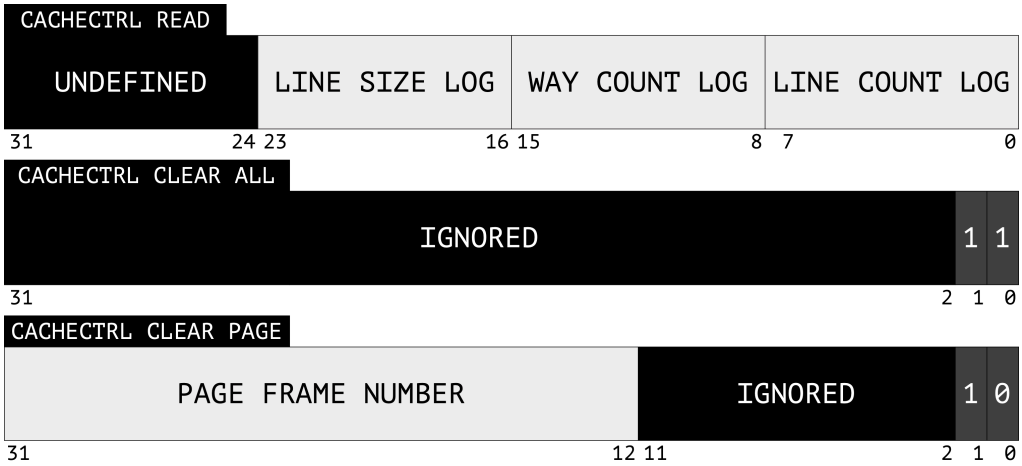
Writes to **ITBCTRL** and **DTBCTRL** can be used to invalidate entries in the ITB or DTB, respectively. The 32-bit value written to the control register should be in one of the three formats enumerated above, distinguished by the low two bits. Any other combination of low bits will yield unpredictable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

	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.

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.

4.1.14. ICACHECTRL/DCACHECTRL

Cache Control



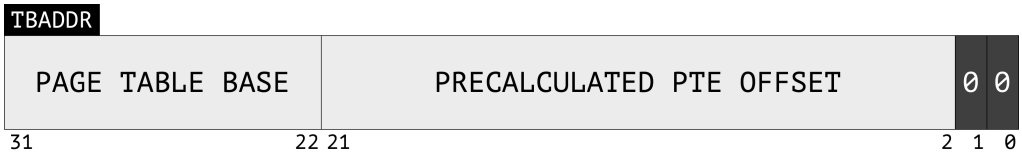
Reads from the **ICACHECTRL** and **DCACHECTRL** control registers yield a 32-bit value whose bit fields indicate the parameters of the Icache and Dcache respectively; the number of lines in the cache, the number of ways (i.e. the set associativity) of the cache, and the size of a cache line are each given, in the form of a binary logarithm. I.e., if the line count field contains 8, then there are $2^8 = 256$ 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. The action of each format is as follows:

	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.

4.1.15. ITBADDR/DTBADDR

Translation Buffer Miss, Page Table Entry 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. 2^{22} = 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.

4.2. 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.

