Qupls3

Robert Finch

Table of Contents

Overview

Qupls3 is a thirty-two-bit processor.

The processor features 32, 32-bit integer registers and 32, 32-bit floating-point registers.

Motivation

The author desired a CPU core to experiment with cache-line constants and operating systems. He also wanted a core he could develop himself. Complexity is something the author must manage to get the project done and a flat 32-bit design is simple.

Good single thread performance is also a goal.

Having worked on Qupls for two years, the author realized that it did not have very good code density. Having a reasonably good code density is desirable as it is unknown where the CPU will end up. So, Qupls3 arrived and is a mix of the best from previous designs.

The CPU is also designed around the idea of using a simple compiler. Some operations like multiply and divide could have been left out and supported with software generated by a compiler rather than having hardware support. But I was after a simple compiler design. There's lots of room for expansion in the future. It could easily be adapted to a 64-bit design in part anticipating more than 4GB of memory available sometime down the road. A 64-bit architecture is doable in FPGA's today, although it uses two or more times the resources that a 32-bit design would.

History

Qupls3 is a work in progress beginning March 2025. It is a major re-write from earlier versions. Thor which originated from RiSC-16 by Dr. Bruce Jacob. RiSC-16 evolved from the Little Computer (LC-896) developed by Peter Chen at the University of Michigan. The author has tried to be innovative with this design borrowing ideas from many other processing cores.

Features of Qupls3

- Fixed 32-bit length instruction set
- Four way out-of-order superscalar operation
- Four operating modes, machine, hypervisor, supervisor, and user.
- 32-bit data path
- 16 (or more) entry re-order buffer
- Separate fixed-point integer and floating-point register files
- 32 general purpose registers
- Eight condition code registers
- Dedicated loop count register
- Eight branch registers
- Register renaming to remove dependencies.
- Standard suite of ALU operations, add subtract, compare, multiply and divide.
- Arithmetic right shift with rounding.
- Conditional branches with 13 effective displacement bits.
- 1024 Entry Three-way TLB shared between data and code.

Programming Model

Register File – Visible Registers

	Registers				
tag	31	0		tag	31	0
0	zero			32	f0 / zero	
1	A0			33	f1	
…	…			…	…	
8	A7			…		
9	T0			…		
…	…			…		
18	T9			…	…	
19	S0			…	…	
…	…			…	…	
28	S9			…	…	
29	GP			61	…	
30	FP			62	f30	
31	SP			63	f31	

tag	Branch / Link Registers
72	BR0
73	BR1
74	BR2
75	BR3
76	BR4
77	BR5
78	BR6
79	PC

	Condition Registers	
80		CR0
81		CR1
82		CR2
83		CR3
84		CR4
85		CR5
86		CR6
87		CR7

88	LC

Register File – Hidden Registers

tag	Micro-Code Support		tag	
68	MC0		64	User SP
69	MC1		65	Supervisor SP
70	MC2		66	Hypervisor SP

71	MC3			67	Machine SP
89		MLR			
91		MPC		90	CB
	Bound				
	BN0				
	BN1				
	BN2				
	BN3				

Physical Registers

There are 256 general purpose physical registers in the CPU. This provides rename coverage for the 96 logical registers in the design. On average there are 2.6 renamed registers available for every register.

Code Address (Branch) Registers

Many architectures have registers dedicated to addressing code. Almost every modern architecture has a program counter or instruction pointer register to identify the location of instructions. Many architectures also have at least one link register or return address register holding the address of the next instruction after a subroutine call. There are also dedicated branch address registers in some architectures. These are all code addressing registers.

Regno	ABI	Encode	ABI Usage
72	Zero	0	No linkage (read-only)
73	BR1	1	Link register #1
74	BR2	2	Link register #2
75	BR3	3	Link register #3
76	BR4	4	Link register #4
77	BR5	5	Link register #5
78	BR6	6	Link register #6
79	PC	7	Program counter reference (read-only)

It is possible to do an indirect method call using any register.

Condition Registers – CR0 to CR7

Register tags 80 to 87 are reserved for condition results for the compare (CMP) and branch instructions. The low order eight bits of the register typically contain a bit vector representing the results of a comparison operation.

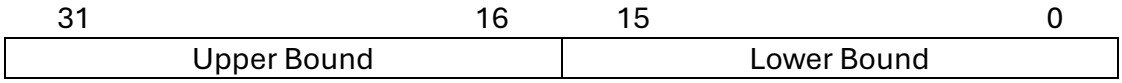
Restricting the CMP and branch instructions to just eight registers conserves opcode bits. Since compares are almost always followed directly by branches, there is not a need for a lot of registers.

7	6	5	4	3	2	1	0
~	OF UN	CA	LE	LT	NOR	NAND	XNOR EQ

Bit		
0	EQ / XNOR	Set if bitwise XNOR of operands is true, equal
1	NAND	Set if logical NAND of operands is true
2	NOR	Set if logical NOR of operands is true
3	LT	Set if less than
4	LE	Set if less than or equal
5	CA	Carry out from operation (addition, subtraction, shift)
6	OF / UN	Overflow status or unordered for floating-point
7		

Bound Registers

The bound registers (M_BOUND) define lower and upper bounds within which a memory reference is restricted for user or applications software. The top sixteen bits of an address are compared against the values in the bounds register. All bound registers are checked in parallel. The address must be greater than or equal to the lower bounds register value and less than the upper bounds register value. Memory access will not be granted unless it is within range of at least one of the bound registers.



There are multiple bound registers to allow a program to occupy different areas of memory. For instance, a program may have separate code, data, and stack areas.

Special Purpose Registers

SR - Status Register (CSR 0x?004)

The processor status register holds bits controlling the overall operation of the processor, state that needs to be saved and restored across interrupts. The bits have individual bit set / clear capability using the CSRRS, CSRRC instructions. Only the user interrupt enable bit is available in user mode, other bits will read as zero.

Bit		Usage
0	uie	User interrupt enable
1	sie	Supervisor interrupt enable
2	hie	Hypervisor interrupt enable
3	mie	Machine interrupt enable
4	die	Debug interrupt enable
5 to 10	ipl	Interrupt level
11	ssm	Single step mode
12	te	Trace enable
13 to 14	om	Operating mode
15 to 16	ps	Pointer size
17	ab	Absolute conditional branches
18	dbg	Debug mode
19	mprv	memory privilege
20 to 22	Swstk	Software stack
23		reserved
24 to 31	cpl	Current privilege level

CPL is the current privilege level the processor is operating at.

T indicates that trace mode is active.

OM processor operating mode.

PS: indicates the size of pointers in use. This may be one of 32, 64 or 128 bits.

AB: indicates that conditional branches should use absolute(1) or relative (0) addressing.

AR: Address Range indicates the number of address bits in use. 0 = near or short (32-bit) addressing is in use. When short addressing is in use only the low order 32-bit are significant and stored or loaded to or from the stack.

IPL is the interrupt mask level

MPRV Memory Privilege, indicates to use previous operating mode for memory privileges

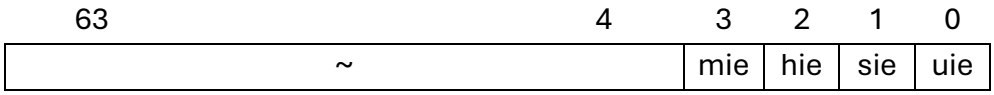
SC - Stack Canary (GPR nn)

This special purpose register is available in the general register file as register nn. The stack canary register is used to alleviate issues resulting from buffer overflows on the stack. The canary register contains a random value which remains consistent throughout the run-time of a program. In the right conditions, the canary register is written to the stack during the function's prolog code. In the function's epilog code, the value of the canary on stack is checked to ensure it is correct, if not a check exception occurs.

[U/S/H/M]_IE (0x?004)

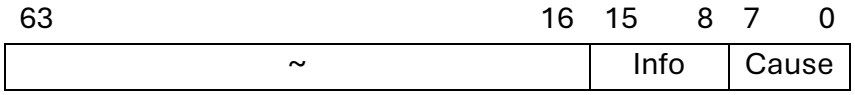
See status register.

This register contains interrupt enable bits. The register is present at all operating levels. Only enable bits at the current operating level or lower are visible and may be set or cleared. Other bits will read as zero and ignore writes. Only the lower four bits of this register are implemented. The bits have individual bit set / clear capability using the CSRRS, CSRRC instructions.



[U/S/H/M]_CAUSE (CSR- 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. The info field, filled in by hardware, may supply additional information related to the exception.



[U/S/H/M]_SCRATCH – CSR 0x?041

This is a scratchpad register. Useful when processing exceptions. There is a separate scratch register for each operating mode.

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. Only the low order sixteen bits of the register are implemented.

S_KEYS (CSR 0x1020 to 0x1027)

These eight registers contain the collection of keys associated with the process for the memory lot system. Each key is twenty-four bits in size. All eight 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.

			23 0
1020			key0
1021			key1
…			…
1027			key7

M_CORENO (CSR 0x3001)

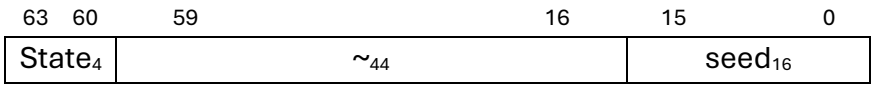
This register contains a number that is externally supplied on the coreno_i input bus to represent the hardware thread id or the core number. It should be non-zero.

M_TICK (CSR 0x3002)

This register contains a tick count of the number of clock cycles that have passed since the last reset. Note that this register should not be used for precise timing as the processor's clock frequency may vary for performance and power reasons. The TIME CSR may be used for wall-clock timing as it has its own timing source.

M_SEED (CSR 0x3003)

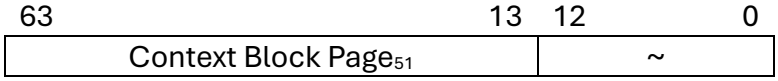
This register contains a random seed value based on an external entropy collector. The most significant bit of the state is a busy bit.



State$_4$ Bit	
0	dead
1	test
2	valid, the seed value is valid
3	Busy, the collector is busy collecting a new seed value

M_CBA (CSR 0x3005)

This register contains the address of the currently active context block, in which the CPU's context will be saved on a context switch. Context switching may be done with the exchange jump XJMP instruction. Only the upper 51-bits may be set; the lower 13 bits of the address are always zero.



M_BADADDR (CSR 0x3007)

This register contains the 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 EIP register.

M_BAD_INSTR (CSR 0x300B)

This register contains a copy of the exceptioned instruction.

M_SEMA (CSR 0x300C)

This register contains semaphores. The semaphores are shared between all cores in the MPU.

M_BOUND (CSR 0x3010 to CSR 0x3013)

This set of four registers establishes boundaries within which a program may operate. Each register is split into two pieces; one for the lower bound and one for the upper bound. Only the top sixteen bits of the address (the memory page number) are compared to the bounds. A program address (instruction or data) which is out of bounds will cause a bounds exception.

		Upper Bound	Lower Bound
0x3010	Code	31 16	15 0
0x3011	Data		
0x3012	Stack		
0x3013	Data		

M_TVEC – CSR 0x3030 to 0x3034

These registers contain the address of the exception handler table for a given operating mode. TVEC[0] to TVEC[2] are used by the REX instruction.

A sync instruction should be used after modifying one of these registers to ensure the update is valid before continuing program execution.

Reg #	
0x3030	TVEC[0] – user mode
0x3031	TVEC[1] - supervisor mode
0x3032	TVEC[2] – hypervisor mode
0x3033	TVEC[3] – machine mode
0x3034	TVEC[4] - debug

M_SR_STACK (CSR 0x3080 to CSR 0x3087)

This set of registers contains a stack of the status register which is pushed during exception processing and popped on return from interrupt. There are only eight slots as that is the maximum nesting depth for interrupts.

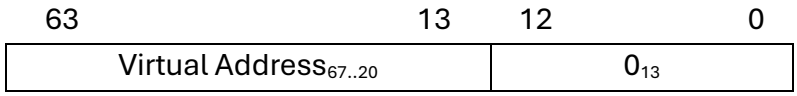
M_MC_STACK (CSR 0x3090 to CSR 0x3097)

This set of registers is a stack for the micro-code instruction register (MCIR) and the micro-code instruction pointer (MCIP). MCIR and MCIP need to be retained through exception processing.

Bits 52 to 63 of the register contain the MCIP. Bits 0 to 51 contain the MCIR.

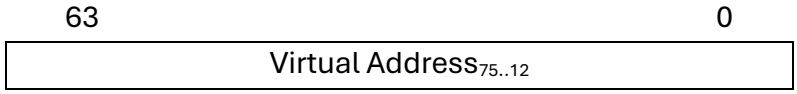
M_IOS – IO Select Register (CSR 0x3100)

The location of IO is determined by the contents of the IOS control register. The select is for a 1MB region. This address is a virtual address. The low order 13 bits of this register should be zero and are ignored.



M_CFGS – Configuration Space Register (CSR 0x3101)

The location of configuration space is determined by the contents of the CFGS control register. The select is for a 256MB region. This address is a virtual address. The low order 12 bits of this address are assumed to be zero. The default value of this registers is $FF...FD000



M_EPC (CSR 0x3108 to 0x310F)

This set of registers contains the address stack for the program counter used in exception handling.

Reg #	Name
0x3108	EPC0
...	

0x310F	EPC7

Operating Modes

The core operates in one of four basic modes: application/user mode, supervisor mode, hypervisor mode or machine mode. Machine mode is switched to when an interrupt or exception occurs, or when debugging is triggered. On power-up the core is running in machine mode. An RFI instruction must be executed to leave machine mode after power-up.

Most modern OSs require at least two modes of operation, a user mode, and a more secure system mode. It can be advantageous to have more operating modes as it eases the software implementation when dealing with multiple operating systems running on the same machine at the same time.

A subset of instructions is limited to machine mode.

Mode Bits	Mode
0	User / App
1	Supervisor
2	Hypervisor
3	Machine / Debug

Each operating mode has its own vector table. Different sets of CSR registers are visible to each operating mode.

Exceptions

External Interrupts

There is little difference between an externally generated exception and an internally generated one. An externally caused exception will set the exception cause code for the currently fetched instruction. A hardware interrupt displaces the instruction at the point the interrupt occurred with a special TRAP instruction.

There are sixty-four priority interrupt levels for external interrupts. When an external interrupt occurs the mask level is set to the level of the current interrupt. A subsequent interrupt must exceed the mask level to be recognized.

Effect on Machine Status

The operating mode is always switched to machine mode on exception. It is up to the machine 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. Machine mode code must enable interrupts at some point.

Exception Stack

The status register and instruction pointer are quickly pushed onto an internal stack when an exception occurs. This stack is at least 8 entries deep to allow for nested interrupts and multiply nested traps and exceptions. The stack pointer is also switched to one corresponding to the machine's operating mode. A hardware interrupt will also cause the stack pointer to change to one specific to the interrupt level.

Vector Table

The machine mode kernel vector is always used to locate the exception routine. The exception routine may then redirect the exception to a lower operating mode using the REX instruction. When an exception occurs the CPU just jumps to the entry in the vector table. The entry should contain a branch instruction to the exception handler.

Cause Code	Usage
0	Debug Breakpoint (BRK)
1	Debug breakpoint – single step
2	Bus Error
3	Address Error

4	Unimplemented Instruction
5	Privilege Violation
6	Page fault
7	Instruction trace
8	Stack Canary
9	Abort
10	Interrupt
11	Non-maskable interrupt
12	Reset
13	Alternate Cause > 31
14, 15	Reserved
16	User / App Environment call
17	Supervisor Environment call
18	Hypervisor environment call
19	Machine environment call
20	Trap
21	Bound
22 to 31	reserved

Applications Usage

32 to 63	reserved
64	Divide by zero
65	Overflow
66	Table Limit
67 to 251	Unassigned usage
252	Reset value of stack pointer
253	Reset value of instruction pointer
254, 255	Reserved

Breakpoint Fault (0)

The breakpoint instruction, 0, was encountered.

Single Step Breakpoint (1)

This fault is performed at the end of a single step operation. Single stepping is turned off so that a debugger may begin processing.

Bus Error Fault (2)

The bus error fault is performed if the bus error signal was active during the bus transaction. This could be due to a bad or missing device.

Address Error (3)

This fault will occur if an instruction address does not have the two LSBs equal to zero.

Unimplemented Instruction Fault (4)

An unimplemented instruction causes this fault.

Page Fault (6)

The page table walker was unable to find a valid translation for the virtual address.

Instruction Trace Fault (7)

An instruction trace was triggered. This fault requires the Trace module to be present.

Stack Canary Fault (8)

This fault is caused if the stack canary was overwritten. A load instruction using the canary register did not match the value in the canary register.

Abort (9)

The external abort input signal was asserted.

Interrupt (10)

The external interrupt signal was asserted, and the interrupt level was greater than the current mask level.

Reset Vector (12)

This vector is the address that the processor begins running at.

Alternate Cause (13)

The alternate cause vector is jumped to if the cause code is greater than 31.

User / App Environment Call (16)

This fault is triggered when a system call instruction is executed while in User / App mode.

Supervisor Environment Call (17)

This fault is triggered when a system call instruction is executed while in Supervisor mode.

Hypervisor Environment Call (18)

This fault is triggered when a system call instruction is executed while in Hypervisor mode.

Machine Environment Call (19)

This fault is triggered when a system call instruction is executed while in Machine mode.

TRAP (20)

This fault is triggered by the TRAP instruction when the trap condition is met.

Bound (21)

This fault is triggered if an address is out of bounds as set by the bound registers.

Reset

Reset is treated as an exception. The reset routine should exit using an RFI instruction. The status register should be setup appropriately for the return.

The core begins executing instructions at the address defined by the reset vector in the exception table. At reset the exception table is set to the last 256 bytes of memory $FF...FFC00. All registers are in an undefined state.

Precision

Exceptions in Qupls3 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 pipeline 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.

Instruction Set

Overview

Qupls3 is a fixed length instruction set with lengths of 32-bits. There are several different classes of instructions including arithmetic, memory operate, branch, floating-point and others.

