ANY-1 Reference Guide

(c) 2021 Robert Finch

Table of Contents

Programming Model

Registers

Overview

The ANY-1 is a vector machine. The ISA is a 64-register machine with a unified register file for integer, floating-point, decimal-floating-point or posit arithmetic. There are many control and special/status (CSR) registers which hold an assortment of specific values relevant to processing.

General Purpose Registers (x0 to x63) / Scalar Registers

The register usage convention probably has more to do with software than hardware. Excepting a few special cases, the registers are general purpose in nature. Registers may hold integer, floating-point, decimal floating-point or posit values.

x0 always has the value zero. Register x63 is a read only alias of the instruction pointer register. Registers x61 and x62 are used for stack references and subject to stack bounds checking.

Register	Description / Suggested Usage	Saver
x0	always reads as zero (hardware)	
x2	constant building / temporary (cb)	
x3-x9	temporaries (t0-t6)	caller
x10-x19	register variables (s0-s9)	callee
x20-x27	function arguments (a0-a7) a7/g2	caller
x59	thread pointer (tp / g1)	
x60	global data pointer (g0)	callee
x61	base / frame pointer (fp)	callee
x62	current stack pointer (sp)	callee
x63	instruction pointer	

Base Registers

Base registers are used as part of the memory management unit of the processing core and are further described in the mmu section of the document.

Base Regno	Usage	Selected By
b0 to b7	data	bits 60 to 63 of effective address
b8, b9	reserved	bits 60 to 63 of effective address
b10	Stack	bits 60 to 63 of effective address
b11	I/O	bits 60 to 63 of effective address
b12 to b15	code	bits 60, 63 of instruction pointer

Control and Status Registers

Overview

There are numerous special purpose control and status registers in the design. Some registers are present to store variables for performance reasons that would otherwise be stored in main memory.

[U/S/H/M/D]_CAUSE (0x?006)

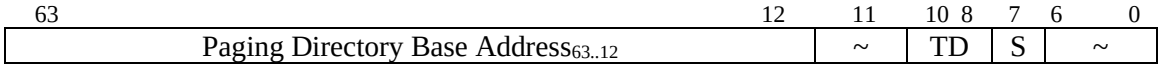
This register contains a code indicating the cause of an exception or interrupt. The break handler will examine this code to determine what to do. Only the low order 16 bits are implemented. The high order bits read as zero and are not updateable.

U_SEMA (CSR 0x000C) Semaphores

This register is available for user semaphores or flag use. Bits in this CSR may be set or cleared with one of the CSRxx instructions. This register has individual bit set / clear capability.

S_PTA (0x1003)

This register contains the base address of the highest-level page directory for memory management, the paging table depth and the size of the pages mapped. The base address must be page aligned (4kB).



TD	
0	1 level lookup
1	2 level lookup
2	3 level lookup
3	4 level lookup
4 to 7	reserved

S	
0	map 16kB pages
1	map 4MB pages

S_TID (CSR 0x1010)

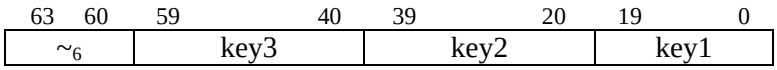
This CSR register is reserved for use to contain the task id for the currently running task.

S_ASID – (CSR 0x101F)

This register contains the address space identifier (ASID) or memory map index (MMI). The ASID is used in this design to select (index into) a memory map in the paging tables.

S_KEYS – (CSR 0x1020 to 0x1022)

These registers contain a collection of keys associated with the process for the memory system. Each key is twenty bits in size. Each register contains three keys for a total of nine keys. All three registers are searched in parallel for keys matching the one associated with the memory page. Keyed memory enhances the security and reliability of the system.



M_TVEC (0x3030 to 0x3033)

These registers are an alias for the D_TVEC registers 0x4030 to 0x4033 which allows access to the TVEC registers from machine mode. They contain the address of the exception handling routine for a given operating level. The lower bits of the exception address are determined from the operating level. TVEC[1] to TVEC[4] are used by the REX instruction.

D_TVEC (0x4030 to 0x4034)

These registers contain the address of the exception handling routine for a given operating level. TVEC[4] (0x4034) is used directly by hardware to form an address of the debug routine. The

lower eight bits of TVEC[4] are not used. The lower bits of the exception address are determined from the operating level. TVEC[1] to TVEC[4] are used by the REX instruction.