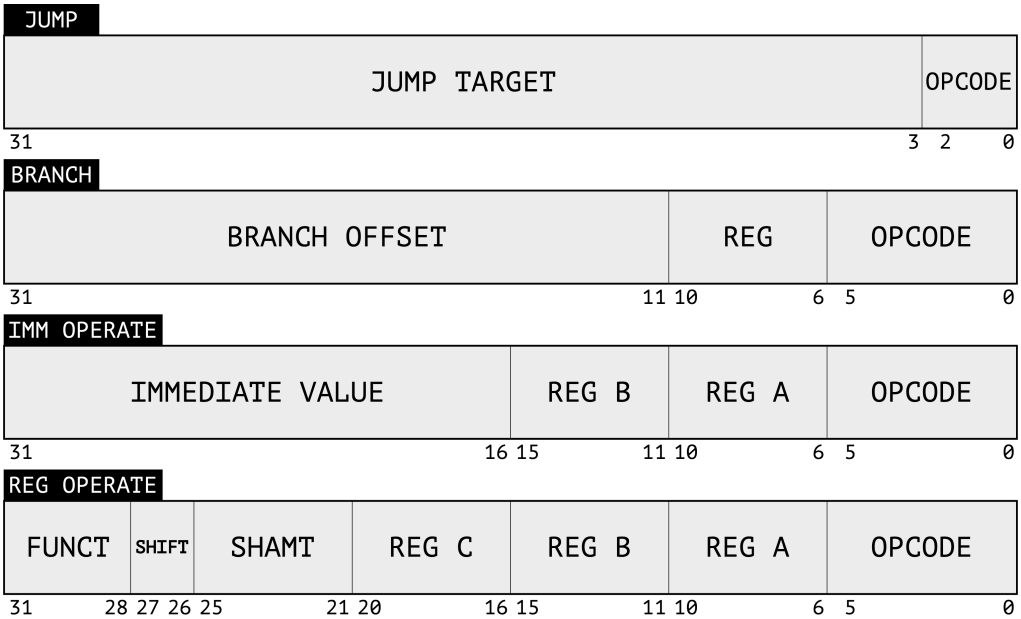
The following events mask NMIs for a short period of at least 64 cycles:

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

5. Instructions

5.1. Instruction Formats Summary

The XR/17032 architecture features only four instruction formats, and each are 32 bits wide. There are a total of 60 instructions, which are summarized below. A more comprehensive description of each format and instruction can be found in Section 5.2.



5.1.1. Jump Instructions Summary

- **J IMM29** *Jump*
- **JAL IMM29** *Jump And Link*

5.1.2. Branch Instructions Summary

- **BEQ RA, IMM21** *Branch Equal*
- **BNE RA, IMM21** *Branch Not Equal*
- **BLT RA, IMM21** *Branch Less Than*
- **BGT RA, IMM21** *Branch Greater Than*
- **BGE RA, IMM21** *Branch Greater Than or Equal*
- **BLE RA, IMM21** *Branch Less Than or Equal*
- **BPE RA, IMM21** *Branch Parity Even*
- **BPO RA, IMM21** *Branch Parity Odd*

5.1.3. Immediate Operate Instructions Summary

- **ADDI RA, RB, IMM16** *Add Immediate*
- **SUBI RA, RB, IMM16** *Subtract Immediate*
- **SLTI RA, RB, IMM16** *Set Less Than Immediate*
- **SLTI SIGNED RA, RB, IMM16** *Set Less Than Immediate, Signed*
- **ANDI RA, RB, IMM16** *And Immediate*
- **XORI RA, RB, IMM16** *Xor Immediate*
- **ORI RA, RB, IMM16** *Or Immediate*

5.1.3. Immediate Operate Instructions Summary

- **LUI RA, RB, IMM16** *Load Upper Immediate*
- **MOV RA, BYTE [RB + IMM16]** *Load Byte, Immediate Offset*
- **MOV RA, INT [RB + IMM16]** *Load Int, Immediate Offset*
- **MOV RA, LONG [RB + IMM16]** *Load Long, Immediate Offset*
- **MOV BYTE [RA + IMM16], RB** *Store Byte, Immediate Offset*
- **MOV INT [RA + IMM16], RB** *Store Int, Immediate Offset*
- **MOV LONG [RA + IMM16], RB** *Store Long, Immediate Offset*
- **MOV BYTE [RA + IMM16], IMM5** *Store Byte, Small Immediate*
- **MOV INT [RA + IMM16], IMM5** *Store Int, Small Immediate*
- **MOV LONG [RA + IMM16], IMM5** *Store Long, Small Immediate*
- **JALR RA, RB, IMM16** *Jump And Link, Register*