Code Alignment

Program code may be relocated at any tetra-byte (4 byte) address. However, within a subroutine code should be contiguous.

Root Opcode

The root opcode determines the class of instructions executed. Some commonly executed instructions are also encoded at the root level to make more bits available for the instruction. The root opcode is always present in all instructions as bits zero to five of the instruction.

						▼
L	LX_2	$Immediate_{12}$	Cr	$Rs1_5$	Rd_5	4_6

Destination Register Spec

Most instructions have a destination register. The register spec for the destination register is always in the same position, bits 6 to 10 of an instruction.

					▼	
L	LX_2	$Immediate_{12}$	Cr	$Rs1_5$	Rd_5	4_6

Source Register Spec

Most instructions have at least one source register. There may be as many a three source register specs. Please refer to individual instruction descriptions for the location of the source register specification fields.

				▼		
L	LX_2	$Immediate_{12}$	Cr	$Rs1_5$	Rd_5	4_6

Constant Field Spec

Many instructions have constants associated with them. Constants may be embedded directly in the instruction, or they may occupy instruction words on the instruction cache line. Most instructions follow the same template for constants.

▼	▼	▼				
L	LX_2	$Immediate_{12}$	Cr	$Rs1_5$	Rd_5	4_6

Format for Reference to Cache Line:

▼	▼	▼						
1	1_2	$\sim_7$	$Offset_4$	0	Cr	$Rs1_5$	Rd_5	4_6

Table of Constant Location Bits – L, LX_2

L	LX_2	Location
0	?	Value is constant encoded directly in instruction, LX_2 is top two bits of the constant field in the instruction or additional opcode bits
1	0	Value comes from register Rs2
1	1	Value is 32-bit constant on the cache line
1	2	Value is 64-bit constant on the cache line
1	3	reserved

CL_3 is a tetra index into the cache line locating the constant; it is an offset from the address of the instruction. Constants may be placed only in the last half of a cache line. Instructions must occupy the first half.

Instruction Format Tables

Compare Instruction Format

	31	30..29	28 ... 17	16	15 ... 11	10..9	8 ... 6	5 ... 0
CMP	L	LX_2	$Immediate_{12}$	0	$Rs1_5$	0	CRd_3	3_6
CMP	1	0_2	$Rs2_5$	0	$Rs1_5$	0	CRd_3	3_6
CMPA	L	LX_2	$Immediate_{12}$	0	$Rs1_5$	1	CRd_3	3_6
CMPA	1	0_2	$Rs2_5$	0	$Rs1_5$	1	CRd_3	3_6
FCMPS	L	LX_2	$Immediate_{12}$	0	$FRs1_5$	2	CRd_3	3_6
FCMPS	1	0_2	$FRs2_5$	0	$FRs1_5$	2	CRd_3	3_6
FCMPD	L	LX_2	$Immediate_{12}$	1	$FRs1_5$	2	CRd_3	3_6
FCMPD	1	0_2	$FRs2_5$	1	$FRs1_5$	2	CRd_3	3_6

Branch Instruction Formats

	31	30..29	28 ... 16	15 ... 11	10 ... 6	5 ... 0	
B[L]	0	LX_2	$Displacement_{22...3}$		BRd_3	13_5	D
BLR[L]	1	0_2	BRs_3 $Immediate_{19...3}$		BRd_3	13_5	D
[Dcc]Bcc[L]	0	D_{12}	BRs_3 Cnd_4 CRs_6 $Disp_{10...3}$		BRd_3	12_5	D
[Dcc]Bcc[L]	1	0_2	BRs_3 Cnd_4 CRs_6 ~ $Rs1_5$	$\sim_2$	BRd_3	12_5	~
[Dcc]Bcc[L]	1	1_2	BRs_3 Cnd_4 CRs_6 $\sim_4$ CL_3 ~		BRd_3	12_5	~

Load and Store Instruction Formats

Load	L	LX_2	$Displacement_{11...0}$	Cr	$Rs1_5$	Rd_5		$Opcode_5$
Indexed Ld	1	LX_2	$Disp_5$ Sc_2 $Rs2_5$	Cr	$Rs1_5$	Rd_5		$Opcode_5$
Store	L	LX_2	$Displacement_{11...0}$	U	$Rs1_5$	$Rs2_5$		$Opcode_5$
Indexed St	1	LX_2	$Disp_5$ Sc_2 $Rs2_5$	U	$Rs1_5$	$Rs3_5$		$Opcode_5$
Store	L	LX_2	$Displacement_{11...0}$	U	$Rs1_5$	~	CL_3	$Opcode_5$
Indexed St	1	LX_2	$Disp_5$ Sc_2 $Rs2_5$	U	$Rs1_5$	~	CL_3	$Opcode_5$

ALU Instruction Formats

	31	30..29	28 ... 17	16	15 ... 11	10 ... 6	5 ... 0
ADD	L	LX_2	$Immediate_{12}$	Cr	$Rs1_5$	Rd_5	4_6
ADD	1	LX_2	0_4 $\sim_3$ $Rs2_5$	Cr	$Rs1_5$	Rd_5	4_6
ADC	1	LX_2	1_4 $\sim_3$ $Rs2_5$	Cr	$Rs1_5$	Rd_5	4_6
ABS	1	LX_2	2_4 $\sim_3$ $Rs2_5$	Cr	$Rs1_5$	Rd_5	4_6
CNTLO	1	LX_2	3_4 $\sim_3$ $\sim_5$	Cr	$Rs1_5$	Rd_5	4_6
CNTLZ	1	LX_2	4_4 $\sim_3$ $\sim_5$	Cr	$Rs1_5$	Rd_5	4_6
CNTPOP	1	LX_2	5_4 $\sim_3$ $\sim_5$	Cr	$Rs1_5$	Rd_5	4_6
CNTTZ	1	LX_2	6_4 $\sim_3$ $\sim_5$	Cr	$Rs1_5$	Rd_5	4_6
ADB	L	LX_2	$Immediate_{12}$	Cr	$\sim_2$ BR_3	Rd_5	5_6
ADB	1	LX_2	$\sim_7$ $Rs2_5$	Cr	$\sim_2$ BR_3	Rd_5	5_6
MULA	L	LX_2	$Immediate_{12}$	Cr	$Rs1_5$	Rd_5	6_6
MULA	1	0_2	0_3 $\sim_4$ $Rs2_5$	Cr	$Rs1_5$	Rd_5	6_6
MUL	1	0_2	1_3 $\sim_4$ $Rs2_5$	Cr	$Rs1_5$	Rd_5	6_6

MULSA	1	0_2	2_3	$\sim_4$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	6_6
MULH	1	0_2	4_3	$\sim_4$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	6_6
AND	L	LX_2	$Immediate_{12}$			Cr	$Rs1_5$	Rd_5	8_6
AND	1	LX_2	0_3	$\sim_4$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	8_6
OR	L	LX_2	$Immediate_{12}$			Cr	$Rs1_5$	Rd_5	9_6
OR	1	LX_2	0_3	$\sim_4$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	9_6
XOR	L	LX_2	$Immediate_{12}$			Cr	$Rs1_5$	Rd_5	10_6
XOR	1	LX_2	0_3	$\sim_4$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	10_6
SUBF	L	LX_2	$Immediate_{12}$			Cr	$Rs1_5$	Rd_5	12_6
SUBF	1	LX_2	0_3	$\sim_4$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	12_6
SBC	1	LX_2	1_3	$\sim_4$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	12_6
PTRDIF	1	LX_2	2_3	Ui_4	$Rs2_5$	Cr	$Rs1_5$	Rd_5	12_6
DIVA	L	LX_2	$Immediate_{12}$			Cr	$Rs1_5$	Rd_5	14_6
DIVA	1	LX_2	0_3	$\sim_4$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	14_6
DIV	1	LX_2	1_3	$\sim_4$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	14_6
DIVSU	1	LX_2	2_3	$\sim_4$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	14_6
SQRT	1	LX_2	3_3	$\sim_4$	$\sim_5$	Cr	$Rs1_5$	Rd_5	14_6
MODA	1	LX_2	4_3	$\sim_4$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	14_6
MOD	1	LX_2	5_3	$\sim_4$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	14_6
LOADA	L	LX_2	$Displacement_{11...0}$			Cr	$Rs1_5$	Rd_5	39_5
LOADA	1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	39_5

	L	LX_2	$Immediate_{12}$	Cr	$Rs1_5$	Rd_5	$Opcode_6$
	1	LX_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	$Opcode_6$

Shift Instruction Formats

	31	3029	28 17	16	15 11	10 6	5 0
SLL	0	0_2	0_3 H ~ $Shamt_6$	Cr	$Rs1_5$	Rd_5	2_6
SLL	1	0_2	0_3 H $\sim_3$ $Rs2_5$	Cr	$Rs1_5$	Rd_5	2_6
SRL	0	0_2	1_3 H ~ $Shamt_6$	Cr	$Rs1_5$	Rd_5	2_6
SRL	1	0_2	1_3 H $\sim_3$ $Rs2_5$	Cr	$Rs1_5$	Rd_5	2_6
SRA	0	0_2	2_3 Rm_2 $Shamt_6$	Cr	$Rs1_5$	Rd_5	2_6
SRA	1	0_2	2_3 Rm_2 $Rs2_5$	Cr	$Rs1_5$	Rd_5	2_6
RO[L|R]	0	0_2	4_3 L ~ $Shamt_6$	Cr	$Rs1_5$	Rd_5	2_6
RO[L|R]	1	0_2	4_3 L $\sim_3$ $Rs2_5$	Cr	$Rs1_5$	Rd_5	2_6
EXT	0	3_2	Me_6 Mb_6	Cr	$Rs1_5$	Rd_5	2_6
EXT[Z]	1	0_2	3_3 z ~ $Rs2_5$	Cr	$Rs1_5$	Rd_5	2_6

CSR Instruction Formats

	31	3029	28 17	16	1514	15 11	10 6	5 0
CSRxx	0	Op_2	$CSRno_{12}$	Cr	$Rs1_5$		Rd_5	7_6
	1	0_2	Op_2 $\sim_5$ $Rs2_5$	Cr	$Rs1_5$		Rd_5	7_6
32-bit data	1	1_2	$CSRno_{12}$	Cr	Op_2	CL_3	Rd_5	7_6

BRK / SYS Instruction Formats

	31	3029	28 17	16	15 11	10 6	5 0
BRK	0	0_2	0_{12}	0	0_5	0_5	0_6
SYS	0	0_2	1_{12}	0	0_5	0_5	0_6
RFI	0	0_2	2_{12}	0	0_5	0_5	0_6
RFI2	0	0_2	3_{12}	0	0_5	0_5	0_6
	1	LX_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	$Opcode_6$

Macro Instruction Formats

	31	3029	28 17	16	1514	15 11	10 6	5 0
ENTER	0	0_2	$Immediate_{12}$	0	$\sim_5$		~ Ns_4	31_6
LEAVE	0	1_2	$Immediate_{12}$	0	$Offs_{6..2}$		~ Nr_4	31_6
PUSHA	0	0_2	$\sim_{12}$	1	$\sim_5$		~ $\sim_4$	31_6
POPA	0	1_2	$\sim_{12}$	1	$\sim_5$		~ $\sim_4$	31_6
PUSH	1	0_2	N_2 $Rs4_5$ $Rs3_5$	0	$Rs2_5$		$Rs1_5$	31_6
POP	1	0_2	N_2 $Rd4_5$ $Rd3_5$	1	$Rd2_5$		$Rd1_5$	31_6
XJMP	0	$Displacement_{26...13}$		0	$Rs1_5$		$Disp_{31...27}$	30_6
XJMP	1	0_2	$Disp_{17...13}$ 0_2 $Rs2_5$	1	$Rs1_5$		$Disp_{22...18}$	30_6

MOV Instruction Format

	31	3028	27			17	16	15 11	10 6	5 0
MOV	0	0_3	$\sim_9$		$Rs1_{65}$	Rd_{65}	Cr	$Rs1_{4...0}$	$Rd_{4...0}$	$Opcode_6$
MOVA	0	1_3	$\sim_9$		$Rs1_{65}$	Rd_{65}	Cr	$Rs1_{4...0}$	$Rd_{4...0}$	$Opcode_6$
MOVSX	0	2_3	$\sim_3$	$Uimm_6$	$Rs1_{65}$	Rd_{65}	Cr	$Rs1_{4...0}$	$Rd_{4...0}$	$Opcode_6$
MOVZX	0	3_3	$\sim_3$	$Uimm_6$	$Rs1_{65}$	Rd_{65}	Cr	$Rs1_{4...0}$	$Rd_{4...0}$	$Opcode_6$

Exception Triggering Instruction Formats

	31	3029	28 17		16	15 11	10 6	5 0
TRAP	1	0_2	$Immediate_{12}$		~	$Rs1_5$	$Cond_5$	28_6
TRAP	0	0_2	$\sim_7$	$Rs2_5$	~	$Rs1_5$	$Cond_5$	28_6
CHK	0	Op_4	$Rs3_5$	$Rs2_5$	~	$Rs1_5$	$Offs_5$	29_6

Instruction Pres/Postfixes and Modifiers Instruction Formats

	31	3029	28 17		16	15 11	10 6	5 0
ATOM	1	0_2	$Mask_{12}$		~	$\sim_5$	IPL_5	60_6
QEXT	0	0_2	$\sim_7$	$Rs2_5$	~	$Rs1_5$	Rd_5	60_6
PRED	0	1_2	$Mask_{15...4}$			$Rs1_5$	~ $Mask_{3...0}$	60_6
PFX	0	LX_2	$Immediate_{25...5}$				Wh_2	61_6

Condition Register Manipulation Instruction Formats

	31	3029	28 18			17 12	11 6	5 0
CRAND	0	0_2	0_4	$\sim_6$	I	$CRs1_6$	CRd_6	11_6
CROR	0	0_2	1_4	$\sim_6$	I	$CRs1_6$	CRd_6	11_6
CRXOR	0	0_2	2_4	$\sim_6$	I	$CRs1_6$	CRd_6	11_6
CRANDC	0	0_2	3_4	$\sim_6$	I	$CRs1_6$	CRd_6	11_6
CRAND	1	0_2	0_4	~ $CRs2_6$		$CRs1_6$	CRd_6	11_6
CROR	1	0_2	1_4	~ $CRs2_6$		$CRs1_6$	CRd_6	11_6
CRXOR	1	0_2	2_4	~ $CRs2_6$		$CRs1_6$	CRd_6	11_6
CRANDC	1	0_2	3_4	~ $CRs2_6$		$CRs1_6$	CRd_6	11_6
CRNAND	1	0_2	4_4	~ $CRs2_6$		$CRs1_6$	CRd_6	11_6
CRNOR	1	0_2	5_4	~ $CRs2_6$		$CRs1_6$	CRd_6	11_6
CRXNOR	1	0_2	6_4	~ $CRs2_6$		$CRs1_6$	CRd_6	11_6
CRORC	1	0_2	7_4	~ $CRs2_6$		$CRs1_6$	CRd_6	11_6

Table of Root Opcodes

	x000	x001	xx010	x011	x100	x101	x110	x111
000x	0 BRK	1 Custom	2 {SHIFT}	3 CMP CMPA	4 ADD	5 APC	6 MUL	7 CSR
001x	8 AND	9 OR	10 XOR	11 {CR}	12 SUBF	13	14 DIV	15 MOV
010x	16	17	18	19	20	21	22	23
011x	24 Bc[L] DBc[L]	25 Bc[L] DBc[L]	26 B[L] BLR[L]	27 B[L] BLR[L]	28 TRAP	29 CHK	30 XJMP FORK	31 ENTER LEAVE PUSH POP
100x	32 LDB	33 LDBZ	34 LDW	35 LDWZ	36 LDT	37 LDTZ	38 LOAD	39 LOADA
101x	40 STB	41 STBI	42 STW	43 STWI	44 STT	45 STTI	46 STORE	47 STOREI
110x	48 LDFS	49 LDFD	50 LDFQ	51 Fence Misc Mem	52 STPTR	53 {BLOCK}	54 {Float}	55
111x	56 STFS	57 STFD	58 STFQ	59 AMO	60 ATOM QEXT PRED	61 PFX	62	63 NOP

Instruction Descriptions

ABS[.] – Absolute Value

Description:

This instruction computes the absolute value of the sum two source operands in registers Rs1 and Rs2 and places the result in Rd. Condition register CR0 may be updated if the Cr bit of the instruction is set.

Instruction Format: R1

	31	3029	28		17	16	15	11	10	6	5	0
ABS	1	0_2	2_4	$\sim_3$	$Rs2_5$	Cr	$Rs1_5$		Rd_5		4_6	

Operation:

```
If (Rs1 + Rs2) < 0
   Rt = -(Rs1 + Rs2)
else
   Rt = (Rs1 + Rs2)
```

Execution Units: Integer ALU #0 only

Clock Cycles: 1

Exceptions: none

Notes:

ADD[.] – Add

Description:

Add two source registers Rs1 and Rs2 or Rs1 and a constant and place the sum in the destination register Rd. All register values are integers. Condition register CR0 may be updated if the Cr bit of the instruction is set.

Instruction Format: R3

	31	3029	28 17			16	15 11	10 6	5 0
ADD	L	LX_2	$Immediate_{12}$			Cr	$Rs1_5$	Rd_5	4_6
ADD	1	0_2	0_4	$\sim_3$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	4_6

Operation:

Rd = Rs1 + Rs2

OR

Rd = Constant + Rs1

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

ADB[.] - Add Immediate to Branch Register

Description:

Add an immediate value to the branch register and place the result in a destination register Rd. This instruction may be used in the formation of program counter relative addresses.

Instruction Format: RI

	31	3029	28 17	16	15 11	10 6	5 0
ADB	L	LX_2	$Immediate_{12}$	Cr	$\sim_2$	Rd_5	5_6
ADB	1	LX_2	$Rs2_5$	Cr	$\sim_2$	Rd_5	5_6

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rd = BR + immediate

Exceptions:

Notes:

ADC[.] – Add with Carry

Description:

Add two source registers Rs1 and Rs2 or Rs1 and a constant and the carry flag and place the sum in the destination register Rd. All register values are integers. Condition register CR0 may be updated if the Cr bit of the instruction is set.

