StarkCPU

Robert Finch

Table of Contents

Overview 15
Motivation 15
History 15
Features of Qupls3 16
Programming Model 17
Register File – Visible Registers 17
Register File – Hidden Registers 17
Physical Registers 18
Code Address (Branch) Registers 18
Condition Registers – CR0 to CR7 18
TCBA 19
MC0 to MC3 19
XH – Exception Handler Address 20
Special Purpose Registers 21
[U/S/H/M]_SR - Status Register (CSR 0x?004) 21
[U/S/H/M]_IE (0x?004) 22
[U/S/H/M]_CAUSE (CSR- 0x?006) 22
U_FPCSR (CSR 0x0009) Floating Point Status and Control Register 22
[U/S/H/M]_XS (CSR 0x?00A) Extension State Register 23
[U/S/H/M]_EVEC – (CSR 0x?038 to 0x?03C) 24
[U/S/H/M]_SCRATCH – CSR 0x?041 24
S_ASID (CSR 0x101F) 24
M_CORENO (CSR 0x3001) 25
M_TICK (CSR 0x3002) 25
M_SEED (CSR 0x3003) 25
M_TCBA (CSR 0x3005) 25
M_BADADDR (CSR 0x3007) 26
M_BAD_INSTR (CSR 0x300B) 26

M_SEMA (CSR 0x300C) 26
M_TVEC – CSR 0x3030 to 0x3034 26
M_SR_TOS (CSR 0x3080) 26
M_IOS – IO Select Register (CSR 0x3100) 26
M_CFGS – Configuration Space Register (CSR 0x3101) 27
M_EPC_TOS (CSR 0x3108) 27
Operating Modes 27
Exceptions 29
External Interrupts 29
Effect on Operating Mode 29
Exception Stack 29
Vector Table 29
Breakpoint Fault (0) 31
Single Step Breakpoint (1) 31
Bus Error Fault (2) 31
Address Error (3) 31
Unimplemented Instruction Fault (4) 31
Page Fault (6) 31
Instruction Trace Fault (7) 31
Stack Canary Fault (8) 31
Abort (9) 31
Interrupt (10) 31
Reset Vector (12) 31
Alternate Cause (13) 32
User / App Environment Call (16) 32
Supervisor Environment Call (17) 32
Hypervisor Environment Call (18) 32
Machine Environment Call (19) 32
TRAP (20) 32

Reset 32
Precision 32
Task Support 33
Task Control Block / Context Block Layout 33
Instruction Set 34
Overview 34
Code Alignment 34
Root Opcode 34
Destination Register Spec 34
Source Register Spec 34
Constant Field Spec 35
Table of Constant Location Bits – L, LX_2 35
Instruction Format Tables 36
Compare Instruction Format 36
Branch Instruction Formats 36
Load and Store Instruction Formats 36
ALU Instruction Formats 37
FPU Instruction Formats 38
Shift Instruction Formats 38
CSR Instruction Formats 38
BRK / RFI Instruction Formats 39
TRAP Instruction Formats 39
Macro Instruction Formats 39
MOVE Instruction Format 40
Instruction Pres/Postfixes and Modifiers Instruction Formats 40
Condition Register Manipulation Instruction Formats 40
Table of Root Opcodes 41
Instruction Descriptions 42
Comparison Instructions 42

CMP – Compare 42
CMPA – Compare Addresses (unsigned) 43
FCMP – Compare Floats 44
Arithmetic Instructions 45
ABS[.] – Absolute Value 45
ADD[.] – Add 46
ADB[.] - Add Immediate to Branch Register 47
~~ADC[.] – Add with Carry (deprecated)~~ 48
CNTLO[.] – Count Leading Ones 49
CNTLZ[.] – Count Leading Zeros 49
CNTPOP[.] – Count Population 50
CNTTZ[.] – Count Trailing Zeros 50
CSR[.] – Control and Special Registers Operations 51
LOADA[.] – Load Address 52
~~SBC[.] – Subtract with Carry (deprecated)~~ 53
SUBF[.] – Subtract From 54
Multiply and Divide 55
DIV[.] – Signed Division 55
DIVA[.] – Address Division 56
MUL[.] – Multiply 57
MULA[.] – Multiply for Addressing 58
Shift and Rotate 59
EXT[.] – Extract Bit Field 59
EXTZ[.] – Extract Bit Field and Zero extend 60
SLL[.] –Shift Left Logical 61
SRA[.] –Shift Right Arithmetic 62
SRL[.] –Shift Right Logical 63
Logical Operations 64
AND[.] – Bitwise And 64

ANDC[.] – Bitwise And with Complement 65
NAND[.] – Bitwise And then invert 66
NOR[.] – Bitwise Or then invert 67
OR[.] – Bitwise Or 68
ORC[.] – Bitwise Or with Complement 69
XNOR[.] – Bitwise Exclusive Or then invert 70
XOR[.] – Bitwise Exclusive Or 71
CHK – Check Register Against Bounds 72
Data Movement 74
BMAP[.] – Byte Map 74
CMOVZ[.]– Conditional Move if Zero to Register 75
CMOVNZ[.]– Conditional Move in Non-zero to Register 76
MOVE[.] / MOVEA[.] / MOVSZ[.] / MOVZX[.] – Move Register to Register 77
Floating-Point Instructions 79
FABS[.] – Absolute Value 79
FADD[.] –Float Addition 80
FDIV[.] – Float Division 84
FMA –Float Multiply and Add 85
FMUL[.] –Float Multiplication 86
FNEG[.] – Negative Value 87
FSGNJ[.] – Float Sign Inject 89
FSGNJN[.] – Float Sign Inject and Negate 90
Load / Store Instructions 94
Overview 94
Addressing Modes 94
AMOADD[.] – AMO Addition 95
AMOAND[.] – AMO Bitwise 'And' 96
AMOOR[.] – AMO Bitwise 'Or' 97
AMOSWAP[.] – AMO Swap 98

CMPSWAP[.] – Compare and Swap 99
LDB[.] Rn, <ea> - Load Byte 100
LDBZ[.] Rn, <ea> - Load Byte and Zero Extend 100
LDT[.] Rn, <ea> - Load Tetra 101
LDTZ[.] Rn, <ea> - Load Tetra and Zero Extend 101
LDW[.] Rn, <ea> - Load Wyde 102
LDWZ[.] Rn, <ea> - Load Wyde and Zero Extend 102
LOAD[.] Rn, <ea> - Load 103
STB Rn, <ea> - Store Byte 104
STBI Rn, <ea> - Store Byte Immediate 104
STORE Rn, <ea> - Store Register 105
STOREI N, <ea> - Store Immediate 105
STPTR Rn, <ea> - Store Pointer 106
STT Rn, <ea> - Store Tetra 107
STTI Rn, <ea> - Store Tetra Immediate 107
STW Rn, <ea> - Store Wyde 108
STWI Rn, <ea> - Store Wyde Immediate 108
Condition Register Instructions 109
CLC – Clear Carry 109
CLV – Clear Overflow 110
CRAND – Bit And 111
CRANDC – Bit And with Complement 112
CROR – Bit Or 113
CRXOR – Bit Exclusive Or 114
SEC – Set Carry 115
SEV – Set Overflow 115
Branch / Flow Control Instructions 116
Overview 116
Conditional Branch Format 116

Branch Target 117

Decrementing Branches 118

Unconditional Branches 118

B – Branch Always 119

BAND –Branch if And 120

BANDL –Branch if And and Link 120

BCC –Branch if Carry Clear 121

BCCL –Branch if Carry Clear and Link 121

BCS –Branch if Carry Set 122

BCSL –Branch if Carry Set and Link 122

BEQ –Branch if Equal 123

BEQL –Branch if Equal and Link 123

BGE –Branch if Greater Than or Equal 123

BGEL –Branch if Less Than or Equal and Link 124

BGT –Branch if Greater Than 124

BGTL –Branch if Less Than or Equal and Link 125

BL – Branch and Link 126

BLR – Branch to Link Register 127

BLRI – Branch to Link Register and Interrupt 128

BLRL – Branch to Link Register and Link 129

BLE –Branch if Less Than or Equal 130

BLEL –Branch if Less Than or Equal and Link 130

BLT –Branch if Less Than 131

BLTL –Branch if Less Than and Link 131

BNAND –Branch if Nand 132

BNANDL –Branch if Nand and Link 132

BNOR –Branch if Nor 133

BNORL –Branch if Nor and Link 133

BNE –Branch if Not Equal 134

BNEL –Branch if Not Equal and Link 134
BOR –Branch if Or 135
BORL –Branch if Or and Link 135
BT –Branch Table 136
BTL –Branch Table and Link 136
BVS –Branch if Overflow Set 137
BVSL –Branch if Overflow Set and Link 137
NOP – No Operation 138
PEQ –Predicate if Equal 138
PNE –Predicate if Not Equal 139
System Instructions 142
BRK – Break 142
REX – Redirect Exception 143
RFI – Return from Interrupt or Exception 144
ECALL – Environment / Escalating Call 145
ERET – Environment Return 146
TRAPcc – Trap if Condition Met 147
VIRT2PHYS[.] – Convert Virtual Address to Physical Address 149
Macro Instructions 150
ENTER – Enter Routine 150
EXIT – Exit Routine 152
POP – Pop Registers from Stack 153
PUSH – Push Registers on Stack 154
Pre/Postfixes and Modifiers 155
ACARRY Modifier 155
ATOM Modifier 157
CARRY Modifier 158
QEXT Prefix 159
FREGS Prefix 159

PFX[ABCD] – A/B/C/D Immediate Postfix 160
~~PRED Modifier (deprecated)~~ 161
MPU Hardware 163
Hardware Description 163
Caches 163
Overview 163
Instructions 163
L1 Instruction Cache 163
Fetch Rate 164
Data Cache 165
Cache Enables 165
Cache Validation 166
Un-cached Data Area 166
Return Address Stack Predictor (RSB) 166
Branch Predictor 167
Branch Target Buffer (BTB) 167
Decode Logic 167
Instruction Queue (ROB) 168
Queue Rate 169
Sequence Numbers 169
Input / Output Management 170
Device Configuration Blocks 170
Reset 170
Devices Built into the CPU / MPU 170
System Devices 170
External Interrupts 171
Overview 171
Interrupt Messages 171
Interrupt Controller 171

Interrupt Vector Table 172
Interrupt Group Filter 172
Interrupt Reflector 172
Interrupt Logger 173
Qupls3 Memory Management 174
Regions 174
Region Table Location 174
Page Allocation Map – PAM 174
Page Management Table - PMT 174
Overview 174
Location 175
PMTE Description 175
Access Control List 175
Share Count 175
Access Count 175
Key 175
Privilege Level 176
N 176
M 176
E 176
AL 176
C 176
urwx, srwx, hrwx, mrwx 176
Qupls3 Hierarchical Page Table Setup 176
Overview 176
Page Table Entry Format – PTE 177
Page Table 1MB Entry Format – PTE 177
QMMU – Qupls3 Memory Management Unit 178
Overview 178

Config Space 178
Registers 179
QIC – Qupls3 Interrupt Controller 183
Overview 183
System Usage 183
Priority Resolution 183
Config Space 183
Registers 184
Base Address Fields 185
CPU Affinity Group Table 186
Interrupt Enable Bits 186
Interrupt Pending Bits 186
Interrupt Vector Table 186
QIT – Qupls3 Interval Timer 187
Overview 187
System Usage 187
Config Space 187
Parameters 188
Registers 189
Programming 192
Interrupts 192
FTA Bus 193
Overview 193
Bus Tags 193
Single Cycle 193
Retry 194
Signal Description 194
Requests 194
Responses 194

Om 195
Cmd 195
BTE 195
CTI 196
Blen 196
Sz 196
Segment 197
TID 197
Cache 197
Message Signaled Interrupts 198
Glossary 199
ABI 199
AMO 199
Assembler 199
ATC 199
Base Pointer 199
Burst Access 200
BTB 200
Card Memory 200
Commit 200
Decimal Floating Point 201
Decode 201
Diadic 201
DUT 201
Endian 201
FIFO 201
FPGA 201
Floating Point 202
Frame Pointer 202

HDL 202
HLL 202
Instruction Bundle 202
Instruction Pointers 202
Instruction Prefix 203
Instruction Modifier 203
ISA 203
IPI 203
JIT 203
Keyed Memory 203
Linear Address 204
Machine Code 204
Milli-code 204
Monadic 204
MSI 204
Opcode 204
Operand 204
Physical Address 205
Physical Memory Attributes (PMA) 205
PIC 205
Posits 205
Program Counter 205
RAT 206
Retire 206
ROB 206
RSB 206
SIMD 206
Stack Pointer 207
Telescopic Memory 207

TLB 207
Trace Memory 207
Triadic 208
Vector Chaining 208
Vector Length (VL register) 208
Vector Mask (VM) 208
Virtual Address 208
Writeback 208
Miscellaneous 209
Reference Material 209
Trademarks 210
WISHBONE Compatibility Datasheet 211

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			
	Secure	Hyper	Super	App
80	CR0	CR0	CR0	CR0
81	CR1	CR1	CR1	CR1
82	CR2	CR2	CR2	CR2
83	CR3	CR3	CR3	CR3
84	CR4	CR4	CR4	CR4
85	CR5	CR5	CR5	CR5
86	CR6	CR6	CR6	CR6
87	CR7	CR7	CR7	CR7

88	LC
92	XH

	Carry Registers
93	CY0
94	CY1
95	CY2

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	Secure SP
89		MLR			
91		MPC		90	TCBA

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 32-bit registers are split into four groups, one for each operating mode. Condition registers at higher operating modes are not accessible at lower ones. 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		

TCBA

This register contains the address of the currently active task control block, in which the CPU’s context will be saved on a context switch. Only the upper 22-bits may be set; the lower 10 bits of the address are always zero.