5.1.4. Register Operate Instructions Summary

Major Opcode 111001

- **MOV RA, BYTE [RB + RC xSH IMM5]** *Load Byte, Register Offset*
- **MOV RA, INT [RB + RC xSH IMM5]** *Load Int, Register Offset*
- **MOV RA, LONG [RB + RC xSH IMM5]** *Load Long, Register Offset*
- **MOV BYTE [RB + RC xSH IMM5], RA** *Store Byte, Register Offset*
- **MOV INT [RB + RC xSH IMM5], RA** *Store Int, Register Offset*
- **MOV LONG [RB + RC xSH IMM5], RA** *Store Long, Register Offset*
- **LSH RA, RB, RC** *Left Shift By Register Amount*
- **RSH RA, RB, RC** *Logical Right Shift By Register Amount*
- **ASH RA, RB, RC** *Arithmetic Right Shift By Register Amount*
- **ROR RA, RB, RC** *Rotate Right By Register Amount*
- **ADD RA, RB, RC xSH IMM5** *Add Register*
- **SUB RA, RB, RC xSH IMM5** *Subtract Register*
- **SLT RA, RB, RC xSH IMM5** *Set Less Than Register*
- **SLT SIGNED RA, RB, RC xSH IMM5** *Set Less Than Register, Signed*
- **AND RA, RB, RC xSH IMM5** *And Register*
- **XOR RA, RB, RC xSH IMM5** *Xor Register*
- **OR RA, RB, RC xSH IMM5** *Or Register*
- **NOR RA, RB, RC xSH IMM5** *Nor Register*

Major Opcode 110001

- **MUL RA, RB, RC** *Multiply*
- **DIV RA, RB, RC** *Divide*
- **DIV SIGNED RA, RB, RC** *Divide, Signed*
- **MOD RA, RB, RC** *Modulo*
- **LL RA, RB** *Load Locked*
- **SC RA, RB, RC** *Store Conditional*
- **MB** *Memory Barrier*
- **WMB** *Write Memory Barrier*
- **BRK** *Breakpoint*
- **SYS** *System Service*

Major Opcode 101001 (Privileged Instructions)

- **MFCR RA, CR** *Move From Control Register*
- **MTCR CR, RA** *Move To Control Register*
- **HLT** *Halt Until Next Interrupt*
- **RFE** *Return From Exception*

5.2. Instruction Listing

The following section contains a comprehensive listing 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.

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 5.3 for a listing of pseudo-instructions.

5.2.1. Jump Format

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.

JAL IMM29

Jump And Link

Opcode: **111** (0x07)

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

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.

J IMM29

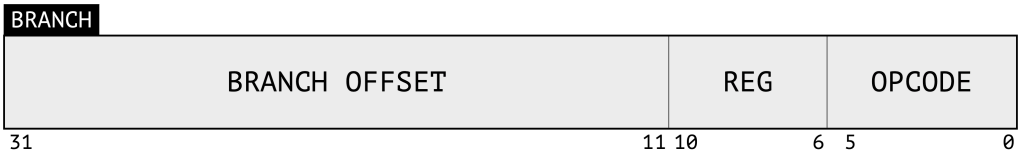
Jump

Opcode: **110** (0x06)

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

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

5.2.2. Branch Format



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.

BEQ RA, IMM21

Branch Equal

Opcode: **111101** (0x3D)

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

The **BEQ** instruction performs a relative jump if the contents of **Register A** are equal to zero.

BNE RA, IMM21

Branch Not Equal

Opcode: **110101** (0x35)

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

The **BNE** instruction performs a relative jump if the contents of **Register A** are not equal to zero.

BLT RA, IMM21

Branch Less Than

Opcode: **101101** (0x2D)

```
IF Reg[RA] & 0x80000000 THEN
  PC += SignExtend(IMM21)
END
```

The **BLT** instruction performs a relative jump if the contents of **Register A** are less than zero, i.e., the sign bit is set.

