ANY-1 Reference Guide

(c) 2021 Robert Finch

Table of Contents

Programming Model 3
Registers 3
Overview 3
General Purpose Registers (x0 to x63) / Scalar Registers 3
Base Registers 3
Control and Status Registers 4
Overview 4
[U/S/H/M/D]_CAUSE (0x?006) 4
U_SEMA (CSR 0x000C) Semaphores 4
S_PTA (0x1003) 4
S_TID (CSR 0x1010) 5
S_ASID – (CSR 0x101F) 5
S_KEYS – (CSR 0x1020 to 0x1022) 5
M_BADADDR (CSR 0x3007) 6
M_BAD_INSTR (CSR 0x300B) 6
M_TVEC (0x3030 to 0x3033) 6
D_TVEC (0x4030 to 0x4034) 7
Operating Modes 9
Switching Operating Modes 9
Memory Management Unit - MMU 16
Introduction 16
Base Registers 16
Base Register Format 16
Base Register Access 17
Linear Address Generation 17
The Page Map 17
Page Map Entry Layout 17
The 16kB Page 18
The MVMAP Instruction 18

TLB – Translation Lookaside Buffer 18
Overview 18
Size / Organization 18
What is Translated 19
Page Size 19
Management 19
Flushing the TLB 19
PAM – Page Allocation Map 19
Overview 19
Memory Usage 19
Organization 19
PMA - Physical Memory Attributes Checker 20
Overview 20
Register Description 20
Attributes 20
Key Cache 21
Overview 21
Card Memory 21
Overview 21
Organization 21
Location 21
Operation 21
Sample Write Barrier 22
System Memory Map 22

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	

General Purpose Vector (v0 to v63) / Registers

v0 always has the value zero.

Register	Description / Suggested Usage	Saver
v0	always reads as zero (hardware)	
v1-v63		

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.

U_FSTAT (CSR 0x0014) Floating Point Status and Control Register

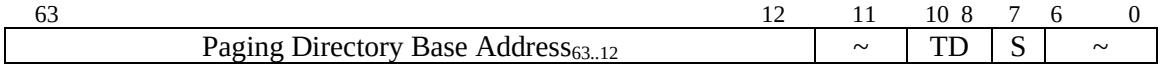
The floating-point status and control register may be read using the CSR instruction. Unlike other CSR's the control register has its own dedicated instructions for update. See the section on floating point instructions for more information.

Bit		Symbol	Description
63:53			reserved
52		inexact	inexact
51		dbz	divide by zero
50		under	underflow
49		over	overflow
48		invop	invalid operation
47		~	reserved
46:44	**RM**	rm	rounding mode
43	**E5**	inexe	- inexact exception enable
42	**E4**	dbzxe	- divide by zero exception enable
41	**E3**	underxe	- underflow exception enable
40	**E2**	overxe	- overflow exception enable
39	**E1**	invopxe	- invalid operation exception enable
38	**NS**	ns	- non standard floating point indicator
Result Status			
32		fractie	- the last instruction (arithmetic or conversion) rounded intermediate result (or caused a disabled overflow exception)
31	**RA**	rawayz	rounded away from zero (fraction incremented)
30	**SC**	C	denormalized, negative zero, or quiet NaN
29	**SL**	neg <	the result is negative (and not zero)
28	**SG**	pos >	the result is positive (and not zero)
27	**SE**	zero =	the result is zero (negative or positive)
26	**SI**	inf ?	the result is infinite or quiet NaN
Exception Occurrence			
21 to 25			reserved
20	**X6**	swt	{reserved} - set this bit using software to trigger an invalid operation
19	**X5**	inerx	- inexact result exception occurred (sticky)
18	**X4**	dbzx	- divide by zero exception occurred

17	**X3**	underx	- underflow exception occurred
16	**X2**	overx	- overflow exception occurred
15	**X1**	giopx	- global invalid operation exception – set if any invalid operation exception has occurred
14	**GX**	gx	- global exception indicator – set if any enabled exception has happened
13	**SX**	sumx	- summary exception – set if any exception could occur if it was enabled - can only be cleared by software
Exception Type Resolution			
8 to 12			reserved
7	**X1T**	cvt	- attempt to convert NaN or too large to integer
6	**X1T**	sqrtx	- square root of non-zero negative
5	**X1T**	NaNCmp	- comparison of NaN not using unordered comparison instructions
4	**X1T**	infzero	- multiply infinity by zero
3	**X1T**	zerozero	- division of zero by zero
2	**X1T**	infdiv	- division of infinities
1	**X1T**	subinfx	- subtraction of infinities
0	**X1T**	snanx	- signaling NaN