Operating Modes

The core has five operating modes. The highest operating mode is operating mode four which is called the debug operating mode. Debug operating mode has complete access to the machine including special registers and features reserved for debug. Other operating levels may have more restricted access. When an interrupt occurs, the operating mode is set to the debug mode. The core vectors to an address depending on the current operating mode. When not operating at user mode addresses are not subjected to translation and the virtual address and physical address are the same.

Operating Mode	Moniker
0	user
1	supervisor
2	hypervisor
3	machine
4	debug

Switching Operating Modes

The operating mode is automatically switched to the debug mode when an interrupt occurs. The BRK instruction may be used to switch operating modes. The REX instruction may also be used by a handler to switch the operating mode to a lower mode. One of the exception return instructions (RTD, RTE) will switch the operating level back to what it was prior to the interrupt or exception.

Memory Management Unit - MMU

Introduction

Many systems can benefit from the provision of virtual memory management. Virtual memory may be used to protect the address space of one app from another. Virtual memory can enhance the reliability and security of a system.

The simplified system MMU provides minimalistic base and bound and paging capabilities for a small to mid size system. There are two options available for paging, a simple page map ram, and a software managed TLB. The page mapping ram is not suitable for larger systems as the paging tables would be too large. Base bound and paging are applied only to user mode apps. In other operating modes the system sees a flat address space with no restrictions on access. Base address generation is applied to virtual addresses first to generate a linear address which is then mapped using a paged mapping system. Access rights are governed by the base register since all pages in the based on the same address are likely to require the same access. Support for access rights is optional if it is desired to reduce the hardware cost. To simplify hardware there are no bound registers. Bounds are determined by what memory is mapped into the base address area.

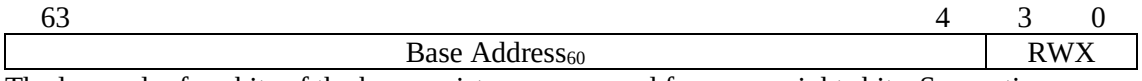
Associated with each memory page and stored in its own table is a memory key. The memory key is matched against the keyset in CSR registers. Access to the memory page is allowed only if one of the keys in the keyset matches the memory key, or if the page is marked generally accessible with the special key of zero. Memory may be shared between apps that share the same memory key.

Base Registers

The upper address bits of a virtual or effective address are not used for addressing memory and are available to select base register. The MMU includes 16 base registers. The base register in use is selected by the upper nybble of the virtual address. If the program address has all ones in bits 24 to 63 then base addressing is bypassed. This provides a shared program area containing the BIOS and OS code.

Base Regno	Usage	Selected By
0 to 7	data	bits 60 to 63 of effective address
8, 9	reserved	bits 60 to 63 of effective address
10	Stack	bits 60 to 63 of effective address
11	I/O	bits 60 to 63 of effective address
12 to 15	code	bits 60, 63 of instruction pointer

Base Register Format



The low order four bits of the base register are reserved for access rights bits. Supporting memory access rights is optional.

R: 1 = segment readable

W: 1 = segment writeable

X: 1 = segment executable

Base Register Access

Base registers may be read and altered using the MVBASE instruction. The MVBASE instruction works in an indirect fashion as described in the text.

Linear Address Generation

The base address value contained in the upper 60 bits of a base register is shifted left 14 bits before being added to the virtual address. This gives potentially a 74-bit address space.

Note there is no limit or bound register. Access is limited by what is mapped into the segment. Unmapped page addresses result in a page fault exception. Pages that are inaccessible use the reserved page number of all ones.

The Page Map

The page directly maps virtual address pages to physical ones. The page map is a dedicated memory internal to the processing core accessible with the custom 'mvmap' instruction. It is similar in operation to a TLB but is much simpler. TLB's cache address translations and create TLB miss exceptions. Page walks of mapping tables are required to update the TLB on a miss. There are no exceptions associated with the page mapping table.

In addition to based addresses, memory is divided up into 16kB pages which are mapped. There are 32 memory maps available. A memory map represents an address space; a five-bit address space identifier is in use. Address spaces will need to be shared if more than 32 apps are running in the system. The desire is to keep the mapping tables small so they may fit into a small number of standard memory blocks. For instance, for the sample system there are 4096 pages required to map the 256MB address space. Any individual app is limited to maximum of 64MB (one quarter of the memory available). The virtual page number is used to lookup the physical page in the page mapping table. Addresses with the top eight bits set are not mapped to allow access to the system ROM. Pages that are inaccessible use the reserved physical page number of all ones.