BGT RA, IMM21

Branch Greater Than

Opcode: **100101** (0x25)

```
IF NOT (Reg[RA] & 0x80000000) AND Reg[RA] ≠ 0 THEN
  PC += SignExtend(IMM21)
END
```

The **BGT** instruction performs a relative jump if the contents of **Register A** are greater than zero, i.e., the sign bit is clear and the register is not equal to zero.

BLE RA, IMM21

Branch Less Than Or Equal

Opcode: **011101** (0x1D)

```
IF Reg[RA] & 0x80000000 OR Reg[RA] == 0 THEN
  PC += SignExtend(IMM21)
END
```

The **BLE** instruction performs a relative jump if the contents of **Register A** are less than or equal to zero, i.e., the sign bit is set or the register is equal to zero.

BGE RA, IMM21

Branch Greater Than Or Equal

Opcode: **010101** (0x15)

```
IF NOT (Reg[RA] & 0x80000000) THEN
  PC += SignExtend(IMM21)
END
```

The **BGE** instruction performs a relative jump if the contents of **Register A** are greater than or equal to zero, i.e., the sign bit is clear.

BPE RA, IMM21

Branch Parity Even

Opcode: **001101** (0x0D)

```
IF NOT (Reg[RA] & 0x1) THEN
  PC += SignExtend(IMM21)
END
```

The **BPE** instruction performs a relative jump if the contents of **Register A** are even, i.e., the low bit is clear.

BPO RA, IMM21

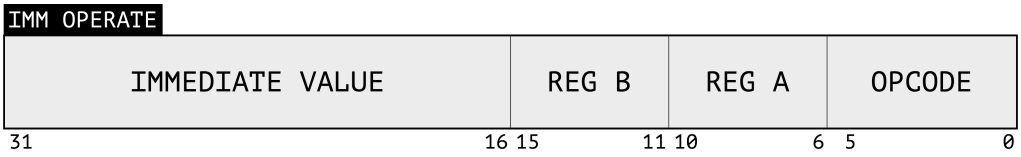
Branch Parity Odd

Opcode: **000101** (0x05)

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

The **BPO** instruction performs a relative jump if the contents of **Register A** are odd, i.e., the low bit is set.

5.2.3. Immediate Operate Format



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.

100 Group

ADDI RA, RB, IMM16
Add Immediate
Opcode: **111100** (0x3C)

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

The **ADDI** instruction performs an addition between the contents of **Register B** and a zero-extended 16-bit immediate value, storing the result in **Register A**.

SUBI RA, RB, IMM16
Subtract Immediate
Opcode: **110100** (0x34)

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

The **SUBI** instruction performs a subtraction between the contents of **Register B** and a zero-extended 16-bit immediate value, storing the result in **Register A**.

SLTI RA, RB, IMM16
Set Less Than Immediate
Opcode: **101100** (0x2C)

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

The **SLTI** instruction performs an unsigned less-than comparison between the contents of **Register B** and a zero-extended 16-bit immediate value. If the comparison is true, a **1** is stored in **Register A**. Otherwise, a **0** is stored.

5.2.3. Immediate Operate Format

SLTI SIGNED RA, RB, IMM16
Set Less Than Immediate, Signed
Opcode: **100100** (0x24)

`Reg[RA] = Reg[RB] s< SignExtend(IMM16)`

The **SLTI SIGNED** instruction performs a signed comparison between the contents of **Register B** and a sign-extended 16-bit immediate value. If the comparison is true, a **1** is stored in **Register A**. Otherwise, a **0** is stored.

ANDI RA, RB, IMM16
And Immediate
Opcode: **011100** (0x1C)

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

The **ANDI** instruction performs a bitwise AND between the contents of **Register B** and a zero-extended 16-bit immediate value, storing the result in **Register A**.