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)

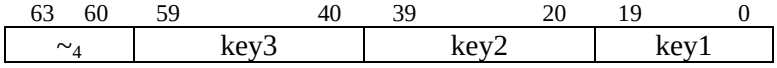
This CSR register is reserved for use to contain the task id for the currently running task.

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.



Control Register Zero (CSR #0x3000)

This register contains miscellaneous control bits including a bit to enable protected mode.

Bit		Description
0	Pe	Protected Mode Enable: 1 = enabled, 0 = disabled
8 to 13		
16		
29	RASE	return address stack predictor enable 1=enabled, 0 = disabled
30	DCE	data cache enable: 1=enabled, 0 = disabled
32	BPE	branch predictor enable: 1=enabled, 0=disabled
33	BTBE	branch target buffer enable 1=enabled, 0=disabled
34	WBM	write buffer merging enable: 1 = enabled, 0 = disabled
35	SPLE	speculative load enable (1 = enable, 0 = disable) (0 default)
36		
63	D	debug mode status. this bit is set during an interrupt routine if the processor was in debug mode when the interrupt occurred.

This register supports bit set / clear CSR instructions.

DCE

Disabling the data cache is useful for some codes with large data sets to prevent cache loading of values that are used infrequently. Disabling the data cache may reduce security risks for some kinds of attacks. The instruction cache may not be disabled. Enabling / disabling the data cache is also available via the cache instruction.

BPE

Disabling branch prediction will significantly affect the cores performance but may be useful for debugging. Disabling branch prediction causes all branches to be predicted as not-taken. No entries will be updated in the branch history table if the branch predictor is disabled.

WBM bit

Merging of values stored to memory may be disabled by setting this bit. On reset write buffer merging is disabled because it is likely desirable to setup I/O devices. Many I/O devices require updates to individual bytes by separate store instructions. (Write buffer merging is not currently implemented).

SPLE

Enabling speculative loads give the processor better performance at an increased security risk to meltdown attacks.

M_BADADDR (CSR 0x3007)

This register contains the effective address for a load / store operation that caused a memory management exception or a bus error. Note that the address of the instruction causing the exception is available in the XL register.

M_BAD_INSTR (CSR 0x300B)

This register contains a copy of the exceptioned instruction.

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.

M_PM_STACK (0x3040)

This register contains an eight-entry operating mode and interrupt mask stack. When an exception or interrupt occurs, this register is shifted to the left by four bits and the low order bits are set according to the exception mode, when an RTI instruction is executed this register is shifted to the right by four bits. On RTI the last stack entry is set to $9 masking all interrupts on stack underflow. The low order four bits represent the current operating mode and interrupt mask. Only the low order 32 bits of the register are implemented.

M_STATUS (0x3044)

This register contains the interrupt mask, operating mode, and privilege level.

Bitno	Field	Description
0 to 3	IM	active interrupt mask level
4 to 5	~	reserved
6 to 13	PL	privilege level
14 to 19	RS	register set selection – general purpose registers, this also controls which bounds register set is viewable in the CSRs.
20	SX	software exception – typically set by a throw() operation and cleared in a catch() handler.
20 to 21	~	reserved
24 to 27	Thrd	active thread
28 to 31	IRQ	The level of interrupt that caused the hardware BRK.
32	VCA	indicates that vector chaining was active prior to an exception
40 to 47	~	reserved
48 to 49	FS	floating point state
50 to 51	XS	additional core extension state
55	MPRV	memory privilege: This bit when true (1) causes memory operations to use the first stack privilege level when evaluating privilege and protection rules. (Bits 0 to 31 in the pm_stack reg).
56 to 60	VM	These bits control virtual memory options. Note that multiple options may be present at the same time. At reset all the bits are set to zero.
63	SD	

VM_5

Bit	Indicates	
0	1 = single bound	
1	1 = separate program and data bounds	
2	1 = lot protection system	
3	1 = simplified paged unit	

4	1 = paging unit	

M_EIP (0x3048)

This register contains the interrupt or exception instruction pointer register.