Instruction Format: R3

	31	3029	28		17	16	15 11	10 6	5 0
ADC	1	0_2	1_4	$\sim_3$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	4_6

Operation:

Rd = Rs1 + Rs2 + carry

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

CNTLO[.] – Count Leading Ones

Description:

This instruction counts the number of consecutive one bits beginning at the most significant bit towards the least significant bit for the register Rs1 and places the count in register Rd.

Instruction Format: R3

	31	3029	28		17	16	15 11	10 6	5 0
CNTLO	1	LX_2	3_4	$\sim_3$	$\sim_5$	Cr	$Rs1_5$	Rd_5	4_6

Operation:

Execution Units: Integer ALU #0 only

Clock Cycles: 1

Exceptions: none

Notes:

CNTLZ[.] – Count Leading Zeros

Description:

This instruction counts the number of consecutive zero bits beginning at the most significant bit towards the least significant bit for the register Rs1 and places the count in register Rd.

Instruction Format: R3

	31	3029	28		17	16	15 11	10 6	5 0
CNTLZ	1	LX_2	4_4	$\sim_3$	$\sim_5$	Cr	$Rs1_5$	Rd_5	4_6

Operation:

Execution Units: Integer ALU #0 only

Clock Cycles: 1

Exceptions: none

Notes:

CNTPOP[.] – Count Population

Description:

This instruction counts the number of bits set in source register Rs1 and places the count in destination register Rd.

Instruction Format:

	31	3029	28		17	16	15 11	10 6	5 0
CNTPOP	1	LX_2	5_4	$\sim_3$	$\sim_5$	Cr	$Rs1_5$	Rd_5	4_6

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

CNTTZ[.] – Count Trailing Zeros

Description:

This instruction counts the number of consecutive zero bits beginning at the least significant bit towards the most significant bit of the value in register Rs1 and places the count in register Rd. This instruction can also be used to get the position of the first one bit from the right-hand side.

Instruction Format: R3

	31	3029	28		17	16	15 11	10 6	5 0
CNTTZ	1	LX_2	6_4	$\sim_3$	$\sim_5$	Cr	$Rs1_5$	Rd_5	4_6

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

CSR[.] – Control and Special Registers Operations

Description:

Perform an operation on a CSR specified either as a constant in the instruction or as a number in source register Rs2. The previous value of the CSR is placed in the destination register Rd. New values for the CSR may come from either the value in Rs1 or an immediate constant.

Operation	**Op_2**	**Mnemonic**
Read CSR	0	CSRRD
Write CSR	1	CSRRW
Or to CSR (set bits)	2	CSRRS
And complement to CSR (clear bits)	3	CSRRC

Supported Operand Sizes: N/A

Instruction Formats:

	31	30..29	28..17	16	15..14	15..11	10..6	5..0
CSRxx	0	Op_2	$CSRno_{12}$	Cr	$Rs1_5$		Rd_5	7_6
	1	0_2	Op_2 | $\sim_5$ | $Rs2_5$	Cr	$Rs1_5$		Rd_5	7_6
32-bit data	1	1_2	$CSRno_{12}$	Cr	Op_2	CL_3	Rd_5	7_6

Notes:

The top two bits of the Regno field correspond to the operating mode.

LOADA[.] – Load Address

Description:

This instruction computes the virtual address following the same format as a load or store instruction and places it in the destination register Rd.

Instruction Format:

	31	3029	28		17	16	15 11	10 6	5 0
LOADA	L	LX_2	$Displacement_{11...0}$			Cr	$Rs1_5$	Rd_5	39_5
LOADA	1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	39_5

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rd = Rs1 + Displacement

OR

Rd = Rs1 + Rs2 *Scale + displacement

Exceptions:

Notes:

SBC[.] – Subtract with Carry

Description:

Subtract two source registers Rs1 and Rs2 or Rs1 and a constant and the carry flag and place the difference in the destination register Rd. All register values are integers. Condition register CR0 may be updated if the Cr bit of the instruction is set.

Instruction Format: R3

	31	3029	28		17	16	15 11	10 6	5 0
SBC	1	0_2	1_4	$\sim_3$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	12_6

Operation:

Rd = Rs2 – Rs1 - carry

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

SUBF[.] – Subtract From

Description:

Subtract two source registers Rs1 and Rs2 or Rs1 and a constant and place the difference in the destination register Rd. All register values are integers. Condition register CR0 may be updated if the Cr bit of the instruction is set.

Instruction Format: R3

	31	3029	28 17			16	15 11	10 6	5 0
SUBF	L	LX_2	$Immediate_{12}$			Cr	$Rs1_5$	Rd_5	12_6
SUBF	1	0_2	0_4	$\sim_3$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	12_6

Operation:

Rd = Rs2 – Rs1

OR

Rd = Constant - Rs1

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

Multiply and Divide

DIV[.] – Signed Division

Description:

Divide source dividend operand in Rs1 by divisor operand in Rs2 and place the quotient in the destination register Rd. All registers are integer registers. Arithmetic is signed twos-complement values.

Instruction Format:

	31	3029	28		17	16	15	11	10	6	5	0
DIV	1	LX_2	1_3		$Rs2_5$	Cr	$Rs1_5$		Rd_5		14_6	

Operation:

Rt = Ra / Rb

Execution Units: ALU #0 Only

Exceptions: DBZ

Notes:

DIVA[.] – Address Division

Description:

Divide source dividend operand in Rs1 by divisor operand in either Rs2 or an immediate constant and place the quotient in the destination register Rd. All registers are integer registers. Arithmetic is unsigned twos-complement values. DIVA may be used in pointer to index conversions.

Instruction Format:

	31	3029	28 17	16	15 11	10 6	5 0
DIVA	L	LX_2	$Immediate_{12}$	Cr	$Rs1_5$	Rd_5	14_6
DIVA	1	LX_2	0_3	$Rs2_5$	Cr	$Rs1_5$	Rd_5	14_6

Operation:

Rd = Rs1 / Rs2

OR

Rd = Rs1 / Constant

Execution Units: ALU #0 Only

Exceptions: none

Notes:

MUL[.] – Multiply

Description:

Multiply two source registers Rs1 and Rs2 and place the product in the destination register Rd. All registers are integer registers. Values are treated as signed integers.

Instruction Format: R3

	31	3029	28 17		16	15 11	10 6	5 0
MUL	1	0_2	1_3	$Rs2_5$	Cr	$Rs1_5$	Rd_5	6_6

Operation: R2

Rd = Rs1 * Rs2

Clock Cycles: 4

Execution Units: All Integer ALUs

Exceptions: none

Notes:

MULA[.] – Multiply for Addressing

Description:

Multiply two source registers Rs1 and Rs2 or Rs1 and an immediate constant, and place the product in the destination register Rd. All registers are integer registers. Values are treated as unsigned integers. This instruction is typically used in address calculations for arrays.

Instruction Format: R3

	31	3029	28 17	16	15 11	10 6	5 0
MULA	L	LX_2	$Immediate_{12}$	Cr	$Rs1_5$	Rd_5	6_6
MULA	1	0_2	0_3 $Rs2_5$	Cr	$Rs1_5$	Rd_5	6_6

Operation: R2

Rd = Rs1 * Rs2

OR

Rd = Rs1 * Constant

Clock Cycles: 4

Execution Units: All Integer ALUs

Exceptions: none

Notes:

Shift and Rotate

SLL[.] –Shift Left Logical

Description:

Left shift a source operand in Rs1 by a source operand value ins Rs2 or a constant, and place the result in the destination register Rd. The second source operand may be either a register specified by the Rs2 field of the instruction, or an immediate value. If the 'H' bit is set, the upper 64-bits of the result are transferred to the destination register, Rd. Condition register CR0 may be updated if the Cr bit of the instruction is set. Also, the carry bit of CR0 will be set if any bit shifted out from the high order bits is non-zero, otherwise it will be cleared.

Instruction Format: SHIFT

	31	3029	28 17	16	15 11	10 6	5 0
SLL	0	0_2	0_3 H ~ $Shamt_6$	Cr	$Rs1_5$	Rd_5	2_6
SLL	1	0_2	0_3 H $\sim_3$ $Rs2_5$	Cr	$Rs1_5$	Rd_5	2_6

Operation:

Rd = Rs1 << Rs2

OR

Rd = Rs1 << constant

Operation Size:

Execution Units: integer ALU

Exceptions: none

Notes:

Left shift instructions are faster than multiply.

Example:

SRA[.] –Shift Right Arithmetic

Description:

Right shift a source operand value in Rs1 by a source operand value in Rs2 or a constant and place the sign extended result in the destination register. The result may be rounded.

Instruction Format: SHIFT

	31	3029	28 17	16	15 11	10 6	5 0
SRA	0	0_2	2_3 Rm_2 $Shamt_6$	Cr	$Rs1_5$	Rd_5	2_6
SRA	1	0_2	2_3 Rm_2 $Rs2_5$	Cr	$Rs1_5$	Rd_5	2_6

Rm_2	
0	Truncate
1	Round towards zero, If the result is negative, then it is rounded up.
2	Round up, one is added to the result if there was a carry out of the LSB.
3	reserved

Operation:

Rt = Ra >> Rc

Operation Size:

Execution Units: integer ALU

Exceptions: none

Example:

SRL[.] –Shift Right Logical

Description:

Right shift a source operand value in Rs1 by a second source operand value in Rs2 or a constant and place the result in the destination register. If the 'H' bit is set, the lower 64-bits of the result are transferred to the destination register, Rd. Condition register CR0 may be updated if the Cr bit of the instruction is set. Also, the carry bit of CR0 will be set if any bit shifted out from the low order bits is non-zero, otherwise it will be cleared.

Instruction Format: SHIFT

	31	3029	28 17	16	15 11	10 6	5 0
SRL	0	0_2	1_3 H ~ $Shamt_6$	Cr	$Rs1_5$	Rd_5	2_6
SRL	1	0_2	1_3 H $\sim_3$ $Rs2_5$	Cr	$Rs1_5$	Rd_5	2_6

Operation:

Rd = Rs1 >> Rs2

OR

Rd = Rs1 >> constant

Operation Size:

Execution Units: integer ALU

Exceptions: none

Example:

Logical Operations

AND[.] – Bitwise And

Description:

And two source registers Rs1 and Rs2 or 'and' Rs1 and a constant and place the result in the destination register Rd. All register values are integers. Condition register zero may be updated with the result of the operation compared to zero.

Instruction Format: R3

	31	3029	28 17	16	15 11	10 6	5 0
AND	L	LX_2	$Immediate_{12}$	Cr	$Rs1_5$	Rd_5	8_6
AND	1	LX_2	0_3 $\sim_4$ $Rs2_5$	Cr	$Rs1_5$	Rd_5	8_6

Operation:

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

XOR[.] – Bitwise Exclusive Or

Description:

Bitwise exclusively or two source registers Rs1 and Rs2 OR Rs1 and a constant and place the result in the destination register. All registers are integer registers.

Instruction Format: R3

	31	3029	28 17	16	15 11	10 6	5 0
XOR	L	LX_2	$Immediate_{12}$	Cr	$Rs1_5$	Rd_5	10_6
XOR	1	LX_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	10_6

Operation: R3

Rt = Ra ^ Rb

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

OR[.] – Bitwise Or

Description:

Bitwise or two source registers Rs1 and Rs2 OR Rs1 and a constant, and place the result in the destination register Rd. All registers are integer registers.

Instruction Format:

	31	3029	28 17	16	15 11	10 6	5 0
OR	L	LX_2	$Immediate_{12}$	Cr	$Rs1_5$	Rd_5	9_6
OR	1	LX_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	9_6

Operation:

Rd = Rs1 | Rs2

OR

Rd = Rs1 | Constant

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

CHK – Check Register Against Bounds

Description:

A register, Rs1, is compared to two values. If the register is outside of the bounds defined by Rs2 and Rs3 then a bounds check exception will occur. Comparisons may be signed or unsigned, indicated by 'S', 1 = signed, 0 = unsigned. The constant $Offs_3$ is multiplied by four and added to the program counter address of the CHK instruction and stored on an internal stack. This allows a return to a point up to 256 bytes after the CHK. Typical values are zero or one.

Instruction Format: R2

ALU Instruction Formats

31	30 27	26 17		16	15 11	10 6	5 0
0	Op_4	$Rs3_5$	$Rs2_5$	~	$Rs1_5$	$Offs_5$	29_6

Op_4	exception when not	
0	Ra >= Rb and Ra < Rc	
1	Ra >= Rb and Ra <= Rc	
2	Ra > Rb and Ra < Rc	
3	Ra > Rb and Ra <= Rc	
4	Not (Ra >= Rb and Ra < Rc)	
5	Not (Ra >= Rb and Ra <= Rc)	
6	Not (Ra > Rb and Ra < Rc)	
7	Not (Ra > Rb and Ra <= Rc)	
8	Ra >= CPL	CHKCPL – code privilege level
9	Ra <= CPL	CHKDPL – data privilege level
10	Ra == SC	Stack canary check

Operation:

IF check failed

PUSH SR onto internal stack

PUSH PC plus O_5 * 4 onto internal stack

PC = vector at (tvec[3])

Clock Cycles: 1

Execution Units: Integer ALU

Exceptions: bounds check

Notes:

The system exception handler will typically transfer processing back to a local exception handler.

Data Movement

MOVE[.] / MOVEA[.] / MOVSZ[.] / MOVZX[.] – Move Register to Register

Description:

Move register-to-register. This instruction may move between different types of registers. Raw binary data is moved. No data conversions are applied. Some registers are accessible only in specific operating modes. Some registers are read-only. Normally referencing the stack pointer register r31 will map to the stack pointer according to the operating mode, however the 'MOVA' instruction may be used to disable this. The MOVSX and MOVZX instructions perform moves with sign and zero extensions from the specified bit respectively.

Instruction Formats: MOV

	31	3028	27		17	16	15 11	10 6	5 0
MOVE	0	0_3	$\sim_9$	$Rs1_{65}$	Rd_{65}	Cr	$Rs1_{4...0}$	$Rd_{4...0}$	$Opcode_6$
MOVEA	0	1_3	$\sim_9$	$Rs1_{65}$	Rd_{65}	Cr	$Rs1_{4...0}$	$Rd_{4...0}$	$Opcode_6$
MOVSX	0	2_3	$\sim_3$ $Uimm_6$	$Rs1_{65}$	Rd_{65}	Cr	$Rs1_{4...0}$	$Rd_{4...0}$	$Opcode_6$
MOVZX	0	3_3	$\sim_3$ $Uimm_6$	$Rs1_{65}$	Rd_{65}	Cr	$Rs1_{4...0}$	$Rd_{4...0}$	$Opcode_6$

Operation: R2

Rt = Ra

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Ra_7 / Rt_7	Register file	Mode Access	RW
0 to 30	General purpose registers 0 to 30	USHM	RW
31	Safe stack pointer	SHM	RW
32 to 63	Floating-point registers	USHM	RW
64	User stack pointer	USHM	RW
65	Supervisor stack pointer	SHM	RW
66	Hypervisor stack pointer	HM	RW
67	Machine stack pointer	M	RW
68 to 71	Micro-code temporaries #0 to #3	HM	RW
72 to 78	Branch registers	USHM	RW

79	Instruction pointer	USHM	R
80 to 87	Condition Registers	USHM	RW
88	Loop Counter	USHM	RW
89	micro-code link register	HM	RW
90	Context block address register	SHM	RW
91	Micro-code program counter	SHM	RW

Load / Store Instructions

Overview

Addressing Modes

Load and store instructions have two addressing modes: register indirect with displacement and scaled indexed addressing. Note that store instructions cannot updated CR0.

Register Indirect with Displacement Format

0	LX_2	$Displacement_{11...0}$	Cr	$Rs1_5$	Rd_5	$Opcode_5$

Scaled Indexed with Displacement Format

For scaled indexed with displacement format the load or store address is the sum of register Rs1, scaled register Rs2, and a displacement constant found in the instruction.

Instruction Format: d[Rs1+Rs2*]

1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	$Opcode_5$

LDB[.] Rn, <ea> - Load Byte

Description:

Load register Rd with a byte of data from source. The source value is sign extended to the machine width.

Instruction Formats

Disp	L	LX_2	$Displacement_{11...0}$			Cr	$Rs1_5$	Rd_5	32_5
Indexed Ld	1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	32_5

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDBZ[.] Rn, <ea> - Load Byte and Zero Extend

Description:

Load register Rd with a byte of data from source. The source value is zero extended to the machine width.

Instruction Formats

Disp	L	LX_2	$Displacement_{11...0}$			Cr	$Rs1_5$	Rd_5	33_5
Indexed Ld	1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	33_5

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDT[.] Rn, <ea> - Load Tetra

Description:

Load register Rd with a tetra of data from source. The source value is sign extended to the machine width.

Instruction Formats

Disp	L	LX_2	$Displacement_{11...0}$			Cr	$Rs1_5$	Rd_5	36_5
Indexed Ld	1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	36_5

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDTZ[.] Rn, <ea> - Load Tetra and Zero Extend

Description:

Load register Rd with a tetra of data from source. The source value is zero extended to the machine width.

Instruction Formats

Disp	L	LX_2	$Displacement_{11...0}$			Cr	$Rs1_5$	Rd_5	37_5
Indexed Ld	1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	37_5

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDW[.] Rn, <ea> - Load Wyde

Description:

Load register Rd with a wyde of data from source. The source value is sign extended to the machine width.

Instruction Formats

Disp	L	LX_2	$Displacement_{11...0}$			Cr	$Rs1_5$	Rd_5	34_5
Indexed Ld	1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	34_5

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDWZ[.] Rn, <ea> - Load Wyde and Zero Extend

Description:

Load register Rd with a wyde of data from source. The source value is zero extended to the machine width.

Instruction Formats

Disp	L	LX_2	$Displacement_{11...0}$			Cr	$Rs1_5$	Rd_5	35_5
Indexed Ld	1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	35_5

Execution Units: AGEN, MEM

Exceptions:

Notes:

LOAD[.] Rn, <ea> - Load

Description:

This is an alternate mnemonic for the LDO instruction. Load register Rd with an octa byte of data from source.