XORI RA, RB, IMM16
Xor Immediate
Opcode: **010100** (0x14)

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

The **XORI** instruction performs a bitwise XOR between the contents of **Register B** and a zero-extended 16-bit immediate value, storing the result in **Register A**.

ORI RA, RB, IMM16
Or Immediate
Opcode: **001100** (0x0C)

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

The **ORI** instruction performs a bitwise OR between the contents of **Register B** and a zero-extended 16-bit immediate value, storing the result in **Register A**.

LUI RA, RB, IMM16
Load Upper Immediate
Opcode: **000100** (0x04)

```
Reg[RA] = Reg[RB] | (IMM16 << 16)
```

The **LUI** instruction performs a bitwise OR between the contents of **Register B** and a zero-extended 16-bit immediate value which is shifted 16 bits to the left, storing the result in **Register A**.

---

011 Group

MOV RA, BYTE [RB + IMM16]
Load Byte, Immediate Offset
Opcode: **111011** (0x3B)

```
Reg[RA] = Load8(Reg[RB] + IMM16)
```

This instruction loads an 8-bit value into **Register A** from the address stored within **Register B** plus a zero-extended 16-bit immediate offset.

MOV RA, INT [RB + IMM16]
Load Int, Immediate Offset
Opcode: **110011** (0x33)

```
Reg[RA] = Load16(Reg[RB] + (IMM16 << 1))
```

This instruction loads a 16-bit value into **Register A** from the address stored within **Register B** plus a zero-extended 16-bit immediate offset shifted to the left by one.

MOV RA, LONG [RB + IMM16]
Load Long, Immediate Offset
Opcode: **101011** (0x2B)

```
Reg[RA] = Load32(Reg[RB] + (IMM16 << 2))
```

This instruction loads a 32-bit value into **Register A** from the address stored within **Register B** plus a zero-extended 16-bit immediate offset shifted to the left by two.

---

010 Group

MOV BYTE [RA + IMM16], RB
Store Byte, Immediate Offset
Opcode: **111010** (0x3A)

Store8(Reg[RA] + IMM16, Reg[RB])

This instruction stores the contents of **Register B** as an 8-bit value to the address stored within **Register A** plus a zero-extended 16-bit immediate offset.

MOV INT [RA + IMM16], RB
Store Int, Immediate Offset
Opcode: **110010** (0x32)

Store16(Reg[RA] + (IMM16 << 1), Reg[RB])

This instruction stores the contents of **Register B** as a 16-bit value to the address stored within **Register A** plus a zero-extended 16-bit immediate offset shifted to the left by one.

MOV LONG [RA + IMM16], RB
Store Long, Immediate Offset
Opcode: **101010** (0x2A)

Store32(Reg[RA] + (IMM16 << 2), Reg[RB])

This instruction stores the contents of **Register B** as a 32-bit value to the address stored within **Register A** plus a zero-extended 16-bit immediate offset shifted to the left by two.

MOV BYTE [RA + IMM16], IMM5
Store Byte, Small Immediate
Opcode: **011010** (0x1A)

Store8(Reg[RA] + IMM16, SignExtend(IMM5))

This instruction stores a sign-extended 5-bit immediate as an 8-bit value to the address stored within **Register A** plus a zero-extended 16-bit immediate offset. The **Register B** field of the instruction is interpreted as the 5-bit immediate.

MOV INT [RA + IMM16], IMM5
Store Int, Small Immediate
Opcode: **010010** (0x12)

```
Store16(Reg[RA] + (IMM16 << 1), SignExtend(IMM5))
```

This instruction stores a sign-extended 5-bit immediate as a 16-bit value to the address stored within **Register A** plus a zero-extended 16-bit immediate offset shifted left by one. The **Register B** field of the instruction is interpreted as the 5-bit immediate.

MOV LONG [RA + IMM16], IMM5
Store Long, Small Immediate
Opcode: **001010** (0x0A)

```
Store32(Reg[RA] + (IMM16 << 2), SignExtend(IMM5))
```

This instruction stores a sign-extended 5-bit immediate as a 32-bit value to the address stored within **Register A** plus a zero-extended 16-bit immediate offset shifted left by two. The **Register B** field of the instruction is interpreted as the 5-bit immediate.