31 10	9 0
Task Control Block Address$_{31…10}$	~

MC0 to MC3

This set of registers allows the processor to use instructions in the micro-code without affecting the visible state.

XH – Exception Handler Address

This register contains the address of the current exception handler.

Special Purpose Registers

[U/S/H/M]_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	dbg	Debug mode
18 to 20	mprv	memory privilege, 7 = use current
21 to 23	Swstk	Software stack
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.

IPL is the interrupt mask level, 0=all interrupts allowed, 63=nmi only

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

MPRV	Selected mode
0	User mode
1	Supervisor mode
2	Hypervisor mode

3	Machine mode
4	Debug
5	Reserved
6	Use stacked mode, om bits in previously stacked SR
7	Use current operating mode, om bits in SR

SWSTK: indicates which OS is running.

SSM: 1=step processor one instruction at a time. Trigger debugger after instruction executes.

[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.

63	4	3	2	1	0
~		mie	hie	sie	uie

[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.

63	16	15	8	7	0
~		Info		Cause	

U_FPCSR (CSR 0x0009) Floating Point Status and Control Register

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

Bit		Symbol	Description

31:29	**RM**	rm	rounding mode
28	**E5**	inexe	- inexact exception enable
27	**E4**	dbzxe	- divide by zero exception enable
26	**E3**	underxe	- underflow exception enable
25	**E2**	overxe	- overflow exception enable
24	**E1**	invopxe	- invalid operation exception enable
23	**NS**	ns	- non standard floating point indicator
Result Status			
22		fractie	- the last instruction (arithmetic or conversion) rounded intermediate result (or caused a disabled overflow exception)
21	**RA**	rawayz	rounded away from zero (fraction incremented)
20	**SC**	C	denormalized, negative zero, or quiet NaN
19	**SL**	neg <	the result is negative (and not zero)
18	**SG**	pos >	the result is positive (and not zero)
17	**SE**	zero =	the result is zero (negative or positive)
16	**SI**	inf ?	the result is infinite or quiet NaN
Exception Occurrence			
15	**X6**	swt	{reserved} - set this bit using software to trigger an invalid operation
14	**X5**	inerx	- inexact result exception occurred (sticky)
13	**X4**	dbzx	- divide by zero exception occurred
12	**X3**	underx	- underflow exception occurred
11	**X2**	overx	- overflow exception occurred
10	**X1**	giopx	- global invalid operation exception – set if any invalid operation exception has occurred
9	**GX**	gx	- global exception indicator – set if any enabled exception has happened
8	**SX**	sumx	- summary exception – set if any exception could occur if it was enabled - can only be cleared by software
Exception Type Resolution			
7	**X1T**	cvt	- attempt to convert NaN or too large to integer
6	**X1T**	sqrtx	- square root of non-zero negative
5	**X1T**	NaNCmp	- comparison of NaN not using unordered comparison instructions
4	**X1T**	infzero	- multiply infinity by zero
3	**X1T**	zerozero	- division of zero by zero
2	**X1T**	infdiv	- division of infinities
1	**X1T**	subinfx	- subtraction of infinities
0	**X1T**	snanx	- signaling NaN

[U/S/H/M]_XS (CSR 0x?00A) Extension State Register

This register contains state bits for processor extended instructions including floating-point. Two-bits are available for each extension. If the extension is not present, the bits will be zero. Otherwise, the bits are written with the value 3 if any changed occurs to the extension. The values 1 and 2 are usable by software.

												r	r	r	FP

The first four bit pairs are reserved for use, the first bit pair is for the floating-point state.

[U/S/H/M]_EVEC – (CSR 0x?038 to 0x?03C)

These registers contain the address of the environment call handler table for a given operating mode. The register is accessible only from the same mode or higher.

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

Reg #	
0x?038	EVEC[0] – user mode
0x?039	EVEC[1] - supervisor mode
0x?03A	EVEC[2] – hypervisor mode
0x?03B	EVEC[3] – machine mode
0x?03C	EVEC[4] - debug

[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.

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.

63 60	59 16	15 0
State$_4$	~$_{44}$	seed$_{16}$

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_TCBA (CSR 0x3005)

This register contains the address of the currently active task control 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 50-bits may be set; the lower 14 bits of the address are always zero.

63 14	13 0
Context Block Page$_{50}$	~

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 EPC 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_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_TOS (CSR 0x3080)

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

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.

63 13	12 0
Virtual Address$_{67..20}$	0_{13}

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

63	0
Virtual Address$_{75..12}$	

M_EPC_TOS (CSR 0x3108)

This register contains the address stack for the program counter used in exception handling. It is the top stack value.

Reg #	Name
0x3108	EPC

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 Operating Mode

For a hardware interrupt, the operating mode of the CPU is switched to the one specified in the interrupt table. For a software interrupt (environment / escalation call), the operating mode is switched to the next highest operating mode.

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.

Vector Table

Hardware interrupts use the vector in the interrupt table to determine where the interrupt service routine is located. Software exceptions excluding environment calls use the vector table located in the TVEC CSR for the next highest operating mode. Environment calls locate the processing routine using the EVEC CSR for the next highest operating mode.

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	Trap
11	reserved
12	reset
13	Alternate Cause > 15
14, 15	Reserved

	Applications Usage
16 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 an environment call instruction is executed while in User / App mode.

Supervisor Environment Call (17)

This fault is triggered when an environment call instruction is executed while in Supervisor mode.

Hypervisor Environment Call (18)

This fault is triggered when an environment call instruction is executed while in Hypervisor mode.

Machine Environment Call (19)

This fault is triggered when an environment call instruction is executed while in Machine mode.

TRAP (20)

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

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.

Task Support

Task Control Block / Context Block Layout

Context blocks have the following memory layout:

Word Number	Registers
0	Not used, reserved
0 to 30	General purpose registers 0 to 30
31	Safe stack pointer
32 to 63	Floating-point registers
64	User stack pointer
65	Supervisor stack pointer
66	Hypervisor stack pointer
67	Machine stack pointer
68 to 71	Micro-code temporaries #0 to #3
72 to 78	Branch registers
79	Program Counter
80 to 87	Condition Registers
88	Loop Counter
89	micro-code link register
90	Context block address register
91	Micro-code program counter
92 to 95	reserved
	CSR registers
96	Status register
97	Program Base and Limit register

The context block is aligned with a 16kB memory page. The processor's registers are stored beginning with the first word of the context block. The first 256 words of the context area are for CPU register file use. Space after the first 1k words of the context block may be used by the OS. Context blocks may be larger than 16kB depending on what information is stored in them. For instance, the Femtiki OS buffers the text mode video screen for the app in the context block.

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
1	?	Value is constant encoded directly in instruction, LX_2 is top two bits of the constant field in the instruction or additional opcode bits
0	0	Value comes from register Rs2
0	1	Value is 32-bit constant following the instruction
0	2	Value is 64-bit constant following the instruction
0	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	3029	28	17	16	15 11	10 9	8 6	5 0
CMP	1	Immediate$_{14}$			0	Rs1$_5$	0	CRd$_3$	3$_6$
CMP	0	0$_2$	~$_7$	Rs2$_5$	0	Rs1$_5$	0	CRd$_3$	3$_6$
CMP	0	1$_2$	~$_{12}$		0	Rs1$_5$	0	CRd$_3$	3$_6$
CMPA	1	Immediate$_{14}$			0	Rs1$_5$	1	CRd$_3$	3$_6$
CMPA	0	0$_2$	~$_7$	Rs2$_5$	0	Rs1$_5$	1	CRd$_3$	3$_6$
FCMPS	1	Immediate$_{14}$			0	FRs1$_5$	2	CRd$_3$	3$_6$
FCMPS	0	0$_2$	~$_7$	FRs2$_5$	0	FRs1$_5$	2	CRd$_3$	3$_6$
FCMPD	1	Immediate$_{14}$			1	FRs1$_5$	2	CRd$_3$	3$_6$
FCMPD	0	0$_2$	~$_7$	FRs2$_5$	1	FRs1$_5$	2	CRd$_3$	3$_6$

Branch Instruction Formats

	31	3029	28			16	15 11	10	6	5 0	
B[L]	1	Displacement$_{24…3}$							BRd$_3$	13$_5$	D
BLR[L]	0	0$_2$	BRs$_3$	Limit$_{13…3}$		~	Rs2$_5$	~$_2$	BRd$_3$	13$_5$	L
BLR[L]	0	1$_2$	BRs$_3$	Limit$_{13…3}$		~$_8$			BRd$_3$	13$_5$	L
[Dcc]Bcc[L]	1	D$_{12}$	BRs$_3$	Cnd$_3$	CRs$_6$	Disp$_{10…3}$			BRd$_3$	12$_5$	D
[Dcc]Bcc[L]	0	0$_2$	BRs$_3$	Cnd$_3$	CRs$_6$	~	Rs2$_5$	~$_2$	BRd$_3$	12$_5$	~
[Dcc]Bcc[L]	0	1$_2$	BRs$_3$	Cnd$_3$	CRs$_6$	~$_8$			BRd$_3$	12$_5$	~
[Dcc]Pcc	1	M$_{10}$	0$_3$	Cnd$_3$	CRs$_6$	Mask$_{8…1}$			7$_3$	12$_5$	M
ATOM	0	0$_2$	1$_3$	63$_6$	Mask$_{11…1}$				7$_3$	12$_5$	M
CARRY	0	0$_2$	2$_3$	CY$_2$	Mask$_{15…1}$				7$_3$	12$_5$	M
ACARRY	0	0$_2$	3$_3$	CY$_2$	Mask$_{15…1}$				7$_3$	12$_5$	M

Load and Store Instruction Formats

Load	1	Displacement$_{13…0}$				Cr	Rs1$_5$	Rd$_5$		Opcode$_5$
Indexed Ld	0	0$_2$	Disp$_5$	Sc$_2$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$		Opcode$_5$
Indexed Ld	0	LX$_2$	~$_5$	Sc$_2$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$		Opcode$_5$
Store	1	Displacement$_{13…0}$				U	Rs1$_5$	Rs2$_5$		Opcode$_5$
Indexed St	0	LX$_2$	Disp$_5$	Sc$_2$	Rs2$_5$	U	Rs1$_5$	Rs3$_5$		Opcode$_5$
Store Imm	1	Displacement$_{13…0}$				U	Rs1$_5$	sz$_2$	CL$_3$	Opcode$_5$
Indexed St	0	LX$_2$	Disp$_5$	Sc$_2$	Rs2$_5$	U	Rs1$_5$	sz$_2$	CL$_3$	Opcode$_5$

Atomic Memory Operations Instruction Formats

	31	3029	28 24	23 22	21 17	16	15 11	10 6	5 0
AMOADD	0	LX$_2$	0$_5$	Sz$_2$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	59$_5$
AMOAND	0	LX$_2$	1$_5$	Sz$_2$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	59$_5$
AMOOR	0	LX$_2$	2$_5$	Sz$_2$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	59$_5$
AMOSWAP	0	LX$_2$	6$_5$	Sz$_2$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	59$_5$
AMOCLEAR	0	LX$_2$	15$_5$	Sz$_2$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	59$_5$
CMPSWAP	0	LX$_2$	~$_2$	Rs3$_5$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	60$_5$