D_TIME (0x?FE0)

The TIME register corresponds to the wall clock real time. This register can be used to compute the current time based on a known reference point. The register value will typically be a fixed number of seconds offset from the real wall clock time. The lower 32 bits of the register are driven by the tm_clk_i clock time base input which is independent of the cpu clock. The tm_clk_i input is a fixed frequency used for timing that cannot be less than 10MHz. The low order 32 bits represent the fraction of one second. The upper 32 bits represent seconds passed. For example, if the tm_clk_i frequency is 100MHz the low order 32 bits should count from 0 to 99,999,999 then cycle back to 0 again. When the low order 32 bits cycle back to 0 again, the upper 32 bits of the register is incremented. The upper 32 bits of the register represent the number of seconds passed since an arbitrary point in the past.

Note that this register has a fixed time basis, unlike the TICK register whose frequency may vary with the cpu clock. The cpu clock input may vary in frequency to allow for performance and power adjustments.

Tags

Instructions are tagged with a four bit value in bits 60 to 63 to help determine the operation of the instruction.

Tag	Meaning	
0000	pointer	
0001	integer	
0010	float	
0011	120 bit float	
0100	decimal float	
0101	120 bit decimal float	
0110	posit	
0111	Boolean	
1000	UTF20 string	three UTF20 characters
1001	UTF10 string	six UTF10 characters
…	reserved	
1111	unknown	

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.

Exceptions

External Interrupts

There is little difference between an externally generated exception and an internally generated one. An externally caused exception will force a BRK instruction into the instruction stream. The BRK instruction contains a cause code identifying the external interrupt source.

Polling for Interrupts

To support code that needs to run with interrupts disabled an interrupt polling instruction (PFI) is provided in the instruction set. For instance, the system could be running a high priority task with interrupts disabled. There may be sections of code where it is possible to process an interrupt however. In some code environments, it is not enough to disable and enable interrupts around critical code. The code must be effectively run with interrupt disabled all the time. This makes it necessary to poll for interrupts in software. For instance, stack prologue code may cause false pointer matches for the garbage collector because stack space is allocated before the contents are defined. If the GC scan occurs on this allocated but undefined area of memory, there could be false matches.

Effect on Machine Status

The operating mode is always switched to the debug mode on exception. It is up to the debug mode code to redirect the exception to a lower operating mode when desired. Further exceptions at the same or lower interrupt level are disabled automatically. Debug mode code must enable interrupts at some point.

Exception Stack

The current register set, operating mode and interrupt enable bits are pushed onto an internal stack when an exception occurs. This stack is only eight entries deep as that is the maximum amount of nesting that can occur. Further nesting of exceptions can be achieved by saving the state contained in the exception registers.

Exception Vectoring

Exceptions are handled through a vector table. The vector table has six entries, one for each operating level the core may be running at. The location of the vector table is determined by TVEC[5]. If the core is operating at mode three for instance and an interrupt occurs vector table address number three is used for the interrupt handler. Note that the interrupt automatically switches the core to operating mode five. An exception handler at the machine level may redirect exceptions to a lower level handler identified in one of the vector registers. More specific exception information is supplied in the cause register.

Operating Level	Address (If TVEC[5] contains $F…FC0000)	
0	$F…FC0000	Handler for operating level zero
1	$F…FC0020	
2	$F…FC0040	
3	$F…FC0060	
4	$F…FC0080	

Reset

The core begins executing instructions at address $F…FD0000. All registers are in an undefined state. Register set #0 is selected.

Precision

Exceptions in ANY1 are precise. They are processed according to program order of the instructions. If an exception occurs during the execution of an instruction, then an exception field is set in the reorder buffer. The exception is processed when the instruction commits which happens in program order. If the instruction was executed in a speculative fashion, then no exception processing will be invoked unless the instruction makes it to the commit stage.