---

000 Group

JALR RA, RB, IMM16
Jump And Link, Register
Opcode: **111000** (0x38)

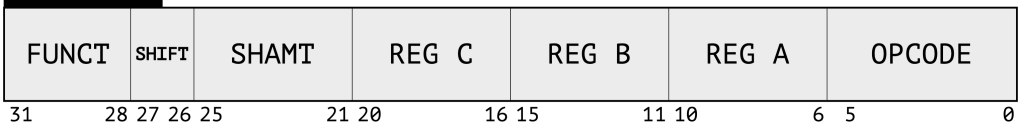
```
Reg[RA] = PC + 4
PC = Reg[RB] + (SignExtend(IMM16) << 2)
```

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 sign-extended immediate value shifted left by two.

This instruction can also be used to jump to the contents of a register in general, by setting the destination register to the **zero** register, thereby discarding the results.

5.2.4. Register Operate Format

REG OPERATE



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 4.1.2).

The value of Register C is shifted in the manner specified by the shift type, by the amount specified by the shift amount. A table of shift types follows:

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

Opcode 111001

MOV RA, BYTE [RB + RC xSH IMM5]
Load Byte, Register Offset
Function Code: **1111** (0xF)

```
Reg[RA] = Load8(Reg[RB] + (Reg[RC] xSH IMM5))
```

This instruction loads an 8-bit value into **Register A** from the address stored within **Register B**, plus the value of **Register C** shifted in the manner specified.

MOV RA, INT [RB + RC xSH IMM5]
Load Int, Register Offset
Function Code: **1110** (0xE)

```
Reg[RA] = Load16(Reg[RB] + (Reg[RC] xSH IMM5))
```

This instruction loads a 16-bit value into **Register A** from the address stored within **Register B**, plus the value of **Register C** shifted in the manner specified.

MOV RA, LONG [RB + RC xSH IMM5]
Load Long, Register Offset
Function Code: **1101** (0xD)

```
Reg[RA] = Load32(Reg[RB] + (Reg[RC] xSH IMM5))
```

This instruction loads a 32-bit value into **Register A** from the address stored within **Register B**, plus the value of **Register C** shifted in the manner specified.

MOV BYTE [RB + RC xSH IMM5], RA
Store Byte, Register Offset
Function Code: **1011** (0xB)

```
Store8(Reg[RB] + (Reg[RC] xSH IMM5), Reg[RA])
```

This instruction stores the contents of **Register A** as an 8-bit value to the address stored within **Register B**, plus the value of **Register C** shifted in the manner specified.

MOV INT [RB + RC xSH IMM5], RA
Store Int, Register Offset
Function Code: **1010** (0xA)

```
Store16(Reg[RB] + (Reg[RC] xSH IMM5), Reg[RA])
```

This instruction stores the contents of **Register A** as a 16-bit value to the address stored within **Register B**, plus the value of **Register C** shifted in the manner specified.

MOV LONG [RB + RC xSH IMM5], RA
Store Long, Register Offset
Function Code: **1001** (0x9)

```
Store32(Reg[RB] + (Reg[RC] xSH IMM5), Reg[RA])
```

This instruction stores the contents of **Register A** as a 32-bit value to the address stored within **Register B**, plus the value of **Register C** shifted in the manner specified.

LSH/RSH/ASH/ROR RA, RC, RB
Various Shift By Register Amount
Function Code: **1000** (0x8)

```
Reg[RA] = Reg[RC] xSH Reg[RB]
```

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 is ignored, and is taken from **Register B** instead.

ADD RA, RB, RC xSH IMM5
Add Register
Function Code: **0111** (0x7)

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

This instruction adds the contents of **Register B** to the contents of **Register C**, and stores the result into **Register A**. The contents of **Register C** are first shifted in the manner specified.