Instruction Formats

Disp	L	LX_2	$Displacement_{11...0}$			Cr	$Rs1_5$	Rd_5	38_5
Indexed Ld	1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	38_5

Execution Units: AGEN, MEM

Exceptions:

Notes:

STB Rn, <ea> - Store Byte

Description:

Store the lowest byte from register Rs to memory.

Instruction Format

Disp	0	LX_2	$Displacement_{11...0}$			U	$Rs1_5$	$Rs2_5$	40_5
Indexed St	1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	U	$Rs1_5$	$Rs3_5$	40_5

Execution Units: AGEN, MEM

Exceptions:

Notes:

STBI Rn, <ea> - Store Byte Immediate

Description:

Store a constant byte to memory. The constant is located in the last half of the cache line offset by CL_3 words.

Instruction Format

Disp	0	LX_2	$Displacement_{11...0}$			U	$Rs1_5$	~	CL_3	41_5
Indexed St	1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	U	$Rs1_5$	~	CL_3	41_5

Execution Units: AGEN, MEM

Exceptions:

Notes:

STORE Rn, <ea> - Store Register

Description:

This is an alternate mnemonic for the STO instruction. Store register Rs to memory.

Instruction Formats

Disp	0	LX_2	$Displacement_{11...0}$			U	$Rs1_5$	$Rs2_5$	$Opcode_5$
Indexed St	1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	U	$Rs1_5$	$Rs3_5$	$Opcode_5$

Execution Units: AGEN, MEM

Exceptions:

Notes:

STOREI N, <ea> - Store Immediate

Description:

This is an alternate mnemonic for the STO instruction. Store immediate value to memory. The immediate value is referenced as a constant on the cache line. The index to the memory containing the constant is specified by CL_3. Note that the immediate constants may be located only in the second half of a cache line. There are only eight possible locations.

Instruction Formats

Disp	0	LX_2	$Displacement_{11...0}$			U	$Rs1_5$	~	CL_3	$Opcode_5$
Indexed St	1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	U	$Rs1_5$	~	CL_3	$Opcode_5$

Execution Units: AGEN, MEM

Exceptions:

Notes:

STT Rn, <ea> - Store Tetra

Description:

Store the lowest tetra (4 bytes) from register Rs2 to memory.

Instruction Format

Disp	0	LX_2	$Displacement_{11...0}$			U	$Rs1_5$	$Rs2_5$	44_5
Indexed St	1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	U	$Rs1_5$	$Rs3_5$	44_5

Execution Units: AGEN, MEM

Exceptions:

Notes:

STTI Rn, <ea> - Store Tetra Immediate

Description:

Store a constant tetra to memory. The constant is located in the last half of the cache line offset by CL_3 words.

Instruction Format

Disp	0	LX_2	$Displacement_{11...0}$			U	$Rs1_5$	~	CL_3	45_5
Indexed St	1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	U	$Rs1_5$	~	CL_3	45_5

Execution Units: AGEN, MEM

Exceptions:

Notes:

STW Rn, <ea> - Store Wyde

Description:

Store the lowest wyde (2 bytes) from register Rs2 to memory.

Instruction Format

Disp	0	LX_2	$Displacement_{11...0}$			U	$Rs1_5$	$Rs2_5$	42_5
Indexed St	1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	U	$Rs1_5$	$Rs3_5$	42_5

Execution Units: AGEN, MEM

Exceptions:

Notes:

STWI Rn, <ea> - Store Wyde Immediate

Description:

Store a constant wyde to memory. The constant is located in the last half of the cache line offset by CL_3 words.

Instruction Format

Disp	0	LX_2	$Displacement_{11...0}$			U	$Rs1_5$	~	CL_3	43_5
Indexed St	1	LX_2	$Disp_5$	Sc_2	$Rs2_5$	U	$Rs1_5$	~	CL_3	43_5

Execution Units: AGEN, MEM

Exceptions:

Notes:

Condition Register Instructions

CLC – Clear Carry

Description:

This is an alternate mnemonic for the CRANDC instruction where the manipulated bit in the condition register is the carry bit (bit 5 of the CR).

Instruction Format: R3

	31	3029	28	18	17 12	11 6	5 0
CLC	0	0_2	3_4	$\sim_6$	1	$CRs1_3,5_3$	$CRd_3,5_3$	11_6

Operation:

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

CLV – Clear Overflow

Description:

This is an alternate mnemonic for the CRANDC instruction where the manipulated bit in the condition register is the overflow bit (bit 6 of the CR).

Instruction Format: R3

	31	3029	28	18	17 12	11 6	5 0
CLV	0	0_2	3_4	$\sim_6$	1	$CRs1_3,6_3$	$CRd_3,6_3$	11_6

Operation:

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

CRAND – Bit And

Description:

Bit ‘and’ two source condition register bits CRs1 and CRs2 OR source condition register Rs1 and a constant bit and place the result in the destination condition register bit CRd.

Instruction Format: R3

	31	3029	28		18	17	12	11	6	5	0
CRAND	0	0_2	0_4	$\sim_6$	I	$CRs1_6$		CRd_6		11_6	
CRAND	1	0_2	0_4	~ $CRs2_6$		$CRs1_6$		CRd_6		11_6	

Operation:

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

CRANDC – Bit And with Complement

Description:

Bit ‘and’ with complement two source condition register bits CRs1 and CRs2 OR source condition register Rs1 and a constant bit and place the result in the destination condition register bit CRd. This instruction may be used to clear the specified bit in the condition register, for example, the carry bit.

Instruction Format: R3

	31	3029	28		18	17	12	11	6	5	0
CRANDC	0	0_2	3_4	$\sim_6$	I	$CRs1_6$		CRd_6		11_6	
CRANDC	1	0_2	3_4	~ $CRs2_6$		$CRs1_6$		CRd_6		11_6	

Operation:

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

CROR – Bit Or

Description:

Bit ‘or’ two source condition register bits CRs1 and CRs2 OR source condition register Rs1 and a constant bit and place the result in the destination condition register bit CRd. This instruction may be used to set a bit in a condition register. For example, the carry bit.

Instruction Format: R3

	31	3029	28 18		17 12	11 6	5 0
CROR	0	0_2	1_4	$\sim_6$ |	$CRs1_6$	CRd_6	11_6
CROR	1	0_2	1_4	~ $CRs2_6$	$CRs1_6$	CRd_6	11_6

Operation:

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

CRXOR – Bit Exclusive Or

Description:

Bit exclusive ‘or’ two source condition register bits CRs1 and CRs2 OR source condition register Rs1 and a constant bit and place the result in the destination condition register bit CRd. This instruction may be used to flip a bit in a condition register. For example, the carry bit.

Instruction Format: R3

	31	3029	28		18	17	12	11	6	5	0
CRXOR	0	0_2	2_4	$\sim_6$	I	$CRs1_6$		CRd_6		11_6	
CRXOR	1	0_2	2_4	~ $CRs2_6$		$CRs1_6$		CRd_6		11_6	

Operation:

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

SEC – Set Carry

Description:

This is an alternate mnemonic for the CROR instruction where the manipulated bit in the condition register is the carry bit (bit 5 of the CR).

Instruction Format: R3

	31	3029	28	18	17 12	11 6	5 0
SEC	0	0_2	1_4	$\sim_6$ 1	$CRs1_3,5_3$	$CRd_3,5_3$	11_6

Operation:

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

SEV – Set Overflow

Description:

This is an alternate mnemonic for the CROR instruction where the manipulated bit in the condition register is the overflow bit (bit 6 of the CR).

Instruction Format: R3

	31	3029	28	18	17 12	11 6	5 0
SEV	0	0_2	1_4	$\sim_6$ 1	$CRs1_3,6_3$	$CRd_3,6_3$	11_6

Operation:

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

Branch / Flow Control Instructions

Overview

Mnemonics

There are mnemonics for specifying the comparison method. Floating-point comparisons prefix the branch mnemonic with 'F' as in FBEQ. There is no prefix for integer branches. For branches that decrement the loop count register LC, the mnemonic is prefixed with the decrement condition as in 'DNZ_BNE'.

Conditional Branch Format

	31	3029	28 16			15 11		10 6		5 0	
[Dcc]Bcc[L]	0	D_{12}	BRs_3	Cnd_4	CRs_6	$Disp_{10...3}$			BRd_3	12_5	D
[Dcc]Bcc[L]	1	0_2	BRs_3	Cnd_4	CRs_6	~	$Rs1_5$	$\sim_2$	BRd_3	12_5	
[Dcc]Bcc[L]	1	1_2	BRs_3	Cnd_4	CRs_6	$\sim_4$	CL_3	~	BRd_3	12_5	

Field	Purpose
Cnd_4	Branch condition that must be met
CRs_6	Condition register bit to test; low 3 bits are bit number, high 3 are regno
BRd_3	Destination branch linkage register to store return address
BRs_3	Source branch linkage register to get address to jump to
$Disp_{13}$	13-bit displacement from BRs_3

Note that extended displacements are possible. A 32 or 64-bit displacement may be selected by setting bits 29 to 31 of the instruction appropriately.

Predicated Execution

Branch instructions will execute only if both the predicate and branch condition are true.

Conditions

Conditional branches branch to the target address only if the condition is true. The condition is determined by the status of the loop count register and the specified condition register bit state. The condition register will have typically been previously set by a compare instruction.

Table of Conditions

Cnd_4	Tested Conditions	Mnemonic
0	Decrement and branch if LC non-zero and condition is false	DBNZ_Bcc
1	Decrement and branch if LC zero and condition is false	DBZ_Bcc
2	Branch if condition false	Bcc
3	Decrement and branch if LC non-zero and condition is true	DBNZ_Bcc
4	Decrement and branch if LC zero and condition is true	DBZ_Bcc
5	Branch if condition true	Bcc
6	Decrement and branch if LC non-zero	DBNZ
7	Decrement and branch if LC zero	DBZ
Others	reserved	

The condition register contains eight bits with the following format:

Bit		
0	EQ / XNOR	Set if bitwise XNOR of operands is true, equal
1	NAND	Set if logical NAND of operands is true
2	NOR	Set if logical NOR of operands is true
3	LT	Set if less than
4	LE	Set if less than or equal
5	CA	Carry out from operation (addition, subtraction, shift)
6	OF / UN	Overflow status or unordered for floating-point
7		

Branch Target

Conditional Branches

For conditional branches, the destination address is formed as the sum of a code address (branch) register and a displacement constant specified in the instruction. Relative branches have a maximum range of 64 displacement bits. Inherent in the first word of the instruction is a 13-bit displacement making the address range ±4kB.

> *The destination displacement field is recommended to be at least 16-bits. It is possible to get by with a displacement as small as 12-bits before a significant percentage of branches must be implemented as two or more instructions.*

Decrementing Branches

Branches may decrement the loop count register by one after performing the branch comparison or logical operation. The condition field of the instruction indicates when a change should occur. Decrementing branches make use of both the flow control unit and an ALU at the same time.

Unconditional Branches

The destination displacement field is large enough to accommodate a $\pm 2^{24}$ range or ±16MB. The range may be extended using extended constants to 32 or 64 bits. The destination address is formed as the sum of a code address register PC and the displacement constant. The return address may be stored in register BRd.

	31	3029	28 16	15 11	10 6	5 0	
B[L]	L	LX_2	$Displacement_{22...3}$		BRd_3	13_5	D
BLR[L]	1	0_2	BRs_3	$Immediate_{19...3}$	BRd_3	13_5	D

B – Branch Always

B label[BR]

Description:

This instruction always jumps to the destination address. The destination address range is $\pm 2^{24}$ bits. This is an alternate mnemonic for the BL instruction where the branch link register is BR0.

Formats Supported: B

	31	3029	28 16	15 11	10 6	5 0	
B	L	LX_2	$Displacement_{22...3}$		0_3	13_5	D

Operation:

PC = PC + Constant

Execution Units: Flow Control

Clock Cycles: 13

Exceptions: none

Notes:

BAND –Branch if And

BAND CRs, label[BRs]

Description:

Branch if the logical and of operation resulted in a true condition.

Formats Supported: BR

	31	3029	28			16	15 11	10 6	5 0	
BAND	0	D_{12}	BRs_3	2_4	$CRs,1_3$		$Disp_{10...3}$	0_3	12_5	D

Clock Cycles: 13

BANDL –Branch if And and Link

BANDL CRs, BRd, label[BRs]

Description:

Branch if the logical and operation resulted in a true condition. Store the address of the next instruction in BRd.

Formats Supported: BR

	31	3029	28			16	15 11	10 6	5 0	
BANDL	0	D_{12}	BRs_3	2_4	$CRs,1_3$		$Disp_{10...3}$	BRd_3	12_5	D

Clock Cycles: 13

BCC –Branch if Carry Clear

BCC CRs, label[BRs]

Description:

Branch if the operation resulted in no carry condition.

Formats Supported: BR

	31	3029	28			16	15 11	10 6	5 0	
BCC	0	D_{12}	BRs_3	2_4	$CRs,5_3$		$Disp_{10...3}$	0_3	12_5	D

Clock Cycles: 13

BCCL –Branch if Carry Clear and Link

BCCL CRs, BRd, label[BRs]

Description:

Branch if the operation did not result in a carry condition. Store the address of the next instruction in BRd.

Formats Supported: BR

	31	3029	28			16	15 11	10 6	5 0	
BCCL	0	D_{12}	BRs_3	2_4	$CRs,5_3$		$Disp_{10...3}$	BRd_3	12_5	D

Clock Cycles: 13

BCS –Branch if Carry Set

BCS CRs, label[BRs]

Description:

Branch if the operation resulted in a carry condition.

Formats Supported: BR

	31	3029	28			16	15 11	10 6	5 0	
BCS	0	D_{12}	BRs_3	5_4	$CRs,5_3$		$Disp_{10...3}$	0_3	12_5	D

Clock Cycles: 13

BCSL –Branch if Carry Set and Link

BCSL CRs, BRd, label[BRs]

Description:

Branch if the operation resulted in a carry condition. Store the address of the next instruction in BRd.

Formats Supported: BR

	31	3029	28			16	15 11	10 6	5 0	
BCSL	0	D_{12}	BRs_3	5_4	$CRs,5_3$		$Disp_{10...3}$	BRd_3	12_5	D

Clock Cycles: 13

BEQ –Branch if Equal

BEQ CRs, label[BRs]

Description:

Branch if source operands were equal as a result of a compare operation.

Formats Supported: BR

	31	3029	28		16	15 11	10 6	5 0	
BEQ	0	D_{12}	BRs_3	5_4	$CRs,0_3$	$Disp_{10...3}$	0_3	12_5	D

Clock Cycles: 13

BEQL –Branch if Equal and Link

BEQL CRs, BRd, label[BRs]

Description:

Branch if source operands were equal as a result of a compare operation.

Formats Supported: BR

	31	3029	28		16	15 11	10 6	5 0	
BEQL	0	D_{12}	BRs_3	5_4	$CRs,0_3$	$Disp_{10...3}$	BRd_3	12_5	D

Clock Cycles: 13

BGE –Branch if Greater Than or Equal

BGE CRs, label[BRs]

Description:

Branch if source operands were greater than or equal as a result of a compare operation.

Formats Supported: BR

	31	3029	28		16	15 11	10 6	5 0	
BGE	0	D_{12}	BRs_3	2_4	$CRs,3_3$	$Disp_{10...3}$	0_3	12_5	D

Clock Cycles: 13

BGEL –Branch if Less Than or Equal and Link

BGEL CRs, BRd, label[BRs]

Description:

Branch if source operands were greater than or equal as a result of a compare operation. The destination address range is $\pm 2^{31}$ bits. Branch register BR7 may not be used to store the return address as it is a reference to the program counter.

Formats Supported: BR

	31	3029	28			16	15 11	10 6	5 0	
BGEL	0	D_{12}	BRs_3	2_4	$CRs,3_3$		$Disp_{10...3}$	BRd_3	12_5	D

Clock Cycles: 13

BGT –Branch if Greater Than

BGT CRs, label[BRs]

Description:

Branch if source operands were greater than as a result of a compare operation.

Formats Supported: BR

	31	3029	28			16	15 11	10 6	5 0	
BGE	0	D_{12}	BRs_3	2_4	$CRs,4_3$		$Disp_{10...3}$	0_3	12_5	D

Clock Cycles: 13

BGTL –Branch if Less Than or Equal and Link

BGTL CRs, BRd, label[BRs]

Description:

Branch if source operands were greater than as a result of a compare operation. The destination address range is $\pm 2^{31}$ bits. Branch register BR7 may not be used to store the return address as it is a reference to the program counter.

Formats Supported: BR

	31	3029	28			16	15 11	10 6	5 0	
BGEL	0	D_{12}	BRs_3	2_4	$CRs,4_3$		$Disp_{10...3}$	BRd_3	12_5	D

Clock Cycles: 13

BL – Branch and Link

BL BR, label[BR]

Description:

This instruction always jumps to the destination address. The destination address range is $\pm 2^{31}$ bits. Branch register BR7 may not be used to store the return address as it is a reference to the program counter.

Formats Supported: BL

	31	3029	28 16	15 11	10 6	5 0	
BL	L	LX_2	$Displacement_{22...3}$		BRd_3	13_5	D

Operation:

BRd = next PC

PC = PC + Constant

Execution Units: Flow Control

Clock Cycles: 13

Exceptions: none

Notes:

BLR – Branch to Link Register

BLR label[BR]

Description:

This instruction always jumps to the destination address. The destination address is formed as the contents of BRs added to a 20-bit displacement. This instruction may be used to return from a subroutine.

Formats Supported: BL

	31	3029	28	16	15 11	10 6	5 0	
BLR	1	0_2	BRs_3	$Immediate_{19...3}$		0_3	13_5	D

Operation:

PC = PC + BRs + Constant

Execution Units: Flow Control

Clock Cycles: 13

Exceptions: none

Notes:

BLRL – Branch to Link Register and Link

BLRL BRd,label[BRs]

Description:

This instruction always jumps to the destination address. The destination address is formed as the contents of BRs added to a 20-bit displacement. The address of the next instruction is stored in a branch link register BRd. Branch register BR7 may not be used to store the return address as it is a reference to the program counter.