ALU Instruction Formats

	31	3029	28		17	16	15 11	10 6	5 0
ADD	1	Immediate$_{14}$				Cr	Rs1$_5$	Rd$_5$	4_6
ADD	0	LX$_2$	0_4	$\sim_3$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	4_6
ADD	0	1	0_4	$\sim_8$		Cr	Rs1$_5$	Rd$_5$	4_6
ADC	0	LX$_2$	1_4	$\sim_3$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	4_6
ABS	0	LX$_2$	2_4	$\sim_3$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	4_6
CNTLO	0	LX$_2$	3_4	$\sim_3$	$\sim_5$	Cr	Rs1$_5$	Rd$_5$	4_6
CNTLZ	0	LX$_2$	4_4	$\sim_3$	$\sim_5$	Cr	Rs1$_5$	Rd$_5$	4_6
CNTPOP	0	LX$_2$	5_4	$\sim_3$	$\sim_5$	Cr	Rs1$_5$	Rd$_5$	4_6
CNTTZ	0	LX$_2$	6_4	$\sim_3$	$\sim_5$	Cr	Rs1$_5$	Rd$_5$	4_6
ADB	1	Immediate$_{14}$				Cr	$\sim_2$ BR$_3$	Rd$_5$	5_6
ADB	0	LX$_2$	$\sim_7$		Rs2$_5$	Cr	$\sim_2$ BR$_3$	Rd$_5$	5_6
MULA	1	Immediate$_{14}$				Cr	Rs1$_5$	Rd$_5$	6_6
MULA	0	0_2	0_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	6_6
MUL	0	0_2	1_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	6_6
MULSA	0	0_2	2_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	6_6
MULH	0	0_2	4_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	6_6
AND	1	Immediate$_{14}$				Cr	Rs1$_5$	Rd$_5$	8_6
AND	0	LX$_2$	0_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	8_6
NAND	0	LX$_2$	1_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	8_6
ANDC	0	LX$_2$	2_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	8_6
QEXT	0	LX$_2$	4_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	8_6
QFEXT	0	LX$_2$	5_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	8_6
REGS	0	LX$_2$	6_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	8_6
FREGS	0	LX$_2$	7_3	$\sim_4$	FRs2$_5$	Cr	FRs1$_5$	FRd$_5$	8_6
OR	1	Immediate$_{14}$				Cr	Rs1$_5$	Rd$_5$	9_6
OR	0	LX$_2$	0_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	9_6
NOR	0	LX$_2$	1_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	9_6
ORC	0	LX$_2$	2_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	9_6
XOR	1	Immediate$_{14}$				Cr	Rs1$_5$	Rd$_5$	10_6
XOR	0	LX$_2$	0_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	10_6
SUBF	1	Immediate$_{14}$				Cr	Rs1$_5$	Rd$_5$	12_6
SUBF	0	LX$_2$	0_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	12_6
SBC	0	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	1	Immediate$_{14}$				Cr	Rs1$_5$	Rd$_5$	14_6
DIVA	0	LX$_2$	0_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	14_6
DIV	0	LX$_2$	1_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	14_6
DIVSU	0	LX$_2$	2_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	14_6
SQRT	0	LX$_2$	3_3	$\sim_4$	$\sim_5$	Cr	Rs1$_5$	Rd$_5$	14_6
MODA	0	LX$_2$	4_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	14_6
MOD	0	LX$_2$	5_3	$\sim_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	14_6
LOADA	1	Displacement$_{13\ldots0}$				Cr	Rs1$_5$	Rd$_5$	39_5
LOADA	0	LX$_2$	Disp$_5$	Sc$_2$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	39_5

FPU Instruction Formats

	31	3029	28		17	16	15 11	10 6	5 0
FADD	0	LX_2	4_4	Rm_3	$Rs2_5$	Cr	$FRs1_5$	FRd_5	54_6
FADD	0	1	4_4	Rm_3	0_5	Cr	$FRs1_5$	FRd_5	54_6
FABS	0	1	4_4	Rm_3	1_5	Cr	$FRs1_5$	FRd_5	54_6
FSUB	0	LX_2	5_4	Rm_3	$Rs2_5$	Cr	$FRs1_5$	FRd_5	54_6
FSUB	0	1	5_4	Rm_3	$\sim_5$	Cr	$FRs1_5$	FRd_5	54_6
FMUL	0	LX_2	6_4	Rm_3	$Rs2_5$	Cr	$FRs1_5$	FRd_5	54_6
FMUL	0	1	6_4	Rm_3	$\sim_5$	Cr	$FRs1_5$	FRd_5	54_6
FDIV	0	LX_2	7_4	Rm_3	$Rs2_5$	Cr	$FRs1_5$	FRd_5	54_6
FDIV	0	1	7_4	Rm_3	$\sim_5$	Cr	$FRs1_5$	FRd_5	54_6
FSGNJ	0	LX_2	8_4	0_3	$Rs2_5$	Cr	$FRs1_5$	FRd_5	54_6
FSGNJ	0	1	8_4	0_3	$\sim_5$	Cr	$FRs1_5$	FRd_5	54_6
FSGNJN	0	LX_2	8_4	1_3	$Rs2_5$	Cr	$FRs1_5$	FRd_5	54_6
FSGNJN	0	1	8_4	1_3	$\sim_5$	Cr	$FRs1_5$	FRd_5	54_6
FSGNJX	0	LX_2	8_4	2_3	$Rs2_5$	Cr	$FRs1_5$	FRd_5	54_6
FSGNJX	0	1	8_4	2_3	$\sim_5$	Cr	$FRs1_5$	FRd_5	54_6
FSCALEB	0	LX_2	8_4	3_3	$Rs2_5$	Cr	$FRs1_5$	FRd_5	54_6
FSCALEB	0	1	8_4	3_3	$\sim_5$	Cr	$FRs1_5$	FRd_5	54_6
FCVTF2I	0	1	10_4	$\sim_3$	0_5	Cr	$FRs1_5$	Rd_5	54_6
FCVTI2F	0	1	10_4	Rm_3	1_5	Cr	$Rs1_5$	FRd_5	54_6
FSIGN	0	1	10_4	$\sim_3$	16_5	Cr	$FRs1_5$	FRd_5	54_6
FSQRT	0	1	10_4	$\sim_3$	17_5	Cr	$FRs1_5$	FRd_5	54_6
FCOS	0	1	11_4	Rm_3	0_5	Cr	$FRs1_5$	FRd_5	54_6
FSIN	0	1	11_4	Rm_3	1_5	Cr	$FRs1_5$	FRd_5	54_6

Shift Instruction Formats

	31	3029	28			17	16	15 11	10 6	5 0
SLL	1	0_2	0_3	H	~	$Shamt_6$	Cr	$Rs1_5$	Rd_5	2_6
SLL	0	0_2	0_3	H	$\sim_3$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	2_6
SRL	1	0_2	1_3	H	~	$Shamt_6$	Cr	$Rs1_5$	Rd_5	2_6
SRL	0	0_2	1_3	H	$\sim_3$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	2_6
SRA	1	0_2	2_3	Rm_2		$Shamt_6$	Cr	$Rs1_5$	Rd_5	2_6
SRA	0	0_2	2_3	Rm_2		$Rs2_5$	Cr	$Rs1_5$	Rd_5	2_6
RO[L|R]	1	0_2	4_3	L	~	$Shamt_6$	Cr	$Rs1_5$	Rd_5	2_6
RO[L|R]	0	0_2	4_3	L	$\sim_3$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	2_6
EXTZ	1	2_2	Me_6			Mb_6	Cr	$Rs1_5$	Rd_5	2_6
EXT	1	3_2	Me_6			Mb_6	Cr	$Rs1_5$	Rd_5	2_6
EXT[Z]	0	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	1	Op_2	$CSRno_{12}$			Cr	$Rs1_5$		Rd_5	7_6
	0	0_2	Op_2	$\sim_5$	$Rs2_5$	Cr	$Rs1_5$		Rd_5	7_6
32-bit data	0	1_2	$CSRno_{12}$			Cr		Op_2	Rd_5	7_6

BRK / RFI Instruction Formats

	31	3029	28	17	16	15 11	10 6	5 0
BRK	1	0_2	0_{12}		0	0_5	0_5	0_6
ERET	1	0_2	2_{12}		0	0_5	Const$_5$	0_6
ERET2	1	0_2	3_{12}		0	0_5	Const$_5$	0_6
	0	LX$_2$		Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	Opcode$_6$

TRAP 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
ECALL	1	0_2	Vector$_{12}$		~	Rs1$_5$	31_5	28_6
ECALL	0	0_2	$\sim_7$	Rs2$_5$	~	Rs1$_5$	31_5	28_6
CHK	0	Op$_4$	Rs3$_5$	Rs2$_5$	~	Rs1$_5$	Offs$_5$	29_6

Macro Instruction Formats

	31	3029	28	17	16	1514	13 11	10 6	5 0
PUSH	1	2_2	Gr$_2$	Rs$_{17\ldots8}$	0	Gr$_2$	Rs$_{7\ldots0}$		30_6
ENTER	1	3_2	$\sim_{12}$		0	Rs2$_5$		Rs1$_5$	30_6
POP	1	2_2	Gr$_2$	Rd$_{17\ldots8}$	0	Gr$_2$	Rd$_{7\ldots0}$		31_6
EXIT	1	3_2	R	Immediate$_{11}$	0	Rs2$_5$		Rs1$_5$	31_6

MOVE Instruction Format

	31	3029	28			17	16	15 11	10 6	5 0
MOVE	1	LX_2	0_3	0_5	$Rs1_{65}$	Rd_{65}	Cr	$Rs1_{4...0}$	$Rd_{4...0}$	15_6
VIRT2PHYS	1	LX_2	0_3	2_5	$Rs1_{65}$	Rd_{65}	Cr	$Rs1_{4...0}$	$Rd_{4...0}$	15_6
MOVEA	1	LX_2	1_3	$\sim_5$	$Rs1_{65}$	Rd_{65}	Cr	$Rs1_{4...0}$	$Rd_{4...0}$	15_6
CMOVZ	1	I_{98}	4_3	$Immediate_{8...0}$			Cr	$Rs1_{4...0}$	$Rd_{4...0}$	15_6
CMOVZ	0	LX_2	4_3	$\sim_4$	$Rs2_5$		Cr	$Rs1_{4...0}$	$Rd_{4...0}$	15_6
CMOVNZ	1	I_{98}	5_3	$Immediate_{8...0}$			Cr	$Rs1_{4...0}$	$Rd_{4...0}$	15_6
CMOVNZ	0	LX_2	5_3	$\sim_4$	$Rs2_5$		Cr	$Rs1_{4...0}$	$Rd_{4...0}$	15_6
MOVSX	1	Ui_{65}	2_3	Ui_{40}	$Rs1_{65}$	Rd_{65}	Cr	$Rs1_{4...0}$	$Rd_{4...0}$	15_6
MOVZX	1	Ui_{65}	3_3	Ui_{40}	$Rs1_{65}$	Rd_{65}	Cr	$Rs1_{4...0}$	$Rd_{4...0}$	15_6
BMAP	1	Sz_2	6_3	Op_4	$Rs2_5$		Cr	$Rs1_{4...0}$	$Rd_{4...0}$	15_6

Instruction Pres/Postfixes and Modifiers Instruction Formats

	31	3029	28 17	16	15 11	10 6	5 0
PFX	1	$Immediate_{27...5}$				Wh_2	61_6

Condition Register Manipulation Instruction Formats

	31	3029	28		18	17 12	11 6	5 0
CRAND	1	0_2	0_4	$\sim_6$	I	$CRs1_6$	CRd_6	11_6
CROR	1	0_2	1_4	$\sim_6$	I	$CRs1_6$	CRd_6	11_6
CRXOR	1	0_2	2_4	$\sim_6$	I	$CRs1_6$	CRd_6	11_6
CRANDC	1	0_2	3_4	$\sim_6$	I	$CRs1_6$	CRd_6	11_6
CRAND	0	0_2	0_4	~	$CRs2_6$	$CRs1_6$	CRd_6	11_6
CROR	0	0_2	1_4	~	$CRs2_6$	$CRs1_6$	CRd_6	11_6
CRXOR	0	0_2	2_4	~	$CRs2_6$	$CRs1_6$	CRd_6	11_6
CRANDC	0	0_2	3_4	~	$CRs2_6$	$CRs1_6$	CRd_6	11_6
CRNAND	0	0_2	4_4	~	$CRs2_6$	$CRs1_6$	CRd_6	11_6
CRNOR	0	0_2	5_4	~	$CRs2_6$	$CRs1_6$	CRd_6	11_6
CRXNOR	0	0_2	6_4	~	$CRs2_6$	$CRs1_6$	CRd_6	11_6
CRORC	0	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 ADB	6 MUL	7 CSR
001x	8 AND QEXT	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] Pcc ATOM	25 Bc[L] DBc[L] Pcc ATOM	26 B[L] BLR[L]	27 B[L] BLR[L]	28 TRAP SYS	29 CHK	30 PUSH PUSHF ENTER	31 POP POPF LEAVE
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 CMPSWAP	61 PFX	62	63 NOP

Instruction Descriptions

Comparison Instructions

CMP – Compare

Description:

Compare two source registers Rs1 and Rs2 or Rs1 and a constant and place the result in the destination condition register CRd. The result is an eight-bit vector representing the relationship between the compared values. All register values are signed integers. The carry bit of the result is set as if the two source operands were subtracted. The summary overflow and reserved bits are unaffected.