SUB RA, RB, RC xSH IMM5
Subtract Register
Function Code: **0110** (0x6)

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

This instruction subtracts the contents of **Register B** by the contents of **Register C**, and stores the result into **Register A**. The contents of **Register C** are first shifted in the manner specified.

SLT RA, RB, RC xSH IMM5
Set Less Than Register
Function Code: **0101** (0x5)

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

This instruction sets **Register A** to the result of an unsigned less-than comparison between the contents of **Register B** and the contents of **Register C**. The result is **1** if the comparison is true, and **0** otherwise. The contents of **Register C** are first shifted in the manner specified.

SLT SIGNED RA, RB, RC xSH IMM5
Set Less Than Register, Signed
Function Code: **0100** (0x4)

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

This instruction sets **Register A** to the result of a signed less-than comparison between the contents of **Register B** and the contents of **Register C**. The result is **1** if the comparison is true, and **0** otherwise. The contents of **Register C** are first shifted in the manner specified.

AND RA, RB, RC xSH IMM5
And Register
Function Code: **0011** (0x3)

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

This instruction performs a bitwise AND 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.

XOR RA, RB, RC xSH IMM5
Xor Register
Function Code: **0010** (0x2)

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

This instruction performs a bitwise XOR 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.

OR RA, RB, RC xSH IMM5
Or Register
Function Code: **0001** (0x1)

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

This instruction performs a bitwise OR 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.

NOR RA, RB, RC xSH IMM5
Nor Register
Function Code: **0000** (0x0)

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

This instruction performs a bitwise NOR 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.

Opcode 110001

MUL RA, RB, RC
Multiply
Function Code: **1111** (0xF)

```
Reg[RA] = Reg[RB] * Reg[RC]
```

This instruction performs an integer multiplication between the contents of **Register B** and the contents of **Register C**, and stores the result into **Register A**.

DIV RA, RB, RC
Divide
Function Code: **1101** (0xD)

```
Reg[RA] = Reg[RB] / Reg[RC]
```

This instruction performs an unsigned integer division between the contents of **Register B** and the contents of **Register C**, and stores the result into **Register A**. The result of the division is rounded down to the last whole integer.

DIV SIGNED RA, RB, RC
Divide, Signed
Function Code: **1100** (0xC)

```
Reg[RA] = Reg[RB] s/ Reg[RC]
```

This instruction performs a signed integer division between the contents of **Register B** and the contents of **Register C**, and stores the result into **Register A**. The result of the division is rounded toward zero to a whole integer.

MOD RA, RB, RC

Modulo

Function Code: **1011** (0xB)

```
Reg[RA] = Reg[RB] % Reg[RC]
```

This instruction performs an unsigned modulo between the contents of **Register B** and the contents of **Register C**, and stores the result into **Register A**. The modulo is the remainder part of the result of an unsigned division.

LL RA, RB

Load Locked

Function Code: **1001** (0x9)

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

This instruction is used to implement atomic sequences. It loads the 32-bit contents of a naturally-aligned memory address within **Register B** into **Register A**. It also sets two “registers” associated with the current processor: a “locked” flag is set to TRUE, and a “locked address” is set to the physical address being accessed. Though it is implementation-dependent, these “registers” likely do not reside on the processor itself, and may be implemented in any way as long as it provides the same semantics.

If the **RFE** *Return From Exception* instruction is executed on the current processor, the “locked” flag is cleared, causing a future **SC** instruction on this processor to fail. This is the only required behavior in a uniprocessor system. In a multiprocessor system, if any other processor performs a store instruction to this processor’s “locked address”, this processor’s “locked” flag is cleared. This can be used to implement atomic sequences in non-privileged (i.e. usermode) code.

SC RA, RB, RC

Store Conditional

Function Code: **1000** (0x8)

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

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**, and (like any other store instruction) clears the "locked" flag of any other processor with the same physical address locked by the **LL** *Load Locked* instruction.