Formats Supported: BL

	31	3029	28 16	15 11	10 6	5 0	
BLR[L]	1	0_2	BRs_3	$Immediate_{19...3}$	BRd_3	13_5	D

Operation:

BRd = next PC

PC = PC + BRs + Constant

Execution Units: Flow Control

Clock Cycles: 13

Exceptions: none

Notes:

BLE –Branch if Less Than or Equal

BLE CRs, label[BRs]

Description:

Branch if source operands were less than or equal as a result of a compare operation.

Formats Supported: BR

	31	3029	28 16			15 11	10 6	5 0	
BLE	0	D_{12}	BRs_3	5_4	$CRs,4_3$	$Disp_{10...3}$	0_3	12_5	D

Clock Cycles: 13

BLEL –Branch if Less Than or Equal and Link

BLEL CRs, BRd, label[BRs]

Description:

Branch if source operands were less than or equal as a result of a compare operation. The destination address range is $\pm 2^{31}$ bits. Branch register BR7 may not be used to store the return address as it is a reference to the program counter.

Formats Supported: BR

	31	3029	28 16			15 11	10 6	5 0	
BLEL	0	D_{12}	BRs_3	5_4	$CRs,4_3$	$Disp_{10...3}$	BRd_3	12_5	D

Clock Cycles: 13

BLT –Branch if Less Than

BLT CRs, label[BRs]

Description:

Branch if source operands were less than as a result of a compare operation.

Formats Supported: BR

	31	3029	28			16	15 11	10 6	5 0	
BLT	0	D_{12}	BRs_3	5_4	$CRs,3_3$		$Disp_{10...3}$	0_3	12_5	D

Clock Cycles: 13

BLTL –Branch if Less Than and Link

BLTL CRs, BRd, label[BRs]

Description:

Branch if source operands were less than as a result of a compare operation. The destination address range is $\pm 2^{31}$ bits. Branch register BR7 may not be used to store the return address as it is a reference to the program counter.

Formats Supported: BR

	31	3029	28			16	15 11	10 6	5 0	
BLTL	0	D_{12}	BRs_3	5_4	$CRs,3_3$		$Disp_{10...3}$	BRd_3	12_5	D

Clock Cycles: 13

BNAND –Branch if Nand

BNAND CRs, label[BRs]

Description:

Branch if the logical nand of operation resulted in a true condition.

Formats Supported: BR

	31	3029	28 16			15 11	10 6	5 0	
BNAND	0	D_{12}	BRs_3	5_4	$CRs,1_3$	$Disp_{10...3}$	0_3	12_5	D

Clock Cycles: 13

BNANDL –Branch if Nand and Link

BNANDL CRs, BRd, label[BRs]

Description:

Branch if the logical nand operation resulted in a true condition. Store the address of the next instruction in BRd.

Formats Supported: BR

	31	3029	28 16			15 11	10 6	5 0	
BNANDL	0	D_{12}	BRs_3	5_4	$CRs,1_3$	$Disp_{10...3}$	BRd_3	12_5	D

Clock Cycles: 13

BNOR –Branch if Nor

BNOR CRs, label[BRs]

Description:

Branch if the logical nor of operation resulted in a true condition.

Formats Supported: BR

	31	3029	28 16			15 11	10 6	5 0	
BNOR	0	D_{12}	BRs_3	5_4	$CRs,2_3$	$Disp_{10...3}$	0_3	12_5	D

Clock Cycles: 13

BNORL –Branch if Nor and Link

BNORL CRs, BRd, label[BRs]

Description:

Branch if the logical nor operation resulted in a true condition. Store the address of the next instruction in BRd.

Formats Supported: BR

	31	3029	28 16			15 11	10 6	5 0	
BNORL	0	D_{12}	BRs_3	5_4	$CRs,2_3$	$Disp_{10...3}$	BRd_3	12_5	D

Clock Cycles: 13

BNE –Branch if Not Equal

BNE CRs, label[BRs]

Description:

Branch if source operands were not equal as a result of a compare operation.

Formats Supported: BR

	31	3029	28 16			15 11	10 6	5 0	
BNE	0	D_{12}	BRs_3	2_4	$CRs,0_3$	$Disp_{10...3}$	0_3	12_5	D

Clock Cycles: 13

BNEL –Branch if Not Equal and Link

BNEL CRs, BRd, label[BRs]

Description:

Branch if source operands were not equal as a result of a compare operation. The destination address range is $\pm 2^{31}$ bits. Branch register BR7 may not be used to store the return address as it is a reference to the program counter.

Formats Supported: BR

	31	3029	28 16			15 11	10 6	5 0	
BNEL	0	D_{12}	BRs_3	2_4	$CRs,0_3$	$Disp_{10...3}$	BRd_3	12_5	D

Clock Cycles: 13

BOR –Branch if Or

BOR CRs, label[BRs]

Description:

Branch if the logical nor of operation resulted in a true condition.

Formats Supported: BR

	31	3029	28			16	15 11	10 6	5 0	
BOR	0	D_{12}	BRs_3	2_4	$CRs,2_3$		$Disp_{10...3}$	0_3	12_5	D

Clock Cycles: 13

BORL –Branch if Or and Link

BORL CRs, BRd, label[BRs]

Description:

Branch if the logical or operation resulted in a true condition. Store the address of the next instruction in BRd.

Formats Supported: BR

	31	3029	28			16	15 11	10 6	5 0	
BORL	0	D_{12}	BRs_3	2_4	$CRs,2_3$		$Disp_{10...3}$	BRd_3	12_5	D

Clock Cycles: 13

BVS –Branch if Overflow Set

BVS CRs, label[BRs]

Description:

Branch if the operation resulted in an overflow condition.

Formats Supported: BR

	31	3029	28 16			15 11	10 6	5 0	
BVS	0	D_{12}	BRs_3	5_4	$CRs,6_3$	$Disp_{10...3}$	0_3	12_5	D

Clock Cycles: 13

BVSL –Branch if Overflow Set and Link

BVSL CRs, BRd, label[BRs]

Description:

Branch if the operation resulted in an overflow condition. Store the address of the next instruction in BRd.

Formats Supported: BR

	31	3029	28 16			15 11	10 6	5 0	
BVSL	0	D_{12}	BRs_3	5_4	$CRs,6_3$	$Disp_{10...3}$	BRd_3	12_5	D

Clock Cycles: 13

NOP – No Operation

NOP

Description:

This instruction does not perform any operation. Any value for bits 17 to 28 may be used.

Instruction Format:

	31	3029	28 17	16	15 11	10 6	5 0
NOP	L	LX_2	$Immediate_{12}$	0	$\sim_5$	0_5	63_6

Notes:

System Instructions

BRK – Break

Description:

This instruction may initiate the processor debug routine if the BRK value matches the value set in the debug control register OR if the value zero is used. BRK instructions are treated as NOPs unless the value matches, excepting for the value zero. The processor enters debug mode. The cause code register is set to indicate execution of a BRK instruction. Interrupts are disabled. The program counter is reset to the vector located from the contents of tvec[3] and instructions begin executing. There should be a jump instruction placed at the break vector location. The address of the BRK instruction is stored in the EPC.

The debug BRK register is set to the value specified in the instruction.

Values with the MSB set will also trigger trace.

Instruction Format: SYS

	31	3029	28 17	16	15 11	10 6	5 0
BRK	0	0_2	$Value_{12}$	0	0_5	0_5	0_6

Operation:

PUSH SR
PUSH PC
EPC = PC
PC = vector at (tvec[3])

Execution Units: Branch

Clock Cycles:

Exceptions: none

Notes:

REX – Redirect Exception

Description:

This instruction redirects an exception from an operating mode to a lower operating mode. This instruction if successful jumps to the target exception handler and does not return. If this instruction fails execution will continue with the next instruction.

This instruction may fail if exceptions are not enabled at the target level.

The location of the target exception handler is found in the trap vector register for that operating mode (tvec[xx]).

The cause (cause) and bad address (badaddr) registers of the originating mode are copied to the corresponding registers in the target mode.

If the 'S' bit of the instruction is set, then the privilege level will be set to either a constant in the PL_8 field or the value in register Rs2. Otherwise the privilege level will remain unchanged.

Instruction Format: EX

	31	3029	28 17				16	15 11	10 6	5 0
REX	1	0_2	1_2	Tm_2	$PL_{7...0}$		S	$\sim_5$	31_5	28_6
REX	0	0_2	1_2	Tm_2	$\sim_3$	$Rs2_5$	S	$\sim_5$	31_5	28_6

Tm_2	
0	redirect to user mode
1	redirect to supervisor mode
2	redirect to hypervisor mode
3	Redirect to machine mode (from debug)

Clock Cycles: 4

Execution Units: Branch

Example:

```
REX 1          ; redirect to supervisor handler
; If the redirection failed, exceptions were likely disabled at the target level.
; Continue processing so the target level may complete its operation.
```

```
RTE                ; redirection failed (exceptions disabled ?)
```

Notes:

Since all exceptions are initially handled in machine mode the machine handler must check for disabled lower mode exceptions.

SYS – System Call

Description:

Perform a system call. Interrupts are disabled. The program counter is reset to the contents of the vector loaded from tvec[3] and instructions begin executing. There should be a jump instruction placed at the vector location. The address of the instruction following the SYS instruction is pushed onto an internal stack.

Instruction Format: SYS

	31	3029	28 17	16	15 11	10 6	5 0
SYS	1	0_2	0_2	Immediate$_{10}$	~	Rs1$_5$	31_5	28_6
SYS	0	0_2	0_2	~$_5$	Rs2$_5$	~	Rs1$_5$	31_5	28_6

Operation:

PUSH SR onto internal stack
PUSH PC + 4 onto internal stack
PC = tvec[3]

Execution Units: Branch

Clock Cycles:

Exceptions: none

Notes:

TRAPcc – Trap if Condition Met

Description:

A register, Rs1, is compared to register Rs2 or an immediate value. If the relationship between the registers matches the trap condition, then a trap exception occurs.

Instruction Format: R2

	31	3029	28 17		16	15 11	10 6	5 0
TRAP	1	0_2	$Immediate_{12}$		~	$Rs1_5$	$Cond_5$	28_6
TRAP	0	0_2	$\sim_7$	$Rs2_5$	~	$Rs1_5$	$Cond_5$	28_6

$Cond_5$	exception when	
0	Rs1 == Rs2	
1	Rs1 <> Rs2	
2	Rs1 < Rs2	
3	Rs1 <= Rs2	
4	Rs1 >= Rs2	
5	Rs1 > Rs2	
6	Rs1 < Rs2 (unsigned)	
7	Rs1 <= Rs2 (unsigned)	
8	Rs1 >= Rs2 (unsigned)	
9	Rs1 > Rs2 (unsigned)	
10		
31	Always trap (system call)	

Operation:

IF check failed
    PUSH SR onto internal stack
    PUSH PC plus 4 onto internal stack
    PC = vector at (tvec[3] + cause*8)

Clock Cycles: 1

Execution Units: Integer ALU

Exceptions: bounds check

Notes:

The system exception handler will typically transfer processing back to a local exception handler.

XJMP – Exchange Jump

Description:

This instruction saves the current context at the address in the context block address register (CSR M_CBA), then loads the processor context from memory at the calculated effective address. After the context load, the context block address register is loaded with the effective address. The address of the context block must be a memory page address – the lower 13 bits of the address are always zero.

Instruction Format: d[Ra] **OR** d[Ra+Rb*Sc]

	31	3029	28 17	16	1514	15 11	10 6	5 0
XJMP	0	$Displacement_{26...13}$		0	$Rs1_5$		$Disp_{31...27}$	30_6
XJMP	1	0_2	$Disp_{17...13}$ 0_2 $Rs2_5$	1	$Rs1_5$		$Disp_{22...18}$	30_6

Operation:

Memory[CB] = CPU context
CPU context = Memory[Ra + (Rb * scale) + displacement]
CB = Ra + (Rb * scale) + displacement

Pre/Postfixes and Modifiers

ATOM Modifier

Description:

Treat the following sequence of instructions as an “atom”. The instruction sequence is executed with interrupts set to the specified interrupt privilege level. Interrupts may be disabled or enabled for up to twelve instructions. The non-maskable interrupt may not be masked. Each bit in the mask represents a subsequent instruction.

Note that since the processor fetches instructions in groups the mask effectively applies to the group. The mask guarantees that at least as many instructions as specified will be masked, but more may be masked depending on group boundaries.

Instruction Format: ATOM

	31	3029	28 17	16	15 12	11 6	5 0
ATOM	1	0_2	$Mask_{12}$	~	$~_4$	IPL_6	60_6

Assembler Syntax:

Example:

```
ATOM “7MMMMM”
LOAD a0,[a3]
SLT t0,a0,a1
PRED t0,~t0,r0,”AAB”
STORE a2,[a3]
LDI a0,1
LDI a0,0
```

```
ATOM “6MMM”
LOAD a1,[a3]
ADD t0,a0,a1
MOV a0,a1
STORE t0,[a3]
```

QEXT Prefix

Description:

This prefix extends the register selection for quad precision. Quad precision operations need to use register pairs to contain a quad precision value. The QEXT prefix specifies the registers used to contain bits 64 to 127 of the quad precision values.

Quad precision values are calculated using the QEXT prefix before the quad precision instruction.

Note that any of 64 registers may be selected.

Instruction Format: QEXT

Instruction Format: ATOM

	31	3029	28 17	16	15 11	10 6	5 0
QEXT	0	0_2	$\sim_7$	$Rs2_5$	~	$Rs1_5$	Rd_5	60_6

PFX[ABCD] – A/B/C/D Immediate Postfix

PFXA $1234

Description:

This instruction supplies immediate constant bits five to N for the preceding instruction, allowing a N-bit constant to be used in place of a register. The first five bits of the constant are specified by the register number field of the instruction. The Wh field of the instruction specifies which register is to be used as a constant.

Wh	Substitute Immediate for:
0	Rs1
1	Rs2
2	Rs3
3	Rd

*Only one postfix is supported per instruction.

Instruction Format:

	31	3029	28 17	16	15 8	7 6	5 0
PFX	0	LX_2	$Immediate_{25...5}$			Wh_2	61_6

Notes:

PRED Modifier

Description:

Apply the predicate to following instructions according to a bit mask. The predicate may be applied to a maximum of eight instructions. If the 'Z' bit is set, target register elements are set to zero if not masked. Each byte of the predicate register contains the mask bits for the corresponding instruction.

Instruction Format: PRED

31	3029	28 17	16	15 11	10 6	5 0
0	1_2	$Mask_{15...4}$	z	$Rs1_5$	~ $Mask_{3...0}$	60_6

Pred Modifier Scope	Mask Bit		Rn_8 Bits Tested
	0,1	Instruction zero	0 to 7
	2,3	Instruction one	8 to 15
	4,5	Instruction two	16 to 23
	6,7	Instruction three	24 to 31
	8,9	Instruction four	32 to 39
	10,11	Instruction five	40 to 47
	12,13	Instruction six	48 to 55
	14,15	Instruction seven	56 to 63

Mask Bit	Meaning
00	Ignore predicate bit (always execute)
01	reserved
10	Execute only if predicate bit in Rs1 is false
11	Execute only if predicate bit in Rs1 is true

Assembler Syntax:

After the instruction mnemonic the register containing the predicate flags is specified. Next a character string containing 'A' for Ra, 'B' for Rb, or 'I' for ignore for the next eight instructions is present.

Example:

```
PRED r2,"TIFIIIII"
  ; execute one if true, ignore one, next execute if false, one after always execute
MUL r3,r4,r5          ; executes if R2 True
ADD r6,r3,r7          ; always executes
```

```
ADD r6,r6,#1234      ; executes if R2 FALSE
DIV r3,r4,r5         ; always executes
```

MPU Hardware

Hardware Description

Caches

Overview

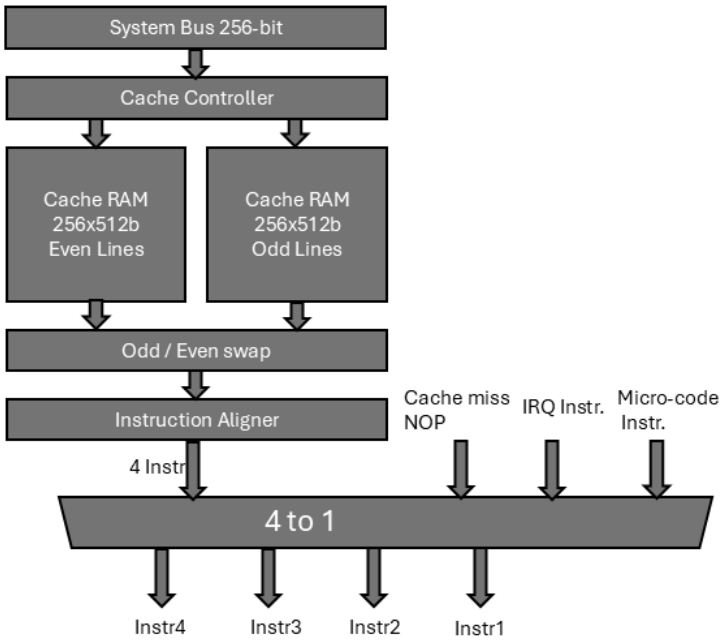
The core has both instruction and data caches to improve performance. Both caches are single level. The cache is four-way associative. The cache sizes of the instruction and data cache are available for reference from one of the info lines return by the CPUID instruction.

Instructions

Since the instruction format affects the cache design it is mentioned here. For this design instructions are of a fixed length being 32 bits in size. Specific formats are listed under the instruction set description section of this book.

L1 Instruction Cache

L1 is 32kB in size and made from block RAM with a single cycle of latency. L1 is organized as an odd, even pair of 256 lines of 64 bytes. The following illustration shows the L1 cache organization for Qupls3.



Note that the upper half of the cache line pair is available for each instruction so that constants may be decoded. This propagation of the cache line is not shown on the above diagram to keep it simple.

The cache is organized into odd and even lines to allow instructions to span a cache line. Two cache lines are fetched for every access; the one the instruction is located on, and the next one in case the instruction spans a line.