The page mapping table is indexed by the ASID and the virtual page number to determine the physical page. The 'mvmap' instruction uses Rs1 to contain a mapping table index. Bits 16 to 20 of Rs1 are the ASID, bits 0 to 15 of Rs1 are used for the virtual page number. It is expected that the virtual page number is a small number, in this case 12 bits. Rs2 contains the new value of the physical page. The current value of the physical page is placed in Rd when the instruction executes.

Page Map Entry Layout

41 36	35 24	23	22	21	20	19	18	17	16	15 0
$ASID_5$	Virtual Page	G	D	A	U	C	R	W	X	Physical Page

0	0									10
	1									11
	…									
	4094									18
	4095									19
1	0									
	1									
	…									
	4094									

	4095									
… 30 more address spaces										

The low order 14 bits of an address pass through both linear address generation and paging unchanged.

The 16kB Page

Many memory systems use a 4kB page size. A 16kB page size is used here mainly to restrict the number of page entries in the page map table. A smaller page size would result in too many pages of memory to support multiple tasks. Even given a 16kB page size there are still 4096 pages of memory available in a map.

> *The author was tempted to divide the page mapping table into several different regions capable of mapping different amounts of the address space (small, medium, and large areas). This potentially could allow more memory maps to be present while at the same time not increasing the page table size. However, it would add extra complexity to the memory system which is currently simpler in nature.*

The MVMAP Instruction

The memory mapping table is managed with a dedicated instruction - MVMAP. MVMAP allows high-speed access to the mapping table.

> *While the memory mapping table could have been managed with CSRs or possibly be mapped into the main memory space, the author feels that having a dedicated instruction makes the software managing the tables simpler and cleaner.*

Rs1:

63 20	20 16	15 0
Unused - should be zero	$ASID_5$	Virtual page number 16 bits max

TLB – Translation Lookaside Buffer

Overview

The page map is limited in the translations it can perform because of its size. The solution to allowing more memory to be mapped is to use main memory to store the translations tables, then cache address translations in a translation look-aside buffer or TLB. This is sometimes also called an address translation cache ATC. The TLB offers a means of address virtualization and memory protection. A TLB works by caching address mappings between a real physical address and a virtual address used by software. The TLB deals with memory organized as pages. Typically, software manages a paging table whose entries are loaded into the TLB as translations are required.

Size / Organization

The TLB has 1024 entries per set. The size was chosen as it is the size of one block ram for 32-bit data in the FPGA. This is quite a large TLB. Many systems use smaller TLBs. There is not really a need for such a large one, however it is available.

The TLB is organized as a four-way set associative cache.

What is Translated

The TLB processes all user mode addresses including both instruction and data addresses. It is known as a *unified* TLB. Addresses in other modes of operation are not translated. Additionally, addresses with the top forty bits set are not translated to allow access to the BIOS and system rom.

Page Size

Because the TLB caches address translations it can get away with a much smaller page size than the page map can for a larger memory system. 4kB is a common size for many systems. In this case the TLB uses 16kB pages to match the size of pages for keyed memory and segmentation. For a 256MB system (the size of the memory in the test system) there are 16,384 16kB pages.

Management

The rtf64 TLB unit is a software managed TLB. When a translation miss occurs, an exception is generated to allow software to update the TLB. It is left up to software to decide how to update the TLB. There may be a set of hierarchal page tables in memory, or there could be a hash table used to store translations.

The TLB is updated using the TLBRW instruction which both reads and writes the TLB. More descriptive text is present at the TLBRW instruction description.

Flushing the TLB

The TLB maintains the address space (ASID) associated with a virtual address. This allows the TLB translations to be used without having to flush old translations from the TLB during a task switch.

Global Bit

In addition to the ASID the TLB entries contain a bit that indicates that the translation is a global translation and should be present in every address space.

PAM – Page Allocation Map

Overview

The PAM is a software structure made up of 131,072 bit-pairs stored in memory. There is a bit pair for each possible physical memory page. The PAM is used by software to manage the allocation of physical pages of memory.

> *The author initially had a specialized PAM unit and supporting instructions in the core. However, there was difficulty getting the unit to work correctly and it was also limited in size. The PAM can be easily managed by software which makes use of EXT and DEP instructions to perform bit manipulations.*

Memory Usage

Total memory used by the PAM is 32kB.

Organization

The PAM is organized as a string of bit-pairs, one pair for each physical memory page. Bit pairs are used rather than single bits to mark allocated pages as it is convenient to also mark runs of pages. Marking runs of pages using bit-pairs makes it possible to free the pages of a previous allocation.

Bit-Pair Value	Meaning
0	Page of memory is free, available for use.
1	reserved
2	Page is allocated, end of run of pages
3	Page is allocated

PMA - Physical Memory Attributes Checker

Overview

The physical memory attributes checker is a hardware module that ensures that memory is being accessed correctly according to its physical attributes.