Instruction Format:

	31	3029	28	17	16	15 11	10 9	8 6	5 0
CMP	1	Immediate_{14}			0	$Rs1_5$	0	CRd_3	3_6
CMP	0	0_2	$\sim_7$	$Rs2_5$	0	$Rs1_5$	0	CRd_3	3_6
CMP	0	LX_2	$\sim_{12}$		0	$Rs1_5$	0	CRd_3	3_6

Operation:

CRd = Rs1 ? Rs2

OR

CRd = Rs1 ? Constant

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

CMPA – Compare Addresses (unsigned)

Description:

Compare two source registers Rs1 and Rs2 or Rs1 and a constant and place the result in the destination condition register CRd. The result is an eight-bit vector representing the relationship between the compared values. All register values are unsigned integers. The carry bit of the result is set as if the two source operands were subtracted. The summary overflow and reserved bits are unaffected.

Instruction Format:

	31	30 29	28	17	16	15 11	10 9	8 6	5 0
CMPA	1	Immediate$_{14}$			0	Rs1$_5$	1	CRd$_3$	3$_6$
CMPA	0	0$_2$	~$_7$	Rs2$_5$	0	Rs1$_5$	1	CRd$_3$	3$_6$
CMPA	0	LX$_2$	~$_{12}$		0	Rs1$_5$	1	CRd$_3$	3$_6$

Operation:

CRd = Rs1 ? Rs2

OR

CRd = Rs1 ? Constant

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

FCMP – Compare Floats

Description:

Compare two source registers Rs1 and Rs2 or Rs1 and a constant and place the result in the destination condition register CRd. The result is an eight-bit vector representing the relationship between the compared values. All register values are floating-point values. The carry/infinite bit of the result is set if the two source operands are both infinite, otherwise it is cleared. The summary overflow / unordered bit is set if the comparison is unordered, otherwise it is not affected. The reserve bit is unaffected. FCMPS performs a single precision (32-bit) comparison. FCMPD performs a double precision (64-bit) comparison.

Instruction Format:

	31	3029	28	17	16	15 11	10 9	8 6	5 0
FCMPS	1	Immediate$_{14}$			0	FRs1$_5$	2	CRd$_3$	3_6
FCMPS	0	0_2	$\sim_7$	FRs2$_5$	0	FRs1$_5$	2	CRd$_3$	3_6
FCMPS	0	LX$_2$	$\sim_{12}$		0	FRs1$_5$	2	CRd$_3$	3_6
FCMPD	1	Immediate$_{14}$			1	FRs1$_5$	2	CRd$_3$	3_6
FCMPD	0	0_2	$\sim_7$	FRs2$_5$	1	FRs1$_5$	2	CRd$_3$	3_6
FCMPD	0	LX$_2$	$\sim_{12}$		1	FRs1$_5$	2	CRd$_3$	3_6

Operation:

CRd = FRs1 ? FRs2

OR

CRd = FRs1 ? Constant

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

Arithmetic Instructions

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	0	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	1	Immediate$_{14}$				Cr	Rs1$_5$	Rd$_5$	4$_6$
ADD	0	0$_2$	0$_4$	~$_3$	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	1	Immediate$_{14}$			Cr	~$_2$	BR$_3$	Rd$_5$	5$_6$
ADB	0	LX$_2$		Rs2$_5$	Cr	~$_2$	BR$_3$	Rd$_5$	5$_6$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rd = BR + immediate

Exceptions:

Notes:

~~ADC[.] – Add with Carry (deprecated)~~

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	0	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	0	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	0	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	0	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	0	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

Operation	Op	Mnemonic
Or to CSR (set bits)	0	CSRRS
And complement to CSR (clear bits)	1	CSRRC

Supported Operand Sizes: N/A

Instruction Formats:

	31	3029	28		17	16	1514	15 11	10 6	5 0
CSRxx	1	Op$_2$	CSRno$_{12}$			Cr	Rs1$_5$		Rd$_5$	7$_6$
	0	0$_2$	Op$_2$	~$_5$	Rs2$_5$	Cr	Rs1$_5$		Rd$_5$	7$_6$
32-bit data	0	1$_2$	CSRno$_{12}$			Cr	~	Op$_2$	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	1	$\text{Displacement}_{13\ldots0}$				Cr	$Rs1_5$	Rd_5	39_5
LOADA	0	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 (deprecated)~~

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	0	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	1	Immediate$_{14}$				Cr	Rs1$_5$	Rd$_5$	12$_6$
SUBF	0	0$_2$	0$_4$	~$_3$	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	0	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	1	Immediate$_{14}$				Cr	Rs1$_5$	Rd$_5$	14$_6$
DIVA	0	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	0	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	1	Immediate$_{14}$				Cr	Rs1$_5$	Rd$_5$	6$_6$
MULA	0	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

EXT[.] – Extract Bit Field

Description:

A bit field is extracted from the source operand, sign extended, and the result placed in the target register. Mb specifies the first bit of the bitfield, Me specifies the last bit of the bitfield. The mask begin and end may also be supplied in register Rs2 as the first and second bytes of the register.

Instruction Format: BITFLD

EXT Rd, Rs1, Mb, Me

Instruction Format:

	31	30 29	28			17	16	15 11	10 6	5 0
EXT	1	3_2	Me_6		Mb_6		Cr	$Rs1_5$	Rd_5	2_6
EXT	0	0_2	3_3	1	~	$Rs2_5$	Cr	$Rs1_5$	Rd_5	2_6

Operation:

Rt = sign extend(Ra[ME:MB])

Clock Cycles:

Execution Units: All Integer ALUs

Exceptions: none

Notes:

EXTZ[.] – Extract Bit Field and Zero extend

Description:

A bit field is extracted from the source operand, zero extended, and the result placed in the target register. Mb specifies the first bit of the bitfield, Me specifies the last bit of the bitfield. The mask begin and end may also be supplied in register Rs2 as the first and second bytes of the register.

Instruction Format: BITFLD

EXTU Rt, Ra, Rb, Rc

Instruction Format:

	31	3029	28			17	16	15 11	10 6	5 0
EXTZ	1	2_2	Me_6			Mb_6	Cr	$Rs1_5$	Rd_5	2_6
EXTZ	0	0_2	3_3	0	~	$Rs2_5$	Cr	$Rs1_5$	Rd_5	2_6

Operation:

Rt = zero extend(Ra[ME:MB])

Clock Cycles:

Execution Units: All Integer ALUs

Exceptions: none

Notes:

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	1	0_2	0_3	H	~	$Shamt_6$	Cr	$Rs1_5$	Rd_5	2_6
SLL	0	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	1	0_2	2_3	Rm_2		$Shamt_6$	Cr	$Rs1_5$	Rd_5	2_6
SRA	0	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	1	0_2	1_3	H	~	Shamt_6	Cr	Rs1_5	Rd_5	2_6
SRL	0	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	30 29	28		17	16	15 11	10 6	5 0
AND	1	Immediate$_{14}$				Cr	Rs1$_5$	Rd$_5$	8$_6$
AND	0	LX$_2$	0$_3$	~$_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	8$_6$

Operation:

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

ANDC[.] – Bitwise And with Complement

Description:

Bitwise ‘And’ two source registers Rs1 and the complement of Rs2 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
ANDC	0	LX_2	2_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:

NAND[.] – Bitwise And then invert

Description:

Bitwise ‘And’ two source registers Rs1 and Rs2 and place the inverted 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
NAND	0	LX_2	1_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:

NOR[.] – Bitwise Or then invert

Description:

Bitwise 'or' two source registers Rs1 and Rs2 and place the inverted result in the destination register Rd. All registers are integer registers.

Instruction Format:

	31	3029	28		17	16	15 11	10 6	5 0
NOR	0	LX_2	1_3	$\sim_4$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	9_6

Operation:

Rd = ~(Rs1 | Rs2)

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	1	Immediate_{14}				Cr	Rs1_5	Rd_5	9_6
OR	0	LX_2	0_3	$\sim_4$	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:

ORC[.] – Bitwise Or with Complement

Description:

Bitwise 'or' two source registers Rs1 and the complement of Rs2 and place the inverted result in the destination register Rd. All registers are integer registers.

Instruction Format:

	31	3029	28		17	16	15 11	10 6	5 0
ORC	0	LX$_2$	2_3	~$_4$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	9_6

Operation:

Rd = ~(Rs1 | Rs2)

Clock Cycles: 1

Execution Units: All Integer ALUs, all FPUs

Exceptions: none

Notes:

XNOR[.] – Bitwise Exclusive Or then invert

Description:

Bitwise exclusively ‘or’ two source registers Rs1 and Rs2 and place the inverted result in the destination register. All registers are integer registers.

Instruction Format: R3

	31	3029	28		17	16	15 11	10 6	5 0
XNOR	0	LX_2	1_3	$\sim_4$	$Rs2_5$	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:

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	1	Immediate$_{14}$				Cr	Rs1$_5$	Rd$_5$	10$_6$
XOR	0	LX$_2$	0$_3$	~$_4$	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:

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 O5 * 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

BMAP[.] – Byte Map

Description:

First the target register is cleared, then bytes are mapped from the byte source Rs1 into bytes in the destination register Rd. This instruction may be used to permute the bytes in register Rs1 and store the result in Rd. This instruction may also pack bytes, wydes or tetras. The map is determined by the low order 32-bits of register Rs2 Bytes which are not mapped will end up as zero in the target register. Each nybble of the 32-bit value indicates the target byte in the target register.

Instruction Format: R3

BMAP Rt, Ra, Rb

	31	3029	28		17	16	15 11	10 6	5 0
BMAP	1	Sz$_2$	6_3	Op$_4$	Rs2$_5$	Cr	Rs1$_{4…0}$	Rd$_{4…0}$	15_6

Sz2	Mnemonic	Unit Operated On
0	BMAPB	Bytes
1	BMAPW	wydes
2	BMAPT	tetras

Op3		
0	Byte map	Uses Rs2
1	Reverse byte order	
2	Broadcast byte	
3	Mix	
4	Shuffle	
5	Alt	
6 to 7	reserved	

Operation:

Vector Operation

Execution Units: First Integer ALU

Exceptions: none

Notes:

CMOVZ[.]– Conditional Move if Zero to Register

Description:

Conditionally move either a register value in Rs2 or an immediate constant into register Rd, if the value in register Rs1 is zero. Rd will be unaffected if Rs1 is non-zero.

Instruction Formats: MOV

	31	3029	28		17	16	15 11	10 6	5 0
CMOVZ	1	I_{109}	4_3	$\text{Immediate}_{8\ldots0}$		Cr	$\text{Rs1}_{4\ldots0}$	$\text{Rd}_{4\ldots0}$	15_6
CMOVZ	0	LX_2	4_3	$\sim_4$	Rs2_5	Cr	$\text{Rs1}_{4\ldots0}$	$\text{Rd}_{4\ldots0}$	15_6

Operation: R2

If Rs1= 0 then Rd = Rs2

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

CMOVNZ[.]– Conditional Move in Non-zero to Register

Description:

Conditionally move either a register value in Rs2 or an immediate constant into register Rd, if the value in register Rs1 is non-zero. Rd will be unaffected if Rs1 is zero.

Instruction Formats: MOV

	31	3029	28		17	16	15 11	10 6	5 0
CMOVNZ	1	I_{109}	5_3	$Immediate_{8...0}$		Cr	$Rs1_{4...0}$	$Rd_{4...0}$	15_6
CMOVNZ	0	LX_2	5_3	$\sim_4$	$Rs2_5$	Cr	$Rs1_{4...0}$	$Rd_{4...0}$	15_6

Operation: R2

If RS1<>0 then Rd = Rs2

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

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	1	LX_2	0_3	0_5	$Rs1_{65}$	Rd_{65}	Cr	$Rs1_{4\ldots0}$	$Rd_{4\ldots0}$	15_6
MOVEA	1	LX_2	1_3	$\sim_5$	$Rs1_{65}$	Rd_{65}	Cr	$Rs1_{4\ldots0}$	$Rd_{4\ldots0}$	15_6
MOVSX	1	Ui_{65}	2_3	Ui_{40}	$Rs1_{65}$	Rd_{65}	Cr	$Rs1_{4\ldots0}$	$Rd_{4\ldots0}$	15_6
MOVZX	1	Ui_{65}	3_3	Ui_{40}	$Rs1_{65}$	Rd_{65}	Cr	$Rs1_{4\ldots0}$	$Rd_{4\ldots0}$	15_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

Floating-Point Instructions

FABS[.] – Absolute Value

Description:

This instruction is an alternate mnemonic for the FSGNJ instruction. It computes the absolute value of the contents of the source operand and places the result in FRd. The sign bit of the value is cleared. No rounding occurs.