A 256-line cache was chosen as that matches the inherent size of block RAM component in the FPGA. It is the author's opinion that it would be better if the L1 cache were larger because it often misses due to its small size. In short, the current design is an attempt to make it easy for the tools to create a fast implementation.

Note that supporting interrupts and cache misses, a requirement for a realistic processor design, adds complexity to the instruction stream. Reading the cache ram, selecting the correct instruction word and accounting for interrupts and cache misses must all be done in a single clock cycle.

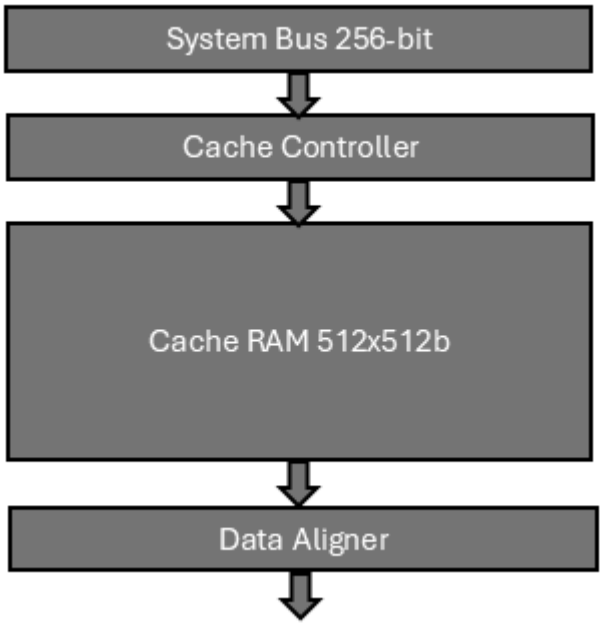
While the L1 cache has single cycle reads it requires two clock cycles to update (write) the cache. The cache line to update needs to be provided by the tag memory which is unknown until after the tag updates.

Fetch Rate

The fetch rate is four instructions per clock cycle.

Data Cache

The data cache organization is somewhat simpler than that of the instruction cache. Data is cached with a single level cache because it's not critical that the data be available within a single clock cycle at least not for the hobby design. Some of the latency of the data cache can be hidden by the presence of non-memory operating instructions in the instruction queue.



The data cache is organized as 512 lines of 64 bytes (32kB) and implemented with block ram. Access to the data cache is multicycle. The data cache may be replicated to allow more memory instructions to be processed at the same time; however, just a single cache is in use for the demo system. The policy for stores is write-through. Stores always write through to memory. Since stores follow a write-through policy the latency of the store operation depends on the external memory system. It isn't critical that the cache be able to update in single cycle as external memory access is bound to take many more cycles than a cache update. There is only a single write port on the data cache.

Cache Enables

The instruction cache is always enabled to keep hardware simpler and faster. Otherwise, an additional multiplexor and control logic would be required in the instruction stream to read from external memory.

For some operations, it may be desirable to disable the data cache so there is a data cache enable bit in control register #0. This bit may be set or cleared with one of the CSR instructions.

Cache Validation

A cache line is automatically marked as valid when loaded. The entire cache may be invalidated using the CACHE instruction. Invalidating a single line of the cache is not currently supported, but it is supported by the ISA. The cache may also be invalidated due to a write by another core via a snoop bus.

Un-cached Data Area

The address range $F...FDxxxxx is an un-cached 1MB data area. This area is reserved for I/O devices. The data cache may also be disabled in control register zero.

Return Address Stack Predictor (RSB)

There is an address predictor for return addresses which can in some cases can eliminate the flushing of the instruction queue when a return instruction is executed. The BLR instruction is detected in the fetch stage of the core and a predicted return address used to fetch instructions following the return. The return address stack predictor has a stack depth of 64 entries. On stack overflow or underflow, the prediction will be wrong, however performance will be no worse than not having a predictor. The return address stack predictor checks the address of the instruction queued following the BLR against the address fetched for the BLR instruction to make sure that the address corresponds.

Branch Predictor

The branch predictor is a (2, 2) correlating predictor. The branch history is maintained in a 512- entry history table. It has four read ports for predicting branch outcomes, one port for each instruction fetched. The branch predictor may be disabled by a bit in control register zero. When disabled all branches are predicted as not taken, unless specified otherwise in the branch instruction.

To conserve hardware the branch predictor uses a fifo that can queue up to four branch outcomes at the same time. Outcomes are removed from the fifo one at a time and used to update the branch history table which has only a single write port. In an earlier implementation of the branch predictor, two write ports were provided on the history table. This turned out to be relatively large compared to its usefulness.

Correctly predicting a branch turns the branch into a single cycle operation. During execution of the branch instruction the address of the following instruction queued is checked against the address depending on the branch outcome. If the address does not match what is expected, then the queue will be flushed, and new instructions loaded from the correct program path.

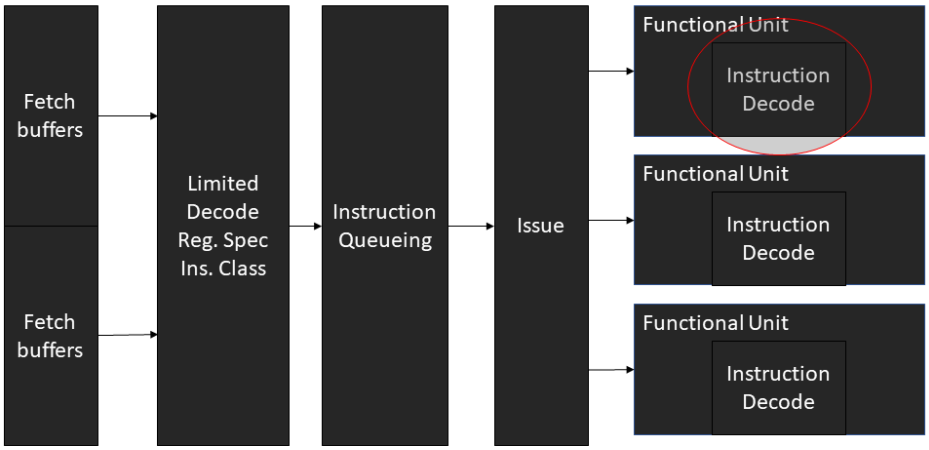
Branch Target Buffer (BTB)

The core has a 1k entry branch target buffer for predicting the target address of flow control instructions where the address is calculated and potentially unknown at time of fetch. Instructions covered by the BTB include jump-and-link, interrupt return and breakpoint instructions and branches to targets contained in a register.

Decode Logic

Instruction decode is distributed about the core. Although some decodes take place between fetch and instruction queue. Broad classes of instructions are decoded for the benefit of issue logic along with register specifications prior to instruction enqueue. Most of the decodes are done with modules under the decoder folder. Decoding typically involves reducing a wide input into a smaller number of output signals. Other decodes are done at instruction execution time with case statements.

Placement of Instruction Decode

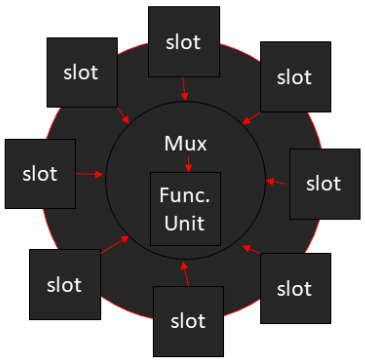


Limited decode takes place between fetch and queue. Between fetch and queue register specifications are decoded along with general instruction classes for the benefit of issue. A handful of additional signals (like sync) that control the overall operation of the core are also decoded. Much of the instruction decode is actually done in the functional unit. The instruction register is passed right through to the functional units in the core.

Instruction Queue (ROB)

The instruction queue is a 32-entry re-ordering buffer (ROB). The instruction queue tracks an instructions progress. Each instruction in queue may be in one of several different states. The instruction queue is a circular buffer with head and tail pointers. Instructions are queued onto the tail and committed to the machine state at the head. Queue and commit takes place in groups of up to four instructions.

Instruction Queue – Re-order Buffer



The instruction queue is circular with eight slots. Each slot feeds a multiplexor which in turn feeds a functional unit. Providing arguments to the functional unit is done under the vise of issue logic. Output from the functional unit is fed back to the same queue slot that issued to the functional unit.
The queue slots are fed from the fetch buffers.

Queue Rate

Up to four instructions may queue during the same clock cycle depending on the availability of queue slots.

Sequence Numbers

The queue maintains a 7-bit instruction sequence number which gives other operations in the core a clue as to the order of instructions. The sequence number is assigned when an instruction queues. Branch instructions need to know when the next instruction has queued to detect branch misses. The program counter cannot be used to determine the instruction sequence because there may be a software loop at work which causes the program counter to cycle backwards even though it's really the next instruction executing.

Input / Output Management

Before getting into memory management a word or two about I/O management is in order. Memory management depends on several I/O devices. I/O in the Qupls3 is memory mapped or MMIO. Ordinary load and store instructions are used to access I/O registers. I/O is mapped as a non-cacheable memory area.

Device Configuration Blocks

I/O devices have a configuration block associated with them that allows the device to be discovered by the OS during bootup. All the device configuration blocks are located in the same 1GB region of memory in the address range $C0000000 to $FFFFFFFF. Each device configuration block is aligned on a 16kB boundary. There is thus a maximum of 16k device configuration blocks.

Reset

At reset the device configuration blocks are not accessible. They must be mapped into memory for access. However, the devices have default addresses assigned to them, so it may not be necessary to map the device control block into memory before accessing the device. The device itself also needs to be mapped into the memory space for access though.

Devices Built into the CPU / MPU

Devices present in the CPU itself include:

Device	Bus	Device	Func	IRQ Priority	Config Block Address	Default Address
Interrupt Controller	0	6	0	~	$D0030000	$FEE2xxxx
Interval Timers	0	4	0	61	$D0020000	$FEE4xxxx
Memory Region Table	0	12	0	~	$D0060000	$FEEFxxxx

System Devices

Device	Bus	Device	Func	IRQ	Config Block Address	Default Address
Interrupt Reflector	0		0	~	TBD	TBD
Interrupt Logger	0		0	~	TBD	TBD

Function is mapped to address bits 14 to 16

Device is mapped to address bits 17 to 21

Bus is mapped to address bits 22 to 29

External Interrupts

Overview

External interrupts are interrupts external to the CPU and are usually generated by peripheral devices. External interrupts are usually events occurring asynchronously with respect to software running on a CPU. Qupls3 external interrupts make use of message signaling. Qupls3 does not follow the MSI / MSI-X standard exactly, although it is similar. The goal of Qupls3's MSI is to be frugal with logic resources. Qupls3 MSI Interrupts are signaled by peripheral devices placing an interrupt message on the peripheral slave response bus. This reuses the response bus pathway to the processing core. Slave peripherals do not need to include bus mastering logic that is normally present with MSI-X.

Interrupt Messages

Interrupt messages are placed on the response bus with an error status indicating an IRQ occurred. The interrupt message identifies the vector number, servicing operating mode, and servicing interrupt controller. This information is stored in a register in the peripheral. An additional 32-bit data word is present in the device to hold extended message information. Qupls3 MSI differs from MSI-X in the storage location of the extended interrupt message information. MSI-X stores this information in the interrupt table whereas Qupls3 stores it in the device. MSI-X requires the device to perform a write operation to the interrupt table, whereas Qupls3 MSI does not. MSI-X interrupts normally specify an I/O address to post to and a 32-bit data word. Unfortunately, in the Qupls3 system there are not enough bits in a 32-bit response bus to mimic MSI-X. The vector number combined with the interrupt controller number take the place of the I/O address. Additional information passed by the interrupt message (in the response address field) identifies the source of the interrupt, the desired priority level, and the software stack required for processing.

Interrupt Controller

The Qupls3 interrupt controller (QIC) is a slave peripheral device that detects interrupt messages occurring on the CPU response bus. It stores the interrupt message in a priority queue. The interrupt vector for the highest priority interrupt is looked up from an internal vector table. Information in the vector determines a list of possible target CPU cores and the software stack that must be available. Either the address of the interrupt subroutine (ISR) or, an instruction for the CPU to execute is

provided. There may be multiple interrupt controllers in the system. Currently a six-bit controller number is present in the interrupt message limiting the number of controllers to 62. With 62 interrupt controllers and each one servicing 62 CPU cores, a maximum of approximately 3800 CPU cores may be connected to interrupts.

The interrupt controller has some capacity to detect interrupt overruns. There is a "stuck interrupt" detector which flags an interrupt signal as being stuck if the same interrupt message is posted in a short time-frame. The queue full status flag is also available in the controller allowing software to detect if a queue is full. A full queue may also indicate a stuck interrupt.

There is more detail pertaining to QIC in the QIC device description later in this document.

Interrupt Vector Table

The interrupt vector table is internal to the interrupt controller. The table is laid out in four sections, one for each available operating mode. There are 2048 (512 in the demo system) vectors available for each operating mode. Note there may be multiple interrupt controllers in the system, and hence multiple vector tables. Which vector table to use is identified in a device control register in the form of specifying an interrupt controller number.

Interrupt Group Filter

There may be more than one CPU core connected to a QIC; up to 62 CPU cores may be connected to a QIC. Note that groups of CPU cores may be specified to handle an interrupt. There is a filter in the MPU that detects the lowest priority CPU core that is ready to handle an interrupt. The information from the QIC about the interrupt is passed to connected CPU cores.

To be ready to handle an interrupt, the current interrupt level of the CPU core must be less than that of the interrupting device, and the CPU core must be operating using the software stack appropriate for the interrupt.

Interrupt Reflector

The interrupt reflector is a peripheral device that allows a bus master to trigger an interrupt. Because interrupts are posted on the response bus for Qupls3 a bus master would not be able to trigger an interrupt directly. The reflector moves a request from the bus master request bus over the response bus. It can then be

detected by the interrupt controller. This allows IPI (inter-processor interrupts) generated by software to be used.

Interrupt Logger

Logging of interrupts can be useful for the system. It is handy for debugging. The interrupt logger is a peripheral device that monitors the CPU response bus for interrupts (like the QIC) and logs all interrupts to a file in memory. The file can be subsequently processed for system management purposes.

Qupls3 Memory Management

Overview

The Qupls3 CPU uses both bounds and paging to manage memory. There is only a single dedicated page mapping table shared between all programs. To prevent two programs from accessing the same map entries bounds registers are used. Only the map entries within the bounds are accessible to the program. Because programs are limited to specific bounded address ranges the program loaded must relocate the programs to the appropriate areas.

Page Table

Qupls3 uses a non-hierarchical memory mapping table which is just a single level deep. Pages mapped are 16kB in size. The table is implemented in a dedicated BRAM memory with 49152 entries. 48k entries is enough to map 768MB of memory; the upper 256MB of memory are not mapped and are dedicated to kernel use.

Each PTE is 32-bits in size. The layout of an entry in the table is as follows:

PTE Format

31	30 28	27	26	25 24	23 22	21	20 18	17 0
V	Rgn_3	M	A	AVL_2	$CACHE_2$	U_1	RWX_3	$PPN_{17...0}$

Bounds

Bound registers are used to define ranges of addresses within the mapping table for a given program. Attempt to access memory outside of the allowed bounds will cause a memory protection fault. There are four bounds registers available which have typical usage for separate code, data, and stack areas.

QIC – Qupls3 Interrupt Controller

Overview

The Qupls3 system uses message-signaled interrupts (STMSI). QIC snoops the response bus going to the CPU core(s) for interrupt responses. Interrupt responses are stored in priority queues in the controller.

The Qupls3 interrupt controller presents an interrupt signal bus to the CPU core(s). The QIC may be used in a multi-CPU system as a shared interrupt controller. The QIC can guide the interrupt to the specified core(s). The QIC is a 64-bit slave I/O device.

System Usage

For the demo system there is just a single interrupt controller in the system. However, there may be up to 62 interrupt controllers in a system, numbered 1 to 62. Each interrupt controller may support up to 62 CPU cores, making the total number of CPU cores processing interrupts approximately 3800. QIC supports 63 different priority levels.

The QIC registers are located at an address determined by BAR0 in the configuration space. The interrupt table is located at a address determined by BAR1.

Priority Resolution

Interrupts have a fixed priority relationship with priority 63 having the highest priority and priority 1 the lowest. As interrupt messages are detected, they are placed in a queue according to their priority. (There are 63 small queues). The QIC sends the highest priority interrupt in the queues to the CPU. Periodically, once every 64 clock cycles, interrupt priorities are inverted.

Config Space

A 256-byte config space is supported. Most of the config space is unused. The only configuration is for the I/O address of the register set.

Regno	Width	R/W	Moniker	Description		
000	32	RO	REG_ID	Vendor and device ID		
004	32	R/W				
008	32	RO				
00C	32	R/W				
010	32	R/W	REG_BAR0	Base Address Register		

014	32	R/W	REG_BAR1	Base Address Register		
018	32	R/W	REG_BAR2	Base Address Register		
01C	32	R/W	REG_BAR3	Base Address Register		
020	32	R/W	REG_BAR4	Base Address Register		
024	32	R/W	REG_BAR5	Base Address Register		
028	32	R/W				
02C	32	RO		Subsystem ID		
030	32	R/W		Expansion ROM address		
034	32	RO				
038	32	R/W		Reserved		
03C	32	R/W		Interrupt		
040 to 0FF	32	R/W		Capabilities area		

REG_BAR0 defaults to $FEE20001 which is used to specify the address of the controller's registers in the I/O address space.

The controller will respond with a memory size request of 0MB (0xFFFFFFFF) when BAR0 is written with all ones. The controller contains its own dedicated memory and does not require memory allocated from the system.

Parameters

CFG_BUS defaults to zero
CFG_DEVICE defaults to six
CFG_FUNC defaults to zero
Config parameters must be set correctly. CFG device and vendors default to zero.

Registers

The QIC contains an interrupt vector table with a maximum of 2048 128-bit vectors available for each of four operating modes. (The number of vectors supported is parameterized). This vector table occupies 128kB of I/O space. An additional 522 registers are spread out through another 8k byte I/O region. All registers are 64-bit and only 64-bit accessible. The interrupt vector table is byte accessible.