Exception Cause Codes

The following table outlines the cause code for a given purpose. These codes are specific to ANY1. Under the HW column an ‘x’ indicates that the exception is internally generated by the processor; the cause code is hard-wired to that use. An ‘e’ indicates an externally generated interrupt, the usage may vary depending on the system.

Cause Code		HW	Description	
0			no exception	
1	IBE	x	instruction bus error	
2	EXF	x	Executable fault	
4	TLB	x	tlb miss	
			FMTK Scheduler	
128		e		
129	KRST	e	Keyboard reset interrupt	
130	MSI	e	Millisecond Interrupt	
131	TICK	e		
156	KBD	e	Keyboard interrupt	
157	GCS	e	Garbage collect stop	
158	GC	e	Garbage collect	
159	TSI	e	FMTK Time Slice Interrupt	
3			Control-C pressed	
20			Control-T pressed	
26			Control-Z pressed	
32	SSM	x	single step	
33	DBG	x	debug exception	
34	TGT	x	call target exception	
35	MEM	x	memory fault	
36	IADR	x	bad instruction address	
37	UNIMP	x	unimplemented instruction	
38	FLT	x	floating point exception	
39	CHK	x	bounds check exception	
40	DBZ	x	divide by zero	
41	OFL	x	overflow	
47				

48	ALN	x	data alignment	
49	KEY	x	memory key fault	
50	DWF	x	Data write fault	
51	DRF	x	data read fault	
52	SGB	x	segment bounds violation	
53	PRIV	x	privilege level violation	
54	CMT	x	commit timeout	
55	BT	x	branch target	
56	STK	x	stack fault	
57	CPF	x	code page fault	
58	DPF	x	data page fault	
60	DBE	x	data bus error	
61	PMA	x	physical memory attributes check fail	
62	NMI	x	Non-maskable interrupt	
225	FPX_IOP	x	Floating point invalid operation	
226	FPX_DBZ	x	Floating point divide by zero	
227	FPX_OVER	x	floating point overflow	
228	FPX_UNDER	x	floating point underflow	
229	FPX_INEXACT	x	floating point inexact	
231	FPX_SWT	x	floating point software triggered	
239			Software exception handling	
240	SYS		Call operating system (FMTK)	
241			FMTK Schedule interrupt	
242	TMR	x	system timer interrupt	
243	GCI	x	garbage collect interrupt	
253	RST	x	reset	
254	NMI	x	non-maskable interrupt	
255	PFI		reserved for poll-for-interrupt instruction	

DBG

A debug exception occurs if there is a match between a data or instruction address and an address in one of the debug address registers.

IADR

This exception is currently not implemented but reserved for the purpose of identifying bad instruction addresses. If the two least significant bits of the instruction address are non-zero then this exception will occur.

UNIMP

This exception occurs if an instruction is encountered that is not supported by the processor. It may also occur if there is an attempt to use an instruction in a mode that does not support it.

OFL

If an arithmetic operation overflows (multiply, add, or shift) and the overflow exception is enabled in the arithmetic exception enable register then an OFL exception will be triggered.

KEY

This fault will occur if an attempt is made to access memory for which the app does not have the key.

FLT

A floating-point exception is triggered if an exceptional condition occurs in the floating-point unit and the exception is enabled. Please see the section on floating-point for more details.

DRF, DWF, EXF

Data read fault, data write fault, and execute fault are exceptions that are returned by the memory management unit when an attempt is made to access memory for which the corresponding access type is not allowed. For instance, if the memory page is marked as non-executable an attempt is made to load the instruction cache from the page then an execute fault EXF exception will occur.

CPF, DPF

The code page fault and data page fault exceptions are activated by the mmu if the page is not present in memory. Access may be allowed but simply unavailable. These faults are not currently implemented.

PRIV

Some instructions and CSR registers are legal to use only at a higher operating level. If an attempt is made to use the privileged instruction by a lower operating level, then a privilege violation exception may occur. For instance, attempting to use RTI instruction from user operating level.

STK

If the value loaded into one of the stack pointer registers (the stack pointer sp or frame pointer fp) is outside of the bounds defined by the stack bounds registers, then a stack fault exception will be triggered.