Integer Instruction Format:

FABS FRd, FRs1

	31	3029	28		17	16	15 11	10 6	5 0
FABS	0	LX_2	8_4	0_3	$FRs2_5$	Cr	0_5	FRd_5	54_6
FABS	0	1	8_4	0_3	$\sim_5$	Cr	0_5	FRd_5	54_6

Operation:

FRd = Abs(FRs1)

Execution Units: All FPUs, All ALUs

Clock Cycles: 1

Exceptions: none

Notes:

FADD[.] –Float Addition

Description:

Add two source operands and place the sum in the destination register. Values are treated as floating-point values.

Supported Operand Sizes:

Instruction Format: FLT

FADD FRd, FRs1, FRs2

	31	3029	28		17	16	15 11	10 6	5 0
FADD	0	LX_2	4_4	Rm_3	$Rs2_5$	Cr	$Rs1_5$	Rd_5	54_6
FADD	0	1	4_4	Rm_3	0_5	Cr	$Rs1_5$	Rd_5	54_6

Operation:

Rt = Ra + Rb

Clock Cycles: 8

Execution Units: All FPU’s

Exceptions: none

Notes:

FCOS[.] – Float Cosine

Description:

This instruction computes an approximation of the co-sine value of the contents of the source operand and places the result in Fd.

Integer Instruction Format: R1

FCOS Fd, Fs1

	31	3029	28		17	16	15 11	10 6	5 0
FCOS	0	1	11_4	Rm_3	0_5	Cr	$Rs1_5$	Rd_5	54_6

Operation:

Fd = fcos(Fs1)

Clock Cycles: 42

Execution Units: FPU #0

Exceptions: none

Notes:

FCVTF2I[.] – Convert Float to Integer

Description:

This instruction converts the contents of the source operand to the equivalent integer and places the result in Rd. The result is rounded up.

Integer Instruction Format: R1

FCVTF2I Rd, Fs1

	31	3029	28		17	16	15 11	10 6	5 0
FCVTF2I	0	1	10_4	$\sim_3$	0_5	Cr	$FRs1_5$	Rd_5	54_6

Clock Cycles: 2

Operation:

Rd = CvtToInt(Fs1)

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FCVTFI2F[.] – Convert Integer to Float

Description:

This instruction converts the contents of the source operand to the equivalent float and places the result in Fd.

Integer Instruction Format: R1

FCVTF2I Fd, Rs1

	31	3029	28		17	16	15 11	10 6	5 0
FCVTI2F	0	1	10_4	Rm_3	1_5	Cr	$Rs1_5$	FRd_5	54_6

Clock Cycles: 2

Operation:

Rd = CvtIntToFloat(Fs1)

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FDIV[.] – Float Division

Description:

Divide source dividend operand in FRs1 by divisor operand in FRs2 and place the quotient in the destination register FRd. All registers are floating-point registers. Condition register CR1 may be updated if the Cr bit of the instruction is set.

Instruction Format:

	31	3029	28		17	16	15 11	10 6	5 0
FDIV	0	LX_2	7_4	Rm_3	$Rs2_5$	Cr	$Rs1_5$	Rd_5	54_6
FDIV	0	1	7_4	Rm_3	$\sim_5$	Cr	$Rs1_5$	Rd_5	54_6

Operation:

FRd = FRs1 / Rs2

Clock Cycles: 46

Execution Units: FPU #0 Only

Exceptions:

Notes:

FMA –Float Multiply and Add

Description:

Multiply two source operands, add a third operand and place the result in the destination register. All register values are treated as floating-point values.

FMA uses the FREGS prefix to specify the third source operand.

FREGS R0,Rs3,R0
FMUL Rd,Rs1,Rs2

Instruction Format: FLT3

FMA Rd, Rs1, Rs2, Rs3

	31	3029	28		17	16	15 11	10 6	5 0
FREGS	0	LX$_2$	7_3	$\sim_4$	FRs4$_5$	Cr	FRs3$_5$	FRd2$_5$	8_6

	31	3029	28		17	16	15 11	10 6	5 0
FMUL	0	LX$_2$	6_4	Rm$_3$	Rs2$_5$	Cr	Rs1$_5$	Rd$_5$	54_6
FMUL	0	1	6_4	Rm$_3$	$\sim_5$	Cr	Rs1$_5$	Rd$_5$	54_6

Operation:

FRd = FRs1 * FRs2 + FRs3

Clock Cycles: 8

Execution Units: All FPUs

Exceptions: none

Notes:

FMUL[.] –Float Multiplication

Description:

Multiply two source operands and place the product in the destination register. All registers values are treated as double precision floating-point values. An immediate value is converted to double precision value from single, or double precision. Condition register CR1 may be updated if the Cr bit of the instruction is set.

Instruction Format: FLT

FMUL Rt, Ra, Rb

	31	3029	28		17	16	15 11	10 6	5 0
FMUL	0	LX_2	6_4	Rm_3	$Rs2_5$	Cr	$Rs1_5$	Rd_5	54_6
FMUL	0	1	6_4	Rm_3	$\sim_5$	Cr	$Rs1_5$	Rd_5	54_6

Operation:

Rd = Rs1 * Rs2 + R0

Clock Cycles: 8

Execution Units: All FPUs

Exceptions: none

Notes:

FNEG[.] – Negative Value

Description:

This instruction is an alternate mnemonic for the FSGNJN instruction where both source operands are the same. This instruction computes the negative value of the contents of the source operand FRs1 and places the result in FRd. The sign bit of the value is inverted. No rounding occurs.

Integer Instruction Format:

FNEG FRd, FRs1

	31	3029	28		17	16	15 11	10 6	5 0
FNEG	0	LX_2	8_4	1_3	$Rs2_5$	Cr	$Rs1_5$	Rd_5	54_6
FNEG	0	1	8_4	1_3	$\sim_5$	Cr	$Rs1_5$	Rd_5	54_6

Operation:

FRd = Neg(FRs1)

Execution Units: All FPUs, All ALUs

Clock Cycles: 1

Exceptions: none

Notes:

FSCALEB[.] –Scale Exponent

Description:

Add the source operand to the exponent. The second source operand is an integer value. No rounding occurs.

Instruction Formats:

FSCALEB Fd, Fs1, Rb

	31	3029	28		17	16	15 11	10 6	5 0
FSCALEB	0	LX_2	8_4	3_3	$Rs2_5$	Cr	$FRs1_5$	FRd_5	54_6
FSCALEB	0	1	8_4	3_3	$\sim_5$	Cr	$FRs1_5$	FRd_5	54_6

Operation:

Clock Cycles: 1

Execution Units: All FPUs

Exceptions: none

Notes:

FSGNJ[.] – Float Sign Inject

Description:

Copy the sign of FRs1 and the exponent and significand of FRs2 into the destination register FRd. No rounding occurs.

Instruction Format:

FSGNJ FRd, FRs1, FRs2

	31	3029	28		17	16	15 11	10 6	5 0
FSGNJ	0	LX_2	8_4	0_3	$Rs2_5$	Cr	$Rs1_5$	Rd_5	54_6
FSGNJ	0	1	8_4	0_3	$\sim_5$	Cr	$Rs1_5$	Rd_5	54_6

Operation:

FRd = {FRs1.sign, FRs2.exp, FRs2.sig}

Clock Cycles: 1

Execution Units: All FPUs, All ALUs

Exceptions: none

Notes:

FSGNJN[.] – Float Sign Inject and Negate

Description:

Copy the sign of FRs1 and the exponent and significand of FRs2 into the destination register FRd. Negate Register FRd. No rounding occurs.

Instruction Format:

FSGNJN FRd, FRs1, FRs2

	31	3029	28		17	16	15 11	10 6	5 0
FSGNJN	0	LX_2	8_4	1_3	$Rs2_5$	Cr	$Rs1_5$	Rd_5	54_6
FSGNJN	0	1	8_4	1_3	$\sim_5$	Cr	$Rs1_5$	Rd_5	54_6

Operation:

FRd = -{FRs1.sign, FRs2.exp, FRs2.sig}

Clock Cycles: 1

Execution Units: All FPUs, All ALUs

Exceptions: none

Notes:

FSIGN[.] – Sign of Number

Description:

This instruction provides the sign of a floating point number contained in a floating-point register as a floating-point result. The result is +1.0 if the number is positive, 0.0 if the number is zero, and -1.0 if the number is negative.

Instruction Format:

	31	3029	28		17	16	15 11	10 6	5 0
FSIGN	0	1	10_4	$\sim_3$	16_5	Cr	$FRs1_5$	FRd_5	54_6

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Fd = sign of (Fs1)

FSQRT[.] – Floating point square root

Description:

Take the square root of the floating-point number in register Fs1 and place the result into destination register Fd. The sign bit (bit 63) of the register is set to zero. This instruction can generate NaNs.

Instruction Format:

	31	3029	28		17	16	15 11	10 6	5 0
FSQRT	0	1	10_4	$\sim_3$	17_5	Cr	$FRs1_5$	FRd_5	54_6

Operation:

Fd = fsqrt (Fs1)

Clock Cycles: 72

Execution Units: Floating Point

FSUB[.] –Float Subtraction

Description:

Subtract two source operands and place the difference in the destination register.

Supported Operand Sizes:

Instruction Format: FLT

FSUB Fd, Fs1, Fs2

	31	3029	28		17	16	15 11	10 6	5 0
FSUB	0	LX_2	5_4	Rm_3	$Rs2_5$	Cr	$Rs1_5$	Rd_5	54_6
FSUB	0	1	5_4	Rm_3	$\sim_5$	Cr	$Rs1_5$	Rd_5	54_6

Operation:

Fd = Fs1 – Fs2

Clock Cycles: 8

Execution Units: All FPUs

Exceptions: none

Notes:

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	$Displacement_{11\ldots0}$	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	$Opcode_5$

AMOADD[.] – AMO Addition

Description:

Atomically add source operand register Rs2 to value from memory and store the result back to memory. The original value of the memory cell is stored in register Rd. The memory address is the sum of Rs1 and a displacement constant.

Supported Operand Sizes: .o

Instruction Formats: AMO

AMOADD Rd, Rs2, d[Rs1]

	31	3029	28 24	23 22	21 17	16	15 11	10 6	5 0
AMOADD	0	LX_2	0_5	Sz_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	59_5

Clock Cycles:

AMOAND[.] – AMO Bitwise ‘And’

Description:

Atomically bitwise ‘and’ source operand register Rs2 to value from memory and store the result back to memory. The original value of the memory cell is stored in register Rd. The memory address the sum of Rs1 and a displacement.

Supported Operand Sizes: .o

Instruction Formats: AMO

AMOAND Rd, Rs2, d[Rs1]

	31	3029	28 24	23 22	21 17	16	15 11	10 6	5 0
AMOAND	0	LX_2	1_5	Sz_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	59_5

Clock Cycles:

AMOOR[.] – AMO Bitwise ‘Or’

Description:

Atomically bitwise ‘or’ source operand register Rs2 to value from memory and store the result back to memory. The original value of the memory cell is stored in register Rd. The memory address is the sum of Rs1 and a displacement.

Supported Operand Sizes: .o

Instruction Formats: AMO

AMOAND Rd, Rs2, d[Rs1]

	31	3029	28 24	23 22	21 17	16	15 11	10 6	5 0
AMOOR	0	LX_2	2_5	Sz_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	59_5

Sz_2		
0	AMOORB	byte
1	AMOORW	wyde
2	AMOORT	Tetra
3	AMOORO	octa

Clock Cycles:

AMOSWAP[.] – AMO Swap

Description:

Atomically swap source operand register Rs2 with value from memory. The original value of the memory cell is stored in register Rd. The memory address is in Rs1.

Supported Operand Sizes: .o

Instruction Formats: AMO

AMOSWAP Rd, Rs2, d[Rs1]

	31	3029	28 24	23 22	21 17	16	15 11	10 6	5 0
AMOSWAP	0	LX_2	6_5	Sz_2	$Rs2_5$	Cr	$Rs1_5$	Rd_5	59_5

Clock Cycles:

CMPSWAP[.] – Compare and Swap

Description:

If the contents of the addressed memory cell equals the contents of Rs2 then a 32-bit value is stored to memory from the source register Rs3. The original contents of the memory cell are loaded into register Rd. The memory address is contained in register Rs1. If the operation was successful then Rd and Rs2 will be equal. The compare and swap operation is an atomic operation.