Regno	Access	Moniker	Purpose
00	RW	UVTB	Base address for user interrupt vector table
08	RW	SVTB	Base address for supervisor interrupt vector table
10	RW	HVTB	Base address for hypervisor interrupt vector table

<table>
<tr><td>18</td><td>RW</td><td>MVTB</td><td>Base address for hypervisor machine vector table</td></tr>
<tr><td>20</td><td>RW</td><td>VTL</td><td>Vector table limit</td></tr>
<tr><td>28</td><td>RW</td><td>STAT</td><td><table><tr><td>Bit</td><td></td></tr><tr><td>0</td><td>Que full, set if any que is full, cleared by software if written with a zero</td></tr><tr><td>1</td><td>Set if stuck interrupt detected</td></tr><tr><td>2 to 62</td><td>reserved</td></tr><tr><td>63</td><td>Set if an interrupt is being requested</td></tr></table></td></tr>
<tr><td>30</td><td>R</td><td>QUEL</td><td>Top output of the priority queues, bits 0 to 63</td></tr>
<tr><td>38</td><td>R</td><td>QUEH</td><td>Top output of the priority queues, bits 64 to 127</td></tr>
<tr><td>40</td><td>R</td><td>EMP</td><td>Queue empty status, one bit for each queue, 1=empty</td></tr>
<tr><td>48</td><td>R</td><td>OVR</td><td>Queue overflow status, one bit for each queue, 1=overflowed</td></tr>
<tr><td>380</td><td>RW</td><td>GE</td><td>Bit 0 = global interrupt enable</td></tr>
<tr><td>390</td><td>RW</td><td>THRES</td><td>Interrupt threshold (0 to 63), IRQ priority must exceed this to be recognized.</td></tr>
<tr><td colspan="4">CPU affinity group table follows
There are 256 groups that may be set. The interrupt vector references one of these groups to determine which CPU cores should be notified of an interrupt.</td></tr>
<tr><td>800</td><td>RW</td><td>AFNx</td><td>CPU group, one bit for each CPU that should be notified</td></tr>
<tr><td>…</td><td>RW</td><td></td><td>More CPU groups</td></tr>
<tr><td>FF8</td><td>RW</td><td></td><td>Last CPU group</td></tr>
<tr><td colspan="4">Interrupt pending and enable tables follow. There are 128 64-bit entries for each table. This is enough to cover up to 2047 interrupts for each of four operating modes. User mode is entries 0 to 31, supervisor mode is entries 32 to 63, hypervisor 64 to 95 and machine 96 to 127.</td></tr>
<tr><td>1000</td><td>RW</td><td>IP</td><td>Interrupt enable bits</td></tr>
<tr><td>…</td><td></td><td></td><td>More IE bit registers</td></tr>
<tr><td>13F8</td><td>RW</td><td>IP</td><td></td></tr>
<tr><td>1400</td><td>RW</td><td>IE</td><td>Interrupt pending bits</td></tr>
<tr><td>…</td><td></td><td></td><td>More interrupt pending bits</td></tr>
<tr><td>17F8</td><td>RW</td><td>IE</td><td></td></tr>
</table>

Base Address Fields

The base address fields default to zero. The address fields are present should the controller be adapted to use main memory instead of dedicated BRAM. The address fields act as an index into the dedicated vector table for the location of the vectors for each operating mode.

CPU Affinity Group Table

This table is an array of groups of CPU cores that should be notified of an interrupt. The interrupt vector selects one of these groups for the group of CPUs to notify. Note that normally only a single CPU core will ultimately be selected to process the interrupt. If bit zero of the CPU group is set, then the interrupt will be broadcast to all CPU cores in the group.

Interrupt Enable Bits

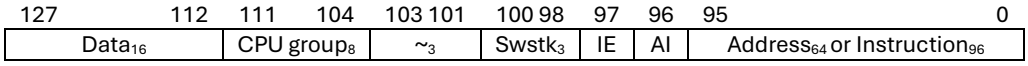
The interrupt enable bit array offers a fast way to enable or disable interrupts without having to update the interrupt vector table. Both the enable bit in the enable bit array and the enable bit in the vector table must be set for an interrupt to be enabled.

Interrupt Pending Bits

Writing a pending bit register clears the bit specified by the write data. If the MSB of the value written is a 1 then the corresponding interrupt is immediately triggered.

Interrupt Vector Table

The interrupt vector table has a default address of $FF…FECC0000 to $FF…FECDFFFF. This address may be changed by altering the BAR1 register in the config space. The interrupt vector table has four consecutive sections to it, one for each CPU operating mode. There are a maximum of 2048 vectors available for each mode. The vector format is as follows:



Field Description

AI: This field indicates that the vector contains an address (0) or an instruction (1)

IE: This field indicates if the interrupt is disabled (0) or enabled (1)

Swstk: This field contains the index of the software stack required to process the interrupt

CPU group: This field is an index into the CPU affinity group table which identifies which processor cores are candidates to receive the interrupt.

Data: This field is populated with data from the interrupt message.

QIT – Qupls3 Interval Timer

Overview

Many systems have at least one timer. The timing device may be built into the CPU, but it is frequently a separate component on its own. The programmable interval timer has many potential uses in the system. It can perform several different timing operations including pulse and waveform generation, along with measurements. While it is possible to manage timing events strictly through software it is quite challenging to perform in that manner. A hardware timer comes into play for the difficult to manage timing events. A hardware timer can supply precise timing. In the test system there are two groups of four timers. Timers are often grouped together in a single component. The QIT is a 64-bit peripheral. The QIT while powerful turns out to be one of the simpler peripherals in the system.

System Usage

One programmable timer component, which may include up 32 timers, is used to generate the system time slice interrupt and timing controls for system garbage collection. The second timer component is used to aid the paged memory management unit. There are free timing channels on the second timer component.

Each QIT is given an 8kB-byte memory range to respond to for I/O access. As is typical for I/O devices part of the address range is not decoded to conserve hardware.

PIT#1 is located at $FFFFFFFFFFEE40000 to $FFFFFFFFFFEE41FFF

PIT#2 is located at $FFFFFFFFFFEE50000 to $FFFFFFFFFFEE51FFF

Config Space

A 256-byte config space is supported. Most of the config space is unused. The only configuration is for the I/O address of the register set and the interrupt line used.

Regno	Width	R/W	Moniker	Description		
000	32	RO	REG_ID	Vendor and device ID		
004	32	R/W				
008	32	RO				
00C	32	R/W				
010	32	R/W	REG_BAR0	Base Address Register		
014	32	R/W	REG_BAR1	Base Address Register		
018	32	R/W	REG_BAR2	Base Address Register		

01C	32	R/W	REG_BAR3	Base Address Register		
020	32	R/W	REG_BAR4	Base Address Register		
024	32	R/W	REG_BAR5	Base Address Register		
028	32	R/W				
02C	32	RO		Subsystem ID		
030	32	R/W		Expansion ROM address		
034	32	RO				
038	32	R/W		Reserved		
03C	32	R/W		Interrupt		
040 to 0FF	32	R/W		Capabilities area		

REG_BAR0 defaults to $FEE40001 which is used to specify the address of the controller's registers in the I/O address space. Note for additional groups of timers the REG_BAR0 must be changed to point to a different I/O address range. Note the core uses only bits determined by the address mask in the address range comparison. It is assumed that a 8kB page is required for the device, matching the MMU page size.

The controller will respond with a mask of 0xFFFFFFFF when BAR0 is written with all ones.

Parameters

CFG_BUS defaults to zero
CFG_DEVICE defaults to four
CFG_FUNC defaults to zero
CFG_ADDR_MASK defaults to 0x00FF0000
CFG_IRQ_LINE defaults to 29
Config parameters must be set correctly. CFG device and vendors default to zero.

Parameters

NTIMER: This parameter controls the number of timers present. The default is eight. The maximum is 32.

BITS: This parameter controls the number of bits in the counters. The default is 48 bits. The maximum is 64.

PIT_ADDR: This parameter sets the I/O address that the QIT responds to. The default is $FEE40001.

PIT_ADDR_ALLOC: This parameter determines which bits of the address are significant during decoding. The default is $00FF0000 for an allocation of 64kB. To compute the address range allocation required, 'or' the value from the register with $FF000000, complement it then add 1.

Registers

The QIT has 134 registers addressed as 64-bit I/O cells. It occupies 2048 consecutive I/O locations. All registers are read-write except for the current counts which are read-only. All registers all 64-bit accessible; all 64 bits must be read or written. Values written to registers do not take effect until the synchronization register is written.

Note the core may be configured to implement fewer timers in which case timers that are not implemented will read as zero and ignore writes. The core may also be configured to support fewer bits per count register in which case the unimplemented bits will read as zero and ignore writes.

Regno	Access	Moniker	Purpose
00	R	CC0	Current Count
08	RW	MC0	Max count
10	RW	OT0	On Time
18	RW	CTRL0	Control
20 to 7F8	...	...	Groups of four registers for timer #1 to #63
800	RW	USTAT	Underflow status
808	RZW	SYNC	Synchronization register
810	RW	IE	Interrupt enable
818	RW	TMP	Temporary register
820	RO	OSTAT	Output status
828	RW	GATE	Gate register
830	RZW	GATEON	Gate on register
838	RZW	GATEOFF	Gate off register

Control Register

This register contains bits controlling the overall operation of the timer.

Bit		Purpose
0	LD	setting this bit will load max count into current count, this bit automatically resets to zero.
1	CE	count enable, if 1 counting will be enabled, if 0 counting is disabled and the current count register holds its value. On counter underflow this bit will be reset to zero causing the count to halt unless auto-reload is set.
2	AR	auto-reload, if 1 the max count will automatically be reloaded into the current count register when it underflows.
3	XC	external clock, if 1 the counter is clocked by an external clock source. The external clock source must be of lower frequency than the clock supplied to the PIT. The PIT contains edge detectors on the external clock source and counting occurs on the detection of a positive edge on the clock source. This bit is forced to 0 for timers 4 to 31.
4	GE	gating enable, if 1 an external gate signal will also be required to be active high for the counter to count, otherwise if 0 the external gate is ignored. Gating the counter using the external gate may allow pulse-width measurement. This bit is forced to 0 for timers 4 to 31.
5 to 63	~	not used, reserved

Current Count

This register reflects the current count value for the timer. The value in this register will change by counting downwards whenever a count signal is active. The current count may be automatically reloaded at underflow if the auto reload bit (bit #2) of the control byte is set. The current count may also be force loaded to the max count by setting the load bit (bit #0) of the counter control byte.

Max Count

This register holds onto the maximum count for the timer. It is loaded by software and otherwise does not change. When the counter underflows the current count may be automatically reloaded from the max count register.

On Time

The on-time register determines the output pulse width of the timer. The timer output is low until the on-time value is reached, at which point the timer output switches high. The timer output remains high until the counter reaches zero at which point the timer output is reset back to zero. So, the on time reflects the length

of time the timer output is high. The timer output is low for max count minus the on-time clock cycles.

Underflow Status

The underflow status register contains a record of which timers underflowed.

Writing the underflow register clears the underflows and disable further interrupts where bits are set in the incoming data. Interrupt processing should read the underflow register to determine which timers underflowed, then write back the value to the underflow register.

Synchronization Register

The synchronization register allows all the timers to be updated simultaneously. Values written to timer registers do not take effect until the synchronization register is written. The synchronization register must be written with a '1' bit in the bit position corresponding to the timer to update. For instance, writing all one's to the sync register will cause all timers to be updated. The synchronization register is write-only and reads as zero.

Interrupt Enable Register

Each bit of the interrupt enable register enables the interrupt for the corresponding timer. Interrupts must also be globally enabled by the interrupt enable bit in the config space for interrupts to occur. A '1' bit enables the interrupt, a '0' bit value disables it.

Temporary Register

This is merely a register that may be used to hold values temporarily.

Output Status

The output status register reflects the current status of the timers output (high or low). This register is read-only.

Gate Register

The internal gate register is used to temporarily halt or resume counting for the timer corresponding to the bit position of this register. Writing a value to this register will turn on all timers where there is a '1' bit in the value and turn off all timers where there is a '0' bit in the value.

Gate On Register

The internal gate 'on' register is used to resume counting for the timer corresponding to the bit position of this register. Writing a value to this register will turn on all timers

where there is a ‘1’ bit in the value. Where there is a ‘0’ in the value the timer will not be affected. This register reads as zero.

Gate Off Register

The internal gate ‘off’ register is used to halt counting for the timer corresponding to the bit position of this register. Writing a value to this register will turn off all timers where there is a ‘1’ bit in the value. Where there is a ‘0’ in the value the timer will not be affected. This register reads as zero.

Programming

The PIT is a memory mapped i/o device. The PIT is programmed using 64-bit load and store instructions (LDO and STO). Byte loads and stores (LDB, STB) may be used for control register access. It must reside in the non-cached address space of the system.

Interrupts

The core is configured use interrupt signal #29 by default. This may be changed with the CFG_IRQ_LINE parameter. Interrupts may be globally disabled by writing the interrupt disable bit in the config space with a ‘1’. Individual interrupts may be enabled or disabled by the setting of the interrupt enable register in the I/O space.

FTA Bus

Overview

The FTA bus is an asynchronous bus meaning it does not wait for responses before beginning the next bus cycle. It is a request and response bus. Requests are outgoing from a bus master and incoming to a bus slave. Responses are output by a bus slave and input by a bus master. FTA bus includes standard signals for address, data, and control. These signals should be like those found on many other busses.

Bus Tags

The bus has tagged transactions; there is an id tag associated with each bus transaction. The id tag contains identifiers for the core, channel, and transaction. The core is a core number for a multi-core CPU. Channel selects a particular channel in the core which may for instance be a data channel or an instruction channel. Finally, the transaction id identifies the specific transaction. Incoming responses are matched against transactions that were outgoing. For instance, a bus master may issue a burst request for four bus transactions to fill a cache line. Each transaction will have an id associated with it. When the slave receives the transactions it sends back responses for each of the four requests with ids that match those in the request. The slave does not necessarily send back responses in the same order. Transaction requests from the master may not arrive in order.

*An id tag of all zeros is illegal – it represents the bus available state.

Single Cycle

The bus operates on a single cycle basis. Transaction requests and responses are routed through the soc interconnect network as the bus is available and are present for only a single clock cycle. Bus bridges may buffer the transactions for a short period of time. Generally, requests going out from masters do not need buffering as access to the bus will have been arbitrated before the bus cycle begins. Responses coming back from slaves may need to be buffered as two slaves may respond at the same time. Slaves are not required to arbitrate for the bus.

Retry

If the bus is unavailable the retry response signal is asserted to the master. The master must retry the transaction.

Signal Description

Following is a signal description for requests and responses for a 128-bit data version of the bus. Signal values have been chosen so that a value of zero represents a bus idle state. If nothing is on the bus it will be all zeros.

Requests

Signal	Width	Description	
Om	2	Operating mode	
Cmd	5	Command for bus controller or memory controller	
Bte	3	Burst type	
Cti	3	Cycle type	
Blen	6	Burst length -1 (0=1 to 63=64)	
sz	4	Transfer size	
Segment	3	Code, data, or stack	
Cyc	1	Bus cycle is valid	
We	1	Write enable	
Asid	16	Address space id	
Vadr	32/64	Virtual address	
Padr	32/64	Physical address	
Sel	16	Byte lane selects	
Data1	128	First data item	
Data2	128	Second data item (for AMO operations)	
Tid	13	Transaction id	
Csr	1	Clear or set address reservation	
Pl	8	Privilege level	
Pri	4	Transaction priority (higher is better)	
Cache	4	Transaction cacheability	

Responses

Signal	Width	Description	
Tid	13	Transaction id	
Stall	1	Stall pipeline	
Next	1	Advance to next transaction	

Ack	1	Request acknowledgement (data is available)	
Rty	1	Retry transaction	
Err	3	Error code	
Pri	4	Transaction priority	
Adr	32/64	Physical address	
Dat	32/64/128/256	Response data	

Om

Operating mode, this corresponds to the operating mode of the CPU. Some devices are limited to specific modes.

Cmd

Command for memory controller. This is how the memory controller knows what to do with the data.

Oridinal		
0	CMD_NONE	No command
1	CMD_LOAD	Perform a sign extended data load operation
2	CMD_LOADZ	Perform a zero extended data load operation
3	CMD_STORE	Perform a data store operation
4	CMD_STOREPTR	Perform a pointer store operation
7	CMD_LEA	Load the effective address
10	CMD_DCACHE_LOAD	Perform load operation intended for data cache
11	CMD_ICACHE_LOAD	Perform load operation intended for instruction cache
13	CMD_CACHE	Issue a cache control command
16	CMD_SWAP	AMO swap operation
18	CMD_MIN	AMO min operation
19	CMD_MAX	AMO max operation
20	CMD_ADD	AMO add operation
22	CMD_ASL	AMO left shift operation
23	CMD_LSR	AMO right shift operation
24	CMD_AND	AMO and operation
25	CMD_OR	AMO or operation
26	CMD_EOR	AMO exclusive or operation
28	CMD_MINU	AMO unsigned minimum operation
29	CMD_MAXU	AMO unsigned maximum operation
31	CMD_CAS	AMO compare and swap
Others		reserved

BTE

Burst type extension.

Ordinal	
0	Linear
1	Wrap 4
2	Wrap 8
3	Wrap 16
4	Wrap 32
5	Wrap 64
6	Wrap 128
7	reserved

CTI

Cycle Type Indicator

Ordinal		Comment
0	Classic	
1	fixed	Constant data address
2	Incr	Incrementing data address
3	erc	Record errors on write
4	Irqa	Interrupt acknowledge
7	Eob	End of burst
others		reserved

Normally write cycles do not send a response back to the master. The ERC cycle type indicates that the master wants a response back from a write operation.

Blen

Burst length, this is the number of transactions in the burst minus one. There is a maximum of 64 transactions. With a 128-bit bus this is 1024 bytes of data.

Sz

Transfer size.