Physical memory attributes are stored in an eight-entry table. This table includes the address range the attributes apply to and the attributes themselves. Address ranges are resolved only to bit four of the address. Meaning the granularity of the check is 16 bytes.

Most of the entries in the table are hard-coded and configured when the system is built.

Physical memory attributes checking is applied in all operating modes.

Register Description

Regno	Bits		
00	64	LB0	lower bound - address bits 4 to 67 of the physical address range
08	64	UB0	upper bound - address bits 4 to 67 of the physical address range
10	16	AT0	memory attributes
18	~	~	reserved
…	…	…	6 more register sets
E0	64	LB7	lower bound - address bits 4 to 67 of the physical address range
E8	64	UB7	upper bound - address bits 4 to 67 of the physical address range
F0	16	AT7	memory attributes
F8	~	~	reserved

Attributes

Bitno		
0	X	may contain executable code
1	W	may be written to
2	R	may be read
3	C	may be cached
4-6	G	granularity G: 0 = byte accessible; 1 = wyde accessible; 2 = tetra accessible; 3 = octa accessible; 4 to 7 = reserved
7	~	reserved
8-15	T	device type (rom, dram, eeprom, I/O, etc)

Key Memory

Overview

Associated with each page of memory is a memory key. To access a page of memory the memory key must match with one of the keys in the applications keyset. The keyset is maintained in the keys CSRs. The key size of 20 bits is a minimum size recommended for security purposes.

Since each page of memory requires a 20-bit key and there are 16,384 pages in the system a 40kB memory is required to store keys (10 block rams).

The SETKEY Instruction

To access the key memory, the SETKEY instruction is used. This instruction may set or get the current key for a memory page. For SETKEY the Rs1 register contains a value in the following format:

63 46	45 32	31 20	19 0
~	Physical Page Number	~	Key Value

More detail is present in the instruction description.

Card Memory

Overview

Also present in the memory system is Card memory. The card memory is a telescopic memory which reflects with increasing detail where in the memory system a pointer write has occurred. This is for the benefit of garbage collection systems. Card memory is updated when a pointer value is stored to memory. (The store pointer STPTR instruction is used to do this).

> *Garbage collection has become more prevalent in many systems and the author feels it makes sense to provide some hardware support for this function.*

Organization

Memory is divided into 512-byte card memory pages. Each card has a single bit recording whether a pointer store has taken place in the corresponding memory area. To cover a 256MB memory system 64kB card memory (16 block rams) is required at the outermost layer. The outer most 64kB card memory layer is itself divided into 64-bit regions and covered with a 512B memory. Note that each bit represents the pointer store status for a 64kB region. The 512B memory is further resolved into 64-byte regions and covered with an 8B (64 bit) memory or one octa-byte.

Table showing increasing resolution of telescopic memory. Note the layer 1 memory access is 32-bits at a time rather than 64-bit like the other layers. This is due to block ram multiplexing issues. The block rams can only adapt from 1 bit wide on one port to 32-bits wide on the other (dual ported ram is required). It is accessible as 128, 32-bit words where each bit in the word represents a 64kB memory area. Each bit in layer zero represents a 4MB region of memory.

Layer	Resolving Power	
0	64 bits	4MB regions
1	4096 bits	64kB regions
2	524288 bits	512B regions

The choice of 512 bytes comes from the desire to minimize the amount of card memory required. Card memory is implemented using dedicated block memory in an FPGA and there are only so many block memories available. The author would have preferred a better resolution, but it may consume to many memory blocks.

Operation

As a program progresses it writes pointer values to memory using the store pointer instruction. Storing a pointer triggers an update to all the layers of card memory corresponding to the main memory location written. A bit is set in each layer of the card memory system corresponding to the memory location of the pointer store.

The garbage collection system can very quickly determine where pointer stores have occurred and skip over memory that has not been modified.

Accessing Card Memory (GCCLR)

Card memory is accessible with the GCCLR instruction. The GCCLR instruction may be used to return the status of the card memory then clear it. The instruction has an ability to atomically access the memory reading current status and updating the memory as one operation.

The author initially had the card memory memory-mapped into different sized areas for each layer of the card memory. However, accessing it as memory would be slow and it complicated the memory states of the core. Accessing the memory via an instruction is an order of magnitude faster than using the memory system.

Summary

Having dedicated card memory and instruction support is more expensive than managing the card memory via software alone. It should offer some improvement in terms of performance and software simplicity.