PAUSE

Pause

Function Code: **0110** (0x8)

```
// Possible implementation.
PauseCount += 1
IF PauseCount ≥ 256 THEN
  PauseCount = 0
  Yield()
END
```

On multiprocessor systems, this instruction should be executed on each iteration of a spin-wait loop for another processor to do something (release a spinlock, acknowledge an IPI, etc). It serves as a hint that the processor isn't doing useful work, which can be used to optimize emulation software among other things.

MB

Memory Barrier

Function Code: **0011** (0x3)

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

This instruction need not be executed in a uniprocessor system. In a multiprocessor system, it ensures that, from the perspective of all other processors and I/O devices in the system, all prior writes performed by this processor have completed, as have all reads. One example of the usage of this instruction is to ensure data coherency after acquiring a spinlock.

WMB

Write Memory Barrier

Function Code: **0010** (0x2)

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

This instruction ensures that, from the perspective of all other processors and I/O devices in the system, all writes performed by this processor have completed. 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.

BRK

Breakpoint

Function Code: **0001** (0x1)

```
Exception(BRK)
```

This instruction causes a breakpoint exception. Its intended use is for debugging purposes. See Section 4.1.4.1.

SYS

System Service

Function Code: **0000** (0x0)

```
Exception(SYS)
```

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 call or “syscall”). See Section 4.1.4.1.

Opcode 101001 (Privileged Instructions)

These instructions all produce a **PRV** exception if executed while usermode is active. See Section 4.1.4.1.

MFCR RA, CR

Move From Control Register

Function Code: **1111** (0xF)

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

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**. See Section 1.4 for a full listing of control registers and their behaviors.

MTCR CR, RB

Move To Control Register

Function Code: **1110** (0xE)

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

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. See Section 1.4 for a full listing of control registers and their behaviors.

HLT

Halt Until Next Interrupt

Function Code: **1100** (0xC)

```
Halt()
```

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 until it is reset.

RFE

Return From Exception

Function Code: **1011** (0xB)

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

This instruction pops the “mode stack” of the **RS** control register (see Section 4.1.2), 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.

5.3. Pseudo-Instructions

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.

B IMM21

Unconditional Relative Branch

BEQ ZERO, IMM21

This pseudo-instruction performs an unconditional relative branch. This is synthesized out of the **BEQ** *Branch Equal* instruction, by comparing the contents of the register **ZERO** with the number zero; by definition, this will always be true.

RET

Return

JALR ZERO, LR, 0

This pseudo-instruction performs a common return from subroutine operation. This is synthesized out of 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).

JR RA

Jump to Register

JALR ZERO, RA, 0

This pseudo-instruction performs a jump to the contents of **Register A**. This is synthesized out of 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).

MOV RA, RB

Move Register

ADD RA, RB, ZERO LSH 0

This pseudo-instruction copies the contents of **Register B** into **Register A**. It is synthesized out of 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**.

LI RA, IMM16
Load 16-bit Immediate

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

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

LA RA, IMM32
Load 32-bit Immediate

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

This pseudo-instruction loads a 32-bit immediate into **Register A**. It is synthesized out of 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**.

NOP
No Operation

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

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, and this instruction set was carefully designed to ensure that that is an invalid instruction, so that exceptions will occur if the processor jumps off "into nowhere".

LSHI/RSHI/ASHI/RORI RA, RB, IMM5
Various Shift By Immediate Amount

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

These pseudo-instructions shift the contents of **Register B** by the 5-bit immediate, and saves the result in **Register A**. They are synthesized with 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.

MOV RA, BYTE/INT/LONG [IMM32]

Load From 32-bit Address

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

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

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
```

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 instructions. The upper 16 bits of the address are loaded into a user-supplied temporary register with **LUI**, and then a store is done with the offset being the low 16 bits of the address.