Ordinal		Transfer size
0	Nul	Nothing is transferred
1	Byt	A single byte
2	Wyde	Two bytes

3	Tetra	Four bytes
4	Penta	Five bytes
5	Octa	Eight bytes
6	Hexi	Sixteen bytes
10	vect	A vector 64 bytes (512 bit bus)
Others		Reserved

Segment

The memory segment associated with the transfer.

Ordinal	
0	data
6	stack
7	code
others	reserved

TID

Transaction ID. This is made up of three fields.

Size	Use
6	Core number
3	Channel
4	Tran id

Cache

Cache-ability of transaction. A transaction may be non-cacheable meaning as it progresses through the cache hierarchy it does not store data in the cache. It only stores data when it reaches the final memory destination.

Ordinal		
0	NC_NB	Non cacheable, non bufferable
1	NON_CACHEABLE	
2	CACHEABLE_NB	Cacheable, non bufferable
3	CACHEABLE	
8	WT_NO_ALLOCATE	Write-through without allocating
9	WT_READ_ALLOCATE	

10	WT_WRITE_ALLOCATE	
11	WT_READWRITE_ALLOCATE	
12	WB_NO_ALLOCATE	Write-back without allocating
13	WB_READ_ALLOCATE	
14	WB_WRITE_ALLOCATE	
15	WB_READWRITE_ALLOCATE	

Message Signaled Interrupts

FTA bus provides for message signaled interrupts. A MSI interrupt transfers the required information to an interrupt controller without needing a request for it. This trims cycle time off an interrupt request. The interrupt controller constantly snoops the CPU response bus for IRQ requests.

Up to 62 interrupt controllers may be targeted to process interrupts messages. The interrupt table located in the controller specifies which of 62 target CPU cores to notify of the interrupt. Therefore about 3800 CPU cores may be easily used for interrupt processing.

There is a response code ('IRQ') on the response bus to support message signaled interrupts. A slave may place an IRQ message on a response bus (the 'err' field) to interrupt the master.

Signal	Description
ack	This signal indicates a valid response; should be high for MSI
err	Value = IRQ
dat	Interrupt message data. Typically 32-bits
tid	The coreno (upper 6 bits) should reflect the target core servicing the interrupt. This is an interrupt controller number. The interrupt priority is in the lower 6 -bits.
adr	The 'adr' field of the response indicates the bus/device/function generating the interrupt.

Glossary

ABI

An acronym for application binary interface. An ABI is a description of the interface between software and hardware, or between software modules. It includes things like the expected register usage by the compiler. Some registers hardware has specific requirements for are noted in the ABI, for instance r0 may always be zero or it may be a usable register. The stack pointer may need to be a specific register. A good ABI is an aid to guaranteeing that software works when coming from multiple sources.

AMO

AMO stands for atomic memory operation. An atomic memory operation typically reads then writes to memory in a fashion that may not be interrupted by another processor. Some examples of AMO operations are swap, add, and, and or. AMO operations are typically passed from the CPU to the memory controller and the memory controller performs the operation.

Assembler

A program that translates mnemonics and operands into machine code OR a low-level language used by programmers to conveniently translate programs into machine code. Compilers are often capable of generating assembler code as an output.

ATC

ATC stands for address translation cache. This buffer is used to cache address translations for fast memory access in a system with an mmu capable of performing address translations. The address translation cache is more commonly known as the TLB.

Base Pointer

An alternate term for frame pointer. The frame or base pointer is used by high-level languages to access variables on the stack.

Burst Access

A burst access is several bus accesses that occur rapidly in a row in a known sequence. If hardware supports burst access the cycle time for access to the device is drastically reduced. For instance, dynamic RAM memory access is fast for sequential burst access, and somewhat slower for random access.

BTB

An acronym for Branch Target Buffer. The branch target buffer is used to improve the performance of a processing core. The BTB is a table that stores the branch target from previously executed branch instructions. A typical table may contain 1024 entries. The table is typically indexed by part of the branch address. Since the target address of a branch type instruction may not be known at fetch time, the address is speculated to be the address in the branch target buffer. This allows the machine to fetch instructions in a continuous fashion without pipeline bubbles. In many cases the calculated branch address from a previously executed instruction remains the same the next time the same instruction is executed. If the address from the BTB turns out to be incorrect, then the machine will have to flush the instruction queue or pipeline and begin fetching instructions from the correct address.

Card Memory

A card memory is a memory reserved to record the location of pointer stores in a garbage collection system. The card memory is much smaller than main memory; there may be card memory entry for a block of main memory addresses. Card memory covers memory in 128 to 512-byte sized blocks. Usually, a byte is dedicated to record the pointer store status even though a bit would be adequate, for performance reasons. The location of card memory to update is found by shifting the pointer value to the right some number of bits (7 to 9 bits) and then adding the base address of the table. The update to the card memory needs to be done with interrupts disabled.

Commit

As in commit stage of processor. This is the stage where the processor is dedicated or committed to performing the operation. There are no prior outstanding exceptions or flow control changes to prevent the instruction from executing. The instruction may execute in the commit stage, but registers and memory are not updated until the retire stage of the processor.

Decimal Floating Point

Floating point numbers encoded specially to allow processing as decimal numbers. Decimal floating point allows processing every-day decimal numbers rounding in the same manner as would be done by hand.

Decode

The stage in a processor where instructions are decoded or broken up into simpler control signals. For instance, there is often a register file write signal that must be decoded from instructions that update the register file.

Diadic

As in diadic instruction. An instruction with two operands.

DUT

An acronym for Design Under Test.

Endian

Computing machines are often referred to as big endian or little endian. The endian of the machine has to do with the order bits and bytes are labeled. Little endian machines label bits from right to left with the lowest bit at the right. Big endian machines label bits from left to right with the lowest numbered bit at the left.

FIFO

An acronym standing for 'first-in first-out'. Fifo memories are used to aid data transfer when the rate of data exchange may have momentary differences. Usually when fifos transfer data the average data rate for input and output is the same. Data is stored in a buffer in order then retrieved from the buffer in order. Uarts often contain fifos.

FPGA

An acronym for Field Programmable Gate Array. FPGA's consist of a large number of small RAM tables, flip-flops, and other logic. These are all connected with a programmable connection network. FPGA's are 'in the field' programmable, and usually re-programmable. An FPGA's re-programmability is typically RAM based. They are often used with configuration PROM's so they may be loaded to perform specific functions.

Floating Point

A means of encoding numbers into binary code to allow processing. Floating point numbers have a range within which numbers may be processed, outside of this range the number will be marked as infinity or zero. The range is usually large enough that it is not a concern for most programs.

Frame Pointer

A pointer to the current working area on the stack for a function. Local variables and parameters may be accessed relative to the frame pointer. As a program progresses a series of "frames" may build up on the stack. In many cases the frame pointer may be omitted, and the stack pointer used for references instead. Often a register from the general register file is used as a frame pointer.

HDL

An acronym that stands for 'Hardware Description Language'. A hardware description language is used to describe hardware constructs at a high level.

HLL

An acronym that stands for "High Level Language"

Instruction Bundle

A group of instructions. It is sometimes required to group instructions together into bundle. For instance, all instructions in a bundle may be executed simultaneously on a processor as a unit. Instructions may also need to be grouped if they are oddball in size for example 41 bits, so that they can be fit evenly into memory. Typically, a bundle has some bits that are global to the bundle, such as template bits, in addition to the encoded instructions.

Instruction Pointers

A processor register dedicated to addressing instructions in memory. It is also often called a program counter. The program counter got its name because it usually increments (or counts) automatically after an instruction is fetched. In early machines in some rare cases the program counter did not count in a sequential binary fashion, but instead used other forms of a counter such as a grey counter or linear feedback shift register. In some

machines the program counter addresses bundles of instructions rather than individual instructions. This is common with some stack machines where multiple instructions are packed into a memory word.

Instruction Prefix

An instruction prefix applies to the following instruction to modify its operation. An instruction prefix may be used to add more bits to a following immediate constant, or to add additional register fields for the instruction. The prefix essentially extends the number of bits available to encode instructions. An instruction prefix usually locks out interrupts between the prefix and following instruction.

Instruction Modifier

An instruction modifier is similar to an instruction prefix except that the modifier may apply to multiple following instructions.

ISA

An acronym for Instruction Set Architecture. The group of instructions that an architecture supports. ISA's are sometimes categorized at extreme edges as RISC or CISC. RTF64 falls somewhere in between with features of both RISC and CISC architectures.

IPI

An acronym for Inter-Processor-Interrupt. An inter-processor interrupt is an interrupt sent from one processor to another.

JIT

An acronym standing for Just-In-Time. JIT compilers typically compile segments of a program just before usage, and hence are called JIT compilers.

Keyed Memory

A memory system that has a key associated with each page to protect access to the page. A process must have a matching key in its key list in order to access the memory page. The key is often 20 bits or larger. Keys for pages are usually cached in the processor for performance reasons. The key may be part of the paging tables.

Linear Address

A linear address is the resulting address from a virtual address after segmentation has been applied.

Machine Code

A code that the processing machine is able execute. Machine code is lowest form of code used for processing and is not usually delt with by programmers except in debugging cases. While it is possible to assemble machine code by hand usually a tool called an assembler is used for this purpose.

Milli-code

A short sequence of code that may be used to emulate a higher-level instruction. For instance, a garbage collection write barrier might be written as milli-code. Milli-code may use an alternate link register to return to obtain better performance.

Monadic

An instruction with just a single operand.

MSI

An acronym for Message Signaled Interrupt. A message signaled interrupt is an interrupt processed using a message sent to a CPU using in-band resources.

Opcode

A short form for operation code, a code that determines what operation the processor is going to perform. Instructions are typically made up of opcodes and operands.

Operand

The data that an opcode operates on, or the result produced by the operation. Operands are often located in registers. Inputs to an operation are referred to as source operands, the result of an operation is a destination operand.

Physical Address

A physical address is the final address seen by the memory system after both segmentation and paging have been applied to a virtual address. One can think of a physical address as one that is “physically” wired to the memory.

Physical Memory Attributes (PMA)

Memory usually has several characteristics associated with it. In the memory system there may be several different types of memory, rom, static ram, dynamic ram, eeprom, memory mapped I/O devices, and others. Each type of memory device is likely to have different characteristics. These characteristics are called the physical memory attributes. Physical memory attributes are associated with address ranges that the memory is located in. There may be a hardware unit dedicated to verifying software is adhering to the attributes associated with the memory range. The hardware unit is called a physical memory attributes checker (PMA checker).

PIC

An acronym for Position Independent Code. Position independent code is code that will execute properly no matter where it is located. The code may be moved in memory without needing to be modified.

Posits

An alternate representation of numbers.

Program Counter

A processor register dedicated to addressing instructions in memory. It is also often and perhaps more aptly called an instruction pointer. The program counter got its name because it usually increments (or counts) automatically after an instruction is fetched. In early machines in some rare cases the program counter did not count in a sequential binary fashion, but instead used other forms of a counter such as a grey counter or linear feedback shift register. In some machines the program counter addresses bundles of instructions rather than individual instructions. This is common with some stack machines where multiple instructions are packed into a memory word.

RAT

Anacronym for Register Alias Table. The RAT stores mappings of architectural registers to physical registers.

Retire

As in retire an instruction. This is the stage in processor in which the machine state is updated. Updates include the register file and memory. Buffers used for instruction storage are freed.

ROB

An acronym for ReOrder Buffer. The re-order buffer allows instructions to execute out of order yet update the machine's state in order by tracking instruction state and variables. In FT64 the re-order buffer is a circular queue with a head and tail pointers. Instructions at the head are committed if done to the machine's state then the head advanced. New instructions are queued at the buffer's tail as long as there is room in the queue. Instructions in the queue may be processed out of the order that they entered the queue in depending on the availability of resources (register values and functional units).

RSB

An acronym that stands for return stack buffer. A buffer of addresses used to predict the return address which increases processor performance. The RSB is usually small, typically 16 entries. When a return instruction is detected at time of fetch the RSB is accessed to determine the address of the next instruction to fetch. Predicting the return address allows the processing core to continuously fetch instructions in a speculative fashion without bubbles in the pipeline. The return address in the RSB may turn out to be detected as incorrect during execution of the return instruction, in which case the pipeline or instruction queue will need to be flushed and instructions fetched from the proper address.

SIMD

An acronym that stands for 'Single Instruction Multiple Data'. SIMD instructions are usually implemented with extra wide registers. The registers contain multiple data items, such as a 128-bit register containing four 32-bit numbers. The same instruction is applied to all the data items in the register

at the same time. For some applications SIMD instructions can enhance performance considerably.

Stack Pointer

A processor register dedicated to addressing stack memory. Sometimes this register is assigned by convention from the general register pool. This register may also sometimes index into a small dedicated stack memory that is not part of the main memory system. Sometimes machines have multiple stack pointers for different purposes, but they all work on the idea of a stack. For instance, in Forth machines there are typically two stacks, one for data and one for return addresses.

Telescopic Memory

A memory system composed of layers where each layer contains simplified data from the topmost layer downwards. At the topmost layer data is represented verbatim. At the bottom layer there may be only a single bit to represent the presence of data. Each layer of the telescopic memory uses far less memory than the layer above. A telescopic memory could be used in garbage collection systems. Normally however the extra overhead of updating multiple layers of memory is not warranted.

TLB

TLB stands for translation look-aside buffer. This buffer is used to store address translations for fast memory access in a system with an mmu capable of performing address translations.

Trace Memory

A memory that traces instructions or data. As instructions are executed the address of the executing instruction is stored in a trace memory. The trace memory may then be dumped to allow debugging of software. The trace memory may compress the storage of addresses by storing branch status (taken or not taken) for consecutive branches rather than storing all addresses. It typically requires only a single bit to store the branch status. However, even when branches are traced, periodically the entire address of the program executing is stored. Often trace buffers support tracing thousands of instructions.

Triadic

An instruction with three operands.

Vector Chaining

Vector chaining is a form of pipelining used with vector processors. A CPU that supports vector chaining can begin processing additional vector instructions before previous ones are complete. The processing of vector instructions is overlapped.

Vector Length (VL register)

The vector length register controls the maximum number of elements of a vector that are processed. The vector length register may not be set to a value greater than the number of elements supported by hardware. Vector registers often contain more elements than are required by program code. It would be wasteful to process all elements when only a few are needed. To improve the processing performance only the elements up to the vector length are examined.

Vector Mask (VM)

A vector mask is used to restrict which elements of a vector are processed during a vector operation. A one bit in a mask register enables the processing for that element, a zero bit disables it. The mask register is commonly set using a vector set operation.

Virtual Address

The address before segmentation and paging has been applied. This is the primary type of address a program will work with. Different programs may use the same virtual address range without being concerned about data being overwritten by another program. Although the virtual address may be the same the final physical addresses used will be different.

Writeback

A stage in a pipelined processing core where the machine state is updated. Values are ‘written back’ to the register file.

Miscellaneous

Reference Material

Below is a short list of some of the reading material the author has studied. The author has downloaded a fair number of documents on computer architecture from the web. Too many to list.

Modern Processor Design Fundamentals of Superscalar Processors by John Paul Shen, Mikko H. Lipasti. Waveland Press, Inc.

Computer Architecture A Quantitative Approach, Second Edition, by John L Hennessy & David Patterson, published by Morgan Kaufman Publishers, Inc. San Franciso, California is a good book on computer architecture. There is a newer edition of the book available.

Memory Systems Cache, DRAM, Disk by Bruce Jacob, Spencer W. Ng., David T. Wang, Samuel Rodriguez, Morgan Kaufman Publishers

PowerPC Microprocessor Developer's Guide, SAMS publishing. 201 West 103rd Street, Indianapolis, Indiana, 46290

80386/80486 Programming Guide by Ross P. Nelson, Microsoft Press

Programming the 286, C. Vieillefond, SYBEX, 2021 Challenger Drive #100, Alameda, CA 94501

Tech. Report UMD-SCA-2000-02 ENEE 446: Digital Computer Design — An Out-of-Order RiSC-16

Programming the 65C816, David Eyes and Ron Lichty, Western Design Centre Inc.

Microprocessor Manuals from Motorola, and Intel,

The SPARC Architecture Manual Version 8, SPARC International Inc, 535 Middlefield Road. Suite210 Menlo Park California, CA 94025

The SPARC Architecture Manual Version 9, SPARC International Inc, Sab Jose California, PTR Prentice Hall, Englewood Cliffs, New Jersey, 07632

The MMIX processor: 5

RISCV 2.0 Spec, Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovi´c CS Division, EECS Department, University of California, Berkeley {waterman|yunsup|pattrsn|krste}@eecs.berkeley.edu

The Garbage Collection Handbook, Richard Jones, Antony Hosking, Eliot Moss published by CRC Press 2012

RISC-V Cryptography Extensions Volume I Scalar & Entropy Source Instructions See github.com/riscv/riscv-crypto for more information.

Trademarks

WISHBONE Compatibility Datasheet

The Qupls3 core now uses the FTA bus which is not compatible with WISHBONE. Many signals serve a similar function to those on the WISHBONE bus so they are listed here. A bus bridge is required to interface FTA bus to WISHBONE as WISHBONE is a synchronous bus and FTA is asynchronous.

WISHBONE Datasheet WISHBONE SoC Architecture Specification, Revision B.3		
Description:	Specifications:	
General Description:	Central processing unit (CPU core)	
Supported Cycles:	MASTER, READ / WRITE MASTER, READ-MODIFY-WRITE MASTER, BLOCK READ / WRITE, BURST READ (FIXED ADDRESS)	
Data port, size: Data port, granularity: Data port, maximum operand size: Data transfer ordering: Data transfer sequencing	128 bit 8 bit 128 bit Little Endian any (undefined)	
Clock frequency constraints:	tm_clk_i must be >= 10MHz	
Supported signal list and cross reference to equivalent WISHBONE signals	Signal Name: Resp.ack_i Req.adr_o(31:0) clk_i resp.dat(127:0) req.dat(127:0) req.cyc req.stb req.wr	WISHBONE Equiv. ACK_I ADR_O() CLK_I DAT_I() DAT_O() CYC_O STB_O WE_O

	req.sel(7:0) req.cti(2:0) req.bte(1:0)	SEL_O CTI_O BTE_O
Special Requirements:		