DBE

A timeout signal is typically wired to the err_i input of the core and if the data memory does not respond with an ack_i signal fast enough an error will be triggered. This will happen most often when the core is attempting to access an unimplemented memory area for which no ack signal is generated. When the err_i input is activated during a data fetch, an exception is flagged in a result register for the instruction. The core will process the exception when the instruction commits. If the instruction does not commit (it could be a speculated load instruction) then the exception will not be processed.

PMA

The addressed memory did not pass the physical memory attributes testing. For example a write operation attempted to a ROM address space.

IBE

A timeout signal is typically wired to the err_i input of the core and if the instruction memory does not respond with an ack_i signal fast enough and error will be triggered. This will happen most often when the core is attempting to access an unimplemented memory area for which no ack signal is generated. When the err_i input is activated during an instruction fetch, a breakpoint instruction is loaded into the cache at the address of the error.

NMI

Non-maskable interrupt.

BT

The core will generate the BT (branch target) exception if a branch instruction points back to itself. Branch instructions in this sense include jump (JMP) and call (CALL) instructions.

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.

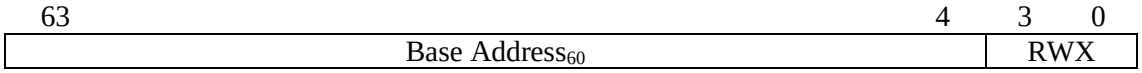
Associated with each memory page and stored in its own table is a memory key. The memory key is matched against the keyset in CSR registers. Access to the memory page is allowed only if one of the keys in the keyset matches the memory key, or if the page is marked generally accessible with the special 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 512MB system (the size of the memory in the test system) there are 32,768 16kB pages.

Management

The 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

Memory is organized into 32,768 16kB pages.

The PAM is a software structure made up of 32,768 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.

Memory Usage

Total memory used by the PAM is 8kB.

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 Cache

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.

The key associated with each memory page is stored in a table in main memory. Each key occupies a tetra-byte of memory to keep caching simple. So that two memory accesses are not required to access a page of memory this table of keys is cached. When a page of memory is accessed the key cache is accessed in parallel.

The key cache is a direct mapped cache organized as 256 lines of 16 keys. Key values are stored in LUT rams. 256 address tags are stored in LUT ram.

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 using a write barrier when a pointer value is stored to memory.

Organization

Memory is divided into 256-byte card memory pages. Each card has a single byte recording whether a pointer store has taken place in the corresponding memory area. To cover a 512MB memory system 2MB card memory is required at the outermost layer. The outer most 2MB card memory layer is itself divided into 4096 256-byte card pages. Note that each byte represents the pointer store status for a 256B region. The 4096B memory is further resolved to single octa indicating if any pointer store has taken place. Thus, for a 512MB memory system a three-level card memory is used.

Layer	Resolving Power	
0	2 MB	256B regions
1	4 kB	128kB regions
2	8 B	64 MB regions

There is only a single card memory in the system, used by all tasks.

Location

Card memory must be based at physical address zero, extending up to the amount of card memory required. This is so that the address calculation of the memory update may be done with a simple right-shift operation.

Operation

As a program progresses it writes pointer values to memory using the write barrier. Storing a pointer triggers an update to all the layers of card memory corresponding to the main memory location written. A byte 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.

Sample Write Barrier

```
; Milli-code routine for garbage collect write barrier.
; Usable with up to 64-bit memory systems.
; Three level card memory
;
GCWriteBarrier:
        STO     a0,[a1]         ; store the value to memory at a1
        SRL     a1,a1,#8        ; compute card address
        STB     x0,[a1]         ; clear bit in card memory
        SRL     a1,a1,#8        ; repeat for each table level
        STB     x0,[a1]
        SRL     a1,a1,#8
        STB     x0,[a1]
        ;… more stores as needed
        JMP     ra1
```

System Memory Map

There are several components to the system which use tables in memory. These tables are statically allocated at the time the system is built. The table sizes depend on the size of main memory. The card memory table must be located at address zero. So, it is probably best to group the tables together at the low end of memory.

Address	Usage	
$00000000 to $001FFFFF	Card Memory (2 MB)	
$00202000 to $00203FFF	PAM (8kB 2 copies)	
$00280000 to $0029FFFF	Key memory (128 kB)	