Instruction Format:

	31	3029	28 24	23 22	21 17	16	15 11	10 6	5 0
CMPSWAP	0	0_2	0_2	$Rs3_5$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	60_5

Operation:

```
Rd = memory[[Rs1]]
if memory[[Rs1]] = Rs2
        memory[[Rs1]] = Rs3
```

Assembler:

CMPSWAP Rd, Rs2, Rs3, [Rs1]

Note:

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	1	Displacement$_{13\ldots0}$				Cr	Rs1$_5$	Rd$_5$	32$_5$
Indexed Ld	0	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	1	Displacement$_{13\ldots0}$				Cr	Rs1$_5$	Rd$_5$	33$_5$
Indexed Ld	0	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	1	LX_2	$\text{Displacement}_{11\ldots0}$			Cr	$Rs1_5$	Rd_5	36_5
Indexed Ld	0	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	0	LX_2	$\text{Displacement}_{11\ldots0}$			Cr	$Rs1_5$	Rd_5	37_5
Indexed Ld	0	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	1	LX_2	$Displacement_{11...0}$			Cr	$Rs1_5$	Rd_5	34_5
Indexed Ld	0	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	1	LX_2	$Displacement_{11...0}$			Cr	$Rs1_5$	Rd_5	35_5
Indexed Ld	0	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	1	LX_2	$Displacement_{11\ldots0}$			Cr	$Rs1_5$	Rd_5	38_5
Indexed Ld	0	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	1	LX_2	$Displacement_{11...0}$			U	$Rs1_5$	$Rs2_5$	40_5
Indexed St	0	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	1	LX_2	$Displacement_{11...0}$			U	$Rs1_5$	~	CL_3	41_5
Indexed St	0	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	1	LX_2	$Displacement_{11...0}$			U	$Rs1_5$	$Rs2_5$	46_5
Indexed St	0	LX_2	$Disp_5$	Sc_2	$Rs2_5$	U	$Rs1_5$	$Rs3_5$	46_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	1	LX_2	$Displacement_{11...0}$			U	$Rs1_5$	CL_4	~	47_5
Indexed St	0	LX_2	$Disp_5$	Sc_2	$Rs2_5$	U	$Rs1_5$	CL_4	~	47_5

Execution Units: AGEN, MEM

Exceptions:

Notes:

STPTR Rn, <ea> - Store Pointer

Description:

Store a pointer from register Rs3 to memory. Storing a pointer to memory also updates the hardware card table, indicating roughly where the pointer was stored.

Instruction Format:

Disp	1	LX_2	$Displacement_{11\ldots0}$			U	$Rs1_5$	$Rs3_5$	52_5
Indexed St	0	LX_2	$Disp_5$	Sc_2	$Rs2_5$	U	$Rs1_5$	$Rs3_5$	52_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	1	LX$_2$	Displacement$_{11\ldots0}$			U	Rs1$_5$	Rs2$_5$	44$_5$
Indexed St	0	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	1	LX$_2$	Displacement$_{11\ldots0}$			U	Rs1$_5$	~	CL$_3$	45$_5$
Indexed St	0	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	1	LX_2	$Displacement_{11…0}$			U	$Rs1_5$	$Rs2_5$	42_5
Indexed St	0	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	1	LX_2	$Displacement_{11…0}$			U	$Rs1_5$	~	CL_3	43_5
Indexed St	0	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:

Instruction Format:

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).

	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

Instruction Format:

This is an alternate mnemonic for the MOVE instruction where the source operand is r0 and the destination register is a carry register.

	31	3028	27			17	16	15 11	10 6	5 0
CLC	1	LX_2	0_3	0_5	0_{65}	Rd_{65}	Cr	$0_{4...0}$	$Rd_{4...0}$	15_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	1	0_2	0_4	$\sim_6$		I	$CRs1_6$	CRd_6	11_6
CRAND	0	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	1	0_2	3_4	$\sim_6$		I	$CRs1_6$	CRd_6	11_6
CRANDC	0	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	1	0_2	1_4	$\sim_6$		I	$CRs1_6$	CRd_6	11_6
CROR	0	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	1	0_2	2_4	$\sim_6$		I	$CRs1_6$	CRd_6	11_6
CRXOR	0	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/predicate 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]	1	D_{12}	BRs_3	Cnd_3	CRs_6		$Disp_{10\ldots3}$			BRd_3	12_5	D
[Dcc]Bcc[L]	0	0_2	BRs_3	Cnd_3	CRs_6	~	$Rs1_5$		$\sim_2$	BRd_3	12_5	
[Dcc]Bcc[L]	0	1_2	BRs_3	Cnd_3	CRs_6		$\sim_8$			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_3	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

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 ±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]	1	Displacement$_{24…3}$			BRd$_3$	13$_5$	D
BLR[L]	0	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 ±2^{24} bits. This is an alternate mnemonic for the BL instruction where the branch link register is BR0. BR7 (the PC) is assumed used in the destination address calculation.

Formats Supported: B

	31	3029	28 16	15 11	10 6	5 0	
B	1	Displacement$_{24\ldots3}$			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	1	D_{12}	BRs_3	2_3	CRs_3,1	$Disp_{10...3}$	BRd_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	1	D_{12}	BRs_3	2_3	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	1	D_{11}	BRs_3	2_3	$CRs,5_3$	$Disp_{10...3}$	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	1	D_{12}	BRs_3	2_3	$CRs,5_3$	$Disp_{10...3}$	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	1	D_{12}	BRs_3	5_3	$CRs,5_3$	$Disp_{10...3}$	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	1	D_{12}	BRs_3	5_3	$CRs,5_3$	$Disp_{10...3}$	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	1	D_{12}	BRs_3	5_3	$CRs,0_3$	$Disp_{10...3}$	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	1	D_{12}	BRs_3	5_3	$CRs,0_3$	$Disp_{10...3}$	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	1	D_{12}	BRs_3	2_3	$CRs,3_3$	$Disp_{10...3}$	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 ±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	1	D_{12}	BRs_3	2_3	$CRs,3_3$	$Disp_{10...3}$	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	1	D_{12}	BRs_3	2_3	$CRs,4_3$	$Disp_{10...3}$	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 ±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	1	D_{12}	BRs_3	2_3	$CRs,4_3$	$Disp_{10…3}$	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 ±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	1	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 32-bit displacement. This instruction may be used to return from a subroutine. The ‘L’ bit of the instruction specifies if a limit should be applied for the value in BRs.

Formats Supported: BL

BLR	0	0_2	BRs_3	$Limit_{13...2}$	~	$Rs1_5$	$\sim_2$	0_3	13_5	L
BLR	0	1_2	BRs_3	$Limit_{13...2}$	$\sim_4$	CL_3	~	0_3	13_5	L

Operation:

PC = PC + BRs + Constant

Execution Units: Flow Control

Clock Cycles: 13

Exceptions: none

Notes:

BLRI – Branch to Link Register and Interrupt

BLRI label[BRs],limit

Description:

This instruction always jumps to the destination address. The destination address is formed as the contents of BRs added to a 32-bit displacement or the contents of Rs1. The address of the next instruction is stored on an internal stack along with the status register.

Formats Supported: BL

BLRI	0	0_2	BRs_3	$\sim_3$	Om_3	$Swstk_4$	~	$Rs1_5$	$\sim_2$	7_3	13_5	0
BLRI	0	1_2	BRs_3	$\sim_3$	Om_3	$Swstk_4$	$\sim_4$	CL_3	~	7_3	13_5	0

Operation:

```
PUSH SR

PUSH PC + 4

IPL = 63

PC = BRs + $Constant_{32}$
OR
PC = BRs + Rs1
```

Execution Units: Flow Control

Clock Cycles: 13

Exceptions: none

Notes:

BLRL – Branch to Link Register and Link

BLRL BRd,label[BRs],limit

Description:

This instruction always jumps to the destination address. The destination address is formed as the contents of BRs added to a 32-bit displacement or the contents of Rs1. 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. The 'L' bit of the instruction specifies if a limit should be applied for the value in BRs.

Formats Supported: BL

BLRL	0	0_2	BRs_3	$Limit_{10…2}$	~	$Rs1_5$		$\sim_2$		BRd_3	13_5	L
BLRL	0	1_2	BRs_3	$Limit_{10…2}$	$\sim_4$		CL_3		~	BRd_3	13_5	L

Operation:

BRd = next PC

PC = BRs + $Constant_{32}$
OR
PC = BRs + Rs1

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	1	D_{12}	BRs_3	5_3	$CRs,4_3$	$Disp_{10…3}$	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 ±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	1	D_{12}	BRs_3	5_3	$CRs,4_3$	$Disp_{10…3}$	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	1	D_{12}	BRs_3	5_3	$CRs,3_3$	$Disp_{10\ldots3}$	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 ±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	1	D_{12}	BRs_3	5_3	$CRs,3_3$	$Disp_{10\ldots3}$	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	1	D_{12}	BRs_3	5_3	$CRs,1_3$	$Disp_{10\ldots3}$	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	1	D_{12}	BRs_3	5_3	$CRs,1_3$	$Disp_{10\ldots3}$	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	1	D_{12}	BRs_3	5_3	$CRs,2_3$	$Disp_{10...3}$	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	1	D_{12}	BRs_3	5_3	$CRs,2_3$	$Disp_{10...3}$	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	1	D_{12}	BRs_3	2_3	$CRs,0_3$	$Disp_{10\ldots3}$	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 $\pm2^{31}$ 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	1	D_{12}	BRs_3	2_3	$CRs,0_3$	$Disp_{10\ldots3}$	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	1	D_{12}	BRs_3	2_3	$CRs,2_3$	$Disp_{10...3}$	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	1	D_{12}	BRs_3	2_3	$CRs,2_3$	$Disp_{10...3}$	BRd_3	12_5	D

Clock Cycles: 13

BT –Branch Table

BT label[BRs],Limit

Description:

Branch to a destination calculated as the sum of a branch register and a displacement. The branch register contains an offset from the start of the table. The offset must be greater than zero and less than the limit. If outside of these bounds, then the entry at the table limit is branched to.

Formats Supported: BR

BT	1	D_{12}	BRs_3	$Limit_{13...2}$	D_2	$Disp_{10...3}$		0	30_6
BT	0	0_2	BRs_3	$Limit_{13...2}$	$\sim_2$	Rs1	$\sim_2$	0_3	30_6

Clock Cycles: 13

BTL –Branch Table and Link

BTL label[BRs],Limit

Description:

Branch to a destination calculated as the sum of a branch register and a displacement. The branch register contains an offset from the start of the table. The offset must be greater than zero and less than the limit. If outside of these bounds, then the entry at the table limit is branched to. The return address is stored in a link register BRd.

Formats Supported: BR

BTL	1	D_{12}	BRs_3	$Limit_{13...2}$	D_2	$Disp_{10...3}$		BRd_3	30_6
BTL	0	0_2	BRs_3	$Limit_{13...2}$	$\sim_2$	Rs1	$\sim_2$	BRd_3	30_6

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	1	D_{12}	BRs_3	5_3	$CRs,6_3$	$Disp_{10\ldots3}$	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	1	D_{12}	BRs_3	5_3	$CRs,6_3$	$Disp_{10\ldots3}$	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	1	LX_2	$Immediate_{12}$	0	$\sim_5$	0_5	63_6

Notes:

PEQ –Predicate if Equal

PEQ CRs, label[BRs]

Description:

Perform instructions under the predicate window if the predicate is true, otherwise skip over the instructions. Up to 11 instructions may be skipped over.

Formats Supported: BR

	31	3029	28		16	15 11	10 6	5 0	
PEQ	1	M_{109}	0_3	2_3	CRs,0_3	$M_{8...1}$	7_3	12_5	M

Clock Cycles: 13

PNE –Predicate if Not Equal

PNE CRs, label[BRs]

Description:

Perform instructions under the predicate window if the predicate is true, otherwise skip over the instructions. Up to 11 instructions may be skipped over.

Formats Supported: BR

	31	3029	28		16	15 11	10 6	5 0	
PNE	1	M_{109}	0_3	5_3	CRs,0_3	$M_{8\ldots1}$	7_3	12_5	M

Clock Cycles: 13

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.

RFI – Return from Interrupt or Exception

Description:

This is an alternate mnemonic for the ERET instruction. This instruction returns from an interrupt or exception routine by transferring program execution to the address stored in an internal stack.

Formats Supported: RFI

	31	3029	28 17	16	15 11	10 6	5 0
RFI	0	0_2	2_{12}	0	0_5	$Const_5$	0_6

Operation:

Optionally pop the status register and always pop the program counter from the internal stack. Add Const*4 bytes to the program counter. If returning from an application trap the status register is not popped from the stack.

Execution Units: Branch

Exceptions: none

Notes:

ECALL – Environment / Escalating Call

Description:

Perform an environment or escalating call. Interrupts are disabled. The program counter is reset to the contents of the vector loaded from svec[] CSR plus the vector number shifted left twice, associated with the next higher operating mode and instructions begin executing in the next higher operating mode. There should be a jump instruction placed at the vector location. The address of the instruction following the ECALL instruction is pushed onto an internal stack.

Instruction Format: SYS

	31	3029	28	17	16	15 11	10 6	5 0
ECALL	1	0_2	Vector_{12}		~	$\sim_5$	31_5	28_6
ECALL	0	0_2	$\sim_7$	Rs2_5	~	$\sim_5$	31_5	28_6

Operation:

PUSH SR onto internal stack
PUSH PC + 4 onto internal stack
PC = svec[om+1]+vector * 4
Om = om + 1

Execution Units: Branch

Clock Cycles:

Exceptions: none

Notes:

ERET – Environment Return

Description:

This instruction returns from an interrupt or exception routine by transferring program execution to the address stored in an internal stack.

Formats Supported: RFI

	31	30 29	28 17	16	15 11	10 6	5 0
ERET	0	0_2	2_{12}	0	0_5	$Const_5$	0_6

Operation:

Optionally pop the status register and always pop the program counter from the internal stack. Add Const*4 bytes to the program counter. If returning from an application trap the status register is not popped from the stack.

Execution Units: Branch

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	~$_7$	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 (environment 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.

VIRT2PHYS[.] – Convert Virtual Address to Physical Address

Description:

This instruction uses the MMU to convert a virtual address to a physical one. It is similar to a load operation except that the address is stored in the destination register instead of data from memory. This instruction may use values in any register and store to any register. The address to convert is supplied in Rs1 and the converted address is stored in Rd.

Since the instruction goes through the MMU, the page fault handler may be triggered if a translation is not available. This instruction may be used to trigger the page fault handler without accessing memory.

Instruction Formats:

	31	3028	27			17	16	15 11	10 6	5 0
VIRT2PHYS	1	LX$_2$	0_3	2_5	Rs1$_{65}$	Rd$_{65}$	Cr	Rs1$_{4\ldots0}$	Rd$_{4\ldots0}$	15_6

Operation: R2

Rd = address of [Rs1]

Clock Cycles:

could be several clock cycles depending on the MMU state. More if the page fault handler is triggered.

Execution Units: Agen / MMU

Exceptions: none

Notes:

Macro Instructions

ENTER – Enter Routine

Description:

This instruction is used for subroutine linkage at entrance into a subroutine. First it optionally pushes the frame pointer and return address onto the stack, next the stack pointer is loaded into the frame pointer, finally saved registers are stored to the stack. This instruction is code dense, replacing eight or more other instructions with a single instruction.

The stack and frame pointers are assumed to be r31 and r30 respectively.

Note that the instruction reserves room for two words in addition to the return address and frame pointer. One use for the extra words may to store exception handling information.

Integer Instruction Format:

	31	3029	28	17	1 6	1514	13 11	10 6	5 0
ENTER	1	3_2	Opt_3	$Uimm_9$	0	$Rs2_5$		$Rs1_5$	30_6

Opt_3	Format	
Bit 2	Fxx	Store fp
Bit 1	xBx	Store br1
Bit 0	xxZ	Zero stack locs

Operation:

```
SP = SP - 32
If (F) Memory[SP] = FP
If (B) Memory8[SP] = BR1
IF (Z) Memory16[SP] = 0        # zero out catch handler address
If (Z) Memory32[SP] = 0
If (F) FP = SP
If (Rs2 <> 0)
    cnt = Rs2 – Rs1 + 1
    cnt2 = 0
    SP = SP – cnt * 8
```

```
          for (cnt2 = 0; cnt > 0; cnt--, cnt2++)
          Memory[SP+cnt2*8] = Rs1 + cnt2
      SP = SP - Uimm
```

EXIT – Exit Routine

Description:

This instruction is used for subroutine linkage at exit from a subroutine. It reverses the operations performed by ENTER. First it pops the specified saved registers from the stack. Next it moves the frame pointer to the stack pointer deallocating any stack memory allocations. Next the frame pointer and return address are optionally popped off the stack. The stack pointer is adjusted by the amount specified in the instruction. Then if the 'R' bit of the instruction is set, a jump is made to the return address. This instruction is code dense, replacing between six and sixteen other instructions with a single instruction. The stack pointer adjustment is multiplied by eight keeping the stack pointer word aligned.

Instruction Format:

	31	3029	28			17	1 6	1514	13 11	10 6	5 0
EXIT	1	3_2	F	B	R	$Immediate_9$	0	$Rs2_5$		$Rs1_5$	31_6

Operation:

```
If Rs2 <> 0
        count = Rs2 – Rs1 + 1
        cnt2 = 0
        While (cnt2 < count)
                Rs1 + cnt2 = Memory[SP+cnt2 * 8]
                cnt2= cnt2 + 1
SP = SP + count * 8
If (F)
        SP = FP
        FP = Memory[SP]
If (B) BR1 = Memory8[SP]
SP = SP + 32
SP = SP + $Constant_{31}$ * 8
If (R)
        PC = Br1
```

POP – Pop Registers from Stack

Description:

This instruction pops up to eighteen registers from the stack. The registers must all be from the same group. There are 12 overlapping groups of registers. Registers are popped off the stack according to the operating mode.

Instruction Format:

	31	3029	28	17	1 6	1514	13 11	10 6	5 0
POP	1	2_2	Gr_2	$Rd_{17...8}$	0	Gr_2	$Rd_{7...0}$		31_6

Operation:

PUSH – Push Registers on Stack

Description:

This instruction pushes up to eighteen registers onto the stack. The registers must all be from the same group. Registers are pushed on the stack according to the operating mode.

Instruction Format:

	31	3029	28	17	16	1514	13 11	10 6	5 0
PUSH	1	2_2	Gr_2	$Rs_{17...8}$	0	Gr_2	$Rs_{7...0}$		30_6

Operation:

Pre/Postfixes and Modifiers

ACARRY Modifier

Description:

Atomic carry performs the operations under the carry window in an atomic fashion with interrupts disabled. Carry specifies the propagation of carry values from one instruction to the next. The register to use for carry is encoded in the instruction. The connections between carry inputs and outputs are specified with a bit-pair mask. A maximum of eight instructions may be covered by the ACARRY modifier.

Instruction Format:

	31	3029	28		16	15 11	10 6	5 0	
ACARRY	0	0_2	3_3	CY_2	$Mask_{15…1}$		7_3	12_5	M

CY_2: specifies one of three carry registers, legal values are from 1 to 3.

	Mask Bit	
Carry Modifier Scope	0,1	Instruction zero
	2,3	Instruction one
	4,5	Instruction two
	6,7	Instruction three
	8,9	Instruction four
	10,11	Instruction five
	12,13	Instruction six
	14,15	Instruction seven

Mask Bit	
00	Ignore carries
01	Carry in from register CY_2
10	Carry out to register CY_2
11	Carry in and out using CY_2

Assembler Syntax:

Example:

```
CARRY CY2,”-OIOIOIO--”
ADD r1,r2,r3
ADD r4,r5,r6
ADD r7,,r8,r9
ADD r10,r11,r12
```

ATOM Modifier

Description:

Treat the following sequence of instructions as an “atom”. The instruction sequence is executed with interrupts masked off, except for an NMI. 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		16 15 11	10 6	5 0	
ATOM	0	0_2	1_3	63_6	$\text{Mask}_{11\ldots1}$	7_3	12_5	M

Assembler Syntax:

Example:

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

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

CARRY Modifier

Description:

Carry specifies the propagation of carry values from one instruction to the next. The register to use for carry is encoded in the instruction. The connections between carry inputs and outputs are specified with a bit-pair mask. A maximum of eight instructions may be covered by the CARRY modifier.

Instruction Format:

	31	3029	28		16 15 11	10 6	5 0	
CARRY	0	0_2	2_3	CY_2	$Mask_{15\ldots1}$	7_3	12_5	M

CY_2: specifies one of three carry registers, legal values are from 1 to 3.

	Mask Bit	
Carry Modifier Scope	0,1	Instruction zero
	2,3	Instruction one
	4,5	Instruction two
	6,7	Instruction three
	8,9	Instruction four
	10,11	Instruction five
	12,13	Instruction six
	14,15	Instruction seven

Mask Bit	
00	Ignore carries
01	Carry in from register CY_2
10	Carry out to register CY_2
11	Carry in and out using CY_2

Assembler Syntax:

Example:

```
CARRY CY2,”-OIOIOIO--”
ADD r1,r2,r3
ADD r4,r5,r6
ADD r7,,r8,r9
ADD r10,r11,r12
```

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 32 registers may be selected.

Instruction Format: QEXT

Instruction Format: ATOM

	31	3029	28		17	16	15 11	10 6	5 0
QEXT	0	LX_2	4_3	$\sim_4$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	8_6
QFEXT	0	LX_2	5_3	$\sim_4$	$Rs2_5$	Cr	$Rs1_5$	Rd_5	8_6

FREGS Prefix

Description:

This prefix adds registers for the following floating-point instruction.

Note that any of 32 registers may be selected.

Instruction Format

	31	3029	28		17	16	15 11	10 6	5 0
FREGS	0	LX_2	7_3	$\sim_4$	$FRs4_5$	Cr	$FRs3_5$	$FRd2_5$	8_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	1	Immediate$_{27\ldots5}$				Wh$_2$	61$_6$

Notes:

~~PRED Modifier (deprecated)~~

~~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

	Mask Bit		Rn_8 Bits Tested
Pred Modifier Scope	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

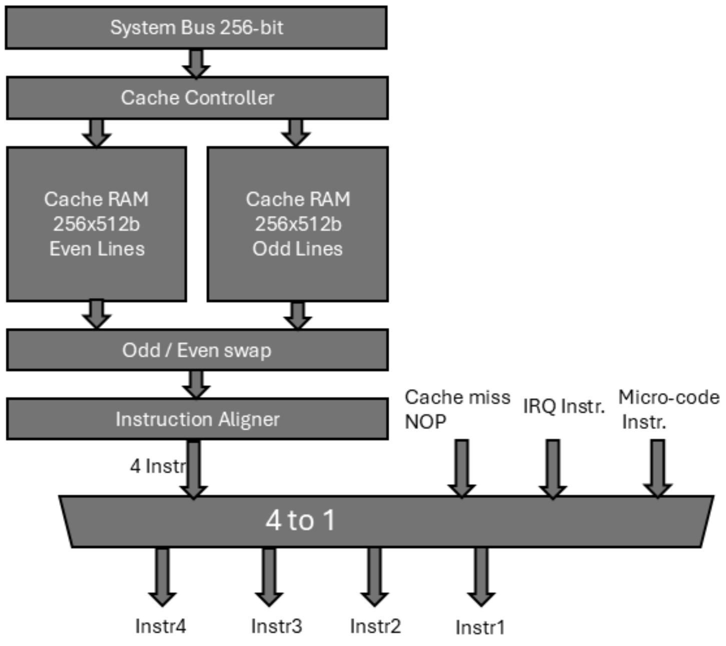
The core has both instruction and data caches to improve performance. Both caches are single level. The cache is four-way associative. The cache sizes of the instruction and data cache are available for reference from one of the info lines return by the CPUID instruction.

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.

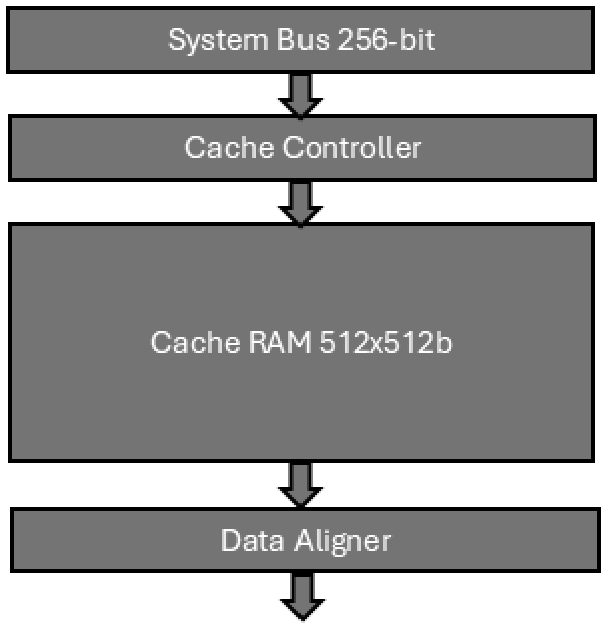
While the L1 cache has single cycle reads it requires two clock cycles to update (write) the cache. The cache line to update needs to be provided by the tag memory which is unknown until after the tag updates.

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.

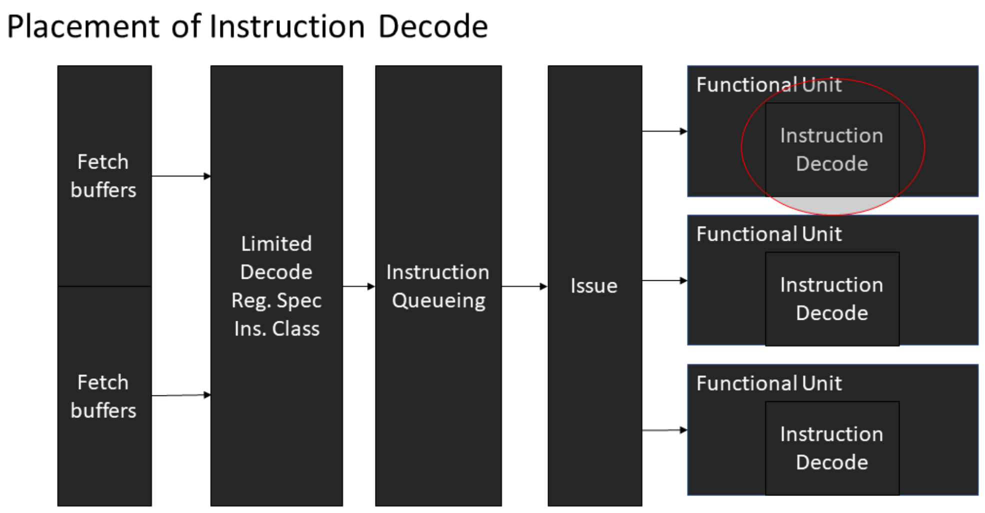
Correctly predicting a branch turns the branch into a single cycle operation. During execution of the branch instruction the address of the following instruction queued is checked against the address depending on the branch outcome. If the address does not match what is expected, then the queue will 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.

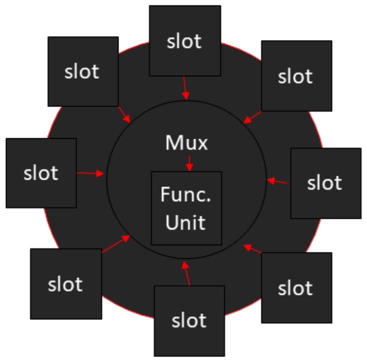


Limited decode takes place between fetch and queue. Between fetch and queue register specifications are decoded along with general instruction classes for the benefit of issue. A handful of additional signals (like sync) that control the overall operation of the core are also decoded. Much of the instruction 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

Regions

In any processing system there are typically several different types of storage assigned to different physical address ranges. These include memory mapped I/O, MMIO, DRAM, ROM, configuration space, and possibly others. Qupls3 has a region table that supports up to eight separate regions.

The region table is a list of region entries. Each entry has a start address, an end address, an access type field, and a pointer to the PMT, page management table. To determine legal access types, the physical address is searched for in the region table, and the corresponding access type returned. The search takes place in parallel for all eight regions.

Once the region is identified the access rights for a particular page within the region can be found from the PMT corresponding to the region. Global access rights for the entire region are also specified in the region table. These rights are gated with value from the PMT and TLB to determine the final access rights.

Region Table Location

The region table in Q+3 is located in the QMMU.

Page Allocation Map – PAM

The page allocation map is a bitmap of pages of memory that have been allocated. It is managed by software. For the test system there are 49152 pages of memory available to the application. This requires 1536 words of memory to represent the page map.

Page Management Table - PMT

Overview

The page management table, PMT, is a software managed table that contains information pertaining to individual memory pages. There is a separate PMT for each memory region. Pieces of information include the key needed to access the page, the privilege level, and read-write-execute permissions for the page. The table is organized as rows of table entries (PMTEs). There are as many PMTEs as there are pages of memory in the region.

Location

The page management table is in main memory and may be accessed with ordinary load and store instructions. The PMT address is specified by the region table.

PMTE Description

There is a wide assortment of information that goes in the page management table. To accommodate all the information an entry size of 128-bits was chosen.

Page Management Table Entry

V	N	M	$\sim_9$	C	E	AL_2	$\sim_{12}$	urwx
ACL_{16}							Share $Count_{16}$	
Access $Count_{32}$								
PL_8			Key_{24}					

Access Control List

The ACL field is a reference to an associated access control list.

Share Count

The share count is the number of times the page has been shared to processes. A share count of zero means the page is free.

Access Count

This part uses the term ‘access count’ to refer to the number of times a page is accessed. This is usually called the reference count, but that phrase is confusing because reference counting may also refer to share counts. So, the phrase ‘reference count’ is avoided. Some texts use the term reference count to refer to the share count. Reference counting is used in many places in software and refers to the number of times something is referenced.

Every time the page of memory is accessed, the access count of the page is incremented. Periodically the access count is aged by shifting it to the right one bit.

The access count may be used by software to help manage the presence of pages of memory.

Key

The access key is a 24-bit value associated with the page and present in the key ring of processes. The keyset is maintained in the keys CSRs. The key size of 20 bits is a minimum size recommended for security purposes. To obtain access to the page it is necessary for the process to have a matching key OR if the key to match is set to zero in the PMTE then a key is not needed to access the page.

Privilege Level

The current privilege level is compared with the privilege level of the page, and if access is not appropriate then a privilege violation occurs. For data access, the current privilege level must be at least equal to the privilege level of the page. If the page privilege level is zero anybody can access the page.

N

indicates a conforming page of executable code. Conforming pages may execute at the current privilege level. In which case the PL field is ignored.

M

indicates if the page was modified, written to, since the last time the M bit was cleared. Hardware sets this bit during a write cycle.

E

indicates if the page is encrypted.

AL

indicates the compression algorithm used.

C

The C indicator bit indicates if the page is compressed.

urwx, srwx, hrwx, mrwx

These are read-write-execute flags for the page.

Qupls3 Hierarchical Page Table Setup

Overview

Qupls3 hierarchical page tables are setup like a tree. Branch pages contain pointers to other pages and leaf pages contain pointers to block of memory that an application has access to. The entries in branch pages are referred to as page table pointers, PTPs, since they point at other page tables. The entries in leaf pages are referred to as page table entries, PTEs.

Page Table Entry Format – PTE

The PTE occupies 64-bits or eight bytes. 2048 PTEs will fit into an 16kB page. A physical address range maximum of 2^{58} bytes of memory may be mapped.

63	6261	6058	5755	54	53	5250	49 48	47	46 44	43 0
V	T_2	LVL_3	RGN_3	M	A	AVL_3	$CACHE_2$	SW_1	$URWX_3$	$PPN_{43…0}$

Page Table 1MB Entry Format – PTE

This PTE format may map up to 2^{64} bytes of contiguous memory using 1MB pages.

63	6261	6058	5755	54	53	5250	49 48	47	46 44	43 0
V	1_2	LVL_3	RGN_3	M	A	AVL_3	$CACHE_2$	SW_1	$URWX_3$	$PPN_{49…6}$

Field	Size	Purpose
PPN	44	Physical page number
URWX	3	User read-write-execute
SRWX	1	Supervisor write protect
CACHE	2	Cache location
AVL	3	OS software usage
A	1	1=accessed/used
M	1	1=modified
RGN	3	Memory region
LVL	3	Page table level pointed at
T	2	2 = PTP, 0 = PTE, 1 = 1MB page PTE
V	1	1 if entry is valid, otherwise 0

QMMU – Qupls3 Memory Management Unit

Overview

The Qupls3 memory management unit is a device in the CPU used to manage pages of memory. It encompasses a region table and a page table walker.

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 $FFF40001 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.

REG_BAR1 defaults to $FFF403C1 which is used to specify the address of the region table registers in the I/O address space.

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

All registers are 64-bit and only 64-bit accessible unless otherwise noted.

Regno	Access	Moniker	Purpose
			Bounds Registers
0000	RW	PBL0	63 … 32: Limit$_{45...14}$; 31 … 0: Base$_{45...14}$
…	…	…	14 other bounds registers
0078	RW	PBL15	
0080 to 0FF8	…	…	31 other sets of bounds registers.
			Region Table
3C00	RW	STADR	Start address of region
3C08	RW	ENADR	End address of region
3C10	RW	PAM	Page allocation map address – PAM
3C18	RW	PMT	Associated PMT address (region #0)
3C20	RW	CTA	Card table address (region #0)
3C28	RW	LOCK	Lock status – “LOCK” or “UNLK” * this register 32-bit accessible
3C2C			
3C30	RW	RGATTR	Four groups of 32-bit memory attributes, 1 group for each of user, supervisor, hypervisor and secure. (region #0) These registers are 32-bit accessible
3C34			
3C38			
3C3C			
3C40 to 3DF8			7 more sets of registers for regions #1 to #7 same as 3C00 to 3C38
3F00	R	PFA	Page fault address
3F08	R	~	Zero (address expansion)

3F10	R	PFS	Fault segment
3F18	R	~	Reserved
3F20	R	PFAS	Fault asid (in bits 48 to 63)
3F28	RW	PTBR	Page table base address register
3F30	R	~	Zero (address expansion)
3F38	RW	PTATTR	Page Table attributes

Bits		
0 to 2	Type	0=hierarchical,1=hash
3 to 6	Page size	Size = 6+page size bits
7,8	PTE size	0=4,1=8,2=16 bytes
9		reserved
10		reserved
11,12		Replacement algorithm
13 to 15	Level	Entry level into hierarchical table
16 to 31	Limit	Root page table limit # entries
32 to 63		reserved

3F40	RW	VIRTADR	Virtual address to translate
3F48	R	~	Zero (address expansion)
3F50	R	PHYSADR	Physical address for virtual address
3F58	R	~	Zero (address expansion)
3F60	R	PADRV	Translated physical address is valid
3F68	RW	BNDX	Bits 0 to 4 register set index, selects active bounds registers

PBL0 to PBL15

This set of sixteen registers establishes boundaries within which a program may operate. Only address bits 14 to 59 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. The four MSBs of the virtual memory address are not considered as they select the bounds register. The test system has 49152 pages of memory (under 1GB).

Register Format

Regno	Moniker	63	32	31	0
0xB00	PBL0	$\sim_{18}$	Limit$_{27…14}$	$\sim_{18}$	Base$_{27…14}$
0xB08	PBL1	$\sim_{18}$	Limit$_{27…14}$	$\sim_{18}$	Base$_{27…14}$
…	…	$\sim_{18}$		$\sim_{18}$	
0xB78	PBL15	$\sim_{18}$	Limit$_{27…14}$	$\sim_{18}$	Base$_{27…14}$

PMT Address

The PMT address specifies the location of the associated PMT for a region.

CTA – Card Table Address

The card table address is used during the execution of the store pointer, STPTR instruction to locate the card table.

Attributes for a Region

Bitno		
0	X	may contain executable code
1	W	may be written to
2	R	may be read
3	~	reserved
4-7	C	Cache-ability bits
8-10	G	granularity

G	
0	byte accessible
1	wyde accessible
2	tetra accessible
3	octa accessible
4	hexi accessible
5 to 7	reserved

11	~	reserved
12-14	S	number of times to shift address to right and store for telescopic STPTR stores.
16-23	T	device type (rom, dram, eeprom, I/O, etc)
24-31	~	reserved

PFA Register

This register holds the virtual address for which a page fault occurred. A fault occurs if no valid address translation can be found in the tables. Accessing this register will clear an outstanding page fault.

PTBR – Page Table Base Address Register

This register points to the root page table in memory. It needs to be saved and restored during context switches.

PTATTR

This register contains details of the organisation of the MMU. The default PTE size is eight bytes. This should be reduced to four bytes during system initialization for Qupls3.

VIRTADR Register

Software may request a virtual to physical address translation using this register. Writing to the register causes the corresponding physical address to be looked up then placed in the PHSYADR register. Since it may take some time for the lookup the PADRV bit should be checked to ensure the physical address is a valid one.

QIC – Qupls3 Interrupt Controller

Overview

The Qupls3 system uses message-signaled interrupts (QMSI). 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

18	RW	MVTB	Base address for hypervisor machine vector table
20	RW	VTL	Vector table limit
28	RW	STAT	Bit 0: Que full, set if any que is full, cleared by software if written with a zero 1: Set if stuck interrupt detected 2 to 62: reserved 63: Set if an interrupt is being requested
30	R	QUEL	Top output of the priority queues, bits 0 to 63
38	R	QUEH	Top output of the priority queues, bits 64 to 127
40	R	EMP	Queue empty status, one bit for each queue, 1=empty
48	R	OVR	Queue overflow status, one bit for each queue, 1=overflowed
380	RW	GE	Bit 0 = global interrupt enable
390	RW	THRES	Interrupt threshold (0 to 63), IRQ priority must exceed this to be recognized.
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.			
800	RW	AFNx	CPU group, one bit for each CPU that should be notified
…	RW		More CPU groups
FF8	RW		Last CPU group
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.			
1000	RW	IP	Interrupt enable bits
…			More IE bit registers
13F8	RW	IP	
1400	RW	IE	Interrupt pending bits
…			More interrupt pending bits
17F8	RW	IE	

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:

127 112	111 104	103 101	100 98	97	96	95 0
Data$_{16}$	CPU group$_{8}$	OM$_{3}$	Swstk$_{3}$	IE	AI	Address$_{64}$ or Instruction$_{96}$

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

OM: This field indicates which operating mode should handle 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	
pv	1	0=physical, 1=virtual address	
Padr	32/64	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:		