

Q+ A 64-bit homebrew superscalar processor

Qupls

Version 4 Guide

Robert Finch

TABLE OF CONTENTS

Preface	21
Who This Book is For	21
About the Author.....	21
Motivation.....	21
History.....	22
Features of Qupl4.....	22
Getting Started.....	23
Choosing an Implementation Language.....	23
Support Tools.....	23
Documenting the Design.....	24
Building the System.....	24
Software for the Target Architecture.....	24
Testing and Debugging.....	25
Test Benches	25
Using Emulators.....	27
Bootstrap Code vs the “Real Code”	27
Data Alignment	28
Get Rid of Complexity	28
Disabling Interrupts.....	28
The IRQ Live Indicator.....	28
Disable Caching	29
Clock Frequency	29
More Advanced Debugging Options.....	29
Debug Registers.....	29
Trace / Program Counter History.....	29
Stuck on a Bug?	30
The Rare Chance	30
Nomenclature	30
Design Choices.....	31
RISC vs CISC	31
Little Endian vs big Endian.....	31
Endian	31
Deciding on the Degree of Pipelining	32
Choosing a Bus Standard	33

Choosing an ISA	33
Readability	33
Planning for the future.....	33
Opcode / Instruction Size:	34
Variable Length Instruction Sets.....	34
Instruction Bundles	34
Data Size	35
Registers.....	35
Number of General-Purpose REgisters.....	35
Register Access.....	36
Segment Registers.....	36
Other Registers	36
Moving Register Values	37
Register Usage	37
Handling Immediate Values.....	37
SETHI.....	38
IMMxx - Prefix	38
IMMxx - POSTfix	39
LW Table	39
Half or SHIFTED Operand Instructions	39
The Branch Set.....	40
Branch Targets	40
Branch Prediction	41
Looping Constructs	41
Other Control Flow Instructions.....	41
Subroutine Calls.....	41
Returning From Subroutines.....	42
System Calls	42
Returning from Interrupt Routines	42
Jumps	43
Conditional Moves.....	43
Predicated Instruction Execution	43
Comparison Results	43
Dual Operation Instructions	44
Arithmetic Operations	44

Logical Operations	44
Shift Instructions	44
Mystery Operations	45
Other Instructions	47
Exception Handling	47
Hardware Interrupts	47
Interrupt Vectoring	48
Interrupt Vector Table	48
Getting and Putting Data	48
Aligned and Unaligned Memory Access	49
Load / Store Multiple	49
The Stack	49
Data Caching	50
Address Modes	50
Scaled Index Addressing	51
Register Indirect with Displacement Addressing	52
Support for Semaphores	52
Memory Management	53
Segmentation and Paging	53
Segmentation Overview	53
Paging Overview	54
Protection Mechanisms	56
Protection Rules	57
Triple Mode Redundancy (TMR)	57
Performance Measurement / Counters	58
TICK count	58
Power Management	58
Floating Point	59
Precision	59
Operations	59
Floating Point Number Format	59
Floating Point Registers	60
Pipeline Design	60
Processor Stages / States	60
RESET	61

IFETCH	61
IALIGN.....	61
Extract / PARSE	61
DECODE	61
Register File Access.....	61
Rename	62
Enqueue	62
Schedule.....	62
EXECUTE	62
Memory Stage.....	63
Writeback.....	63
Commit	63
Instruction Cache	63
Nice-to-Have Hardware Features.....	63
CPU Block Diagram.....	65
Programming Model.....	66
Register File	66
General Purpose Registers	66
Logical Registers.....	67
Floating-Point Registers.....	67
Physical Registers	67
Register Flags.....	67
Predicate Registers.....	68
Link Registers	68
Loop Counter	68
Instruction Pointer.....	68
SR - Status Register (CSR 0x?004)	69
Vector Programming Model.....	70
Vector Registers.....	70
Tail Elements	70
Vector Masking.....	70
Vector Operands	71
Unsupported Functionality.....	71
Special Purpose Registers	72
[U/S/H/M]_IE (0x?004).....	72

[U/S/H/M]_CAUSE (CSR- 0x?006).....	72
U_FPCSR - Floating Point Status and Control Register (CSR 0x0014).....	72
U_FXCSR – Fixed Point Control and Status (CSR – 0x0015).....	73
[U/S/H/M]_SCRATCH – CSR 0x?041	74
S_ASID (CSR 0x101F).....	74
S_KEYS (CSR 0x1020 to 0x1027).....	74
M_CORENO (CSR 0x3001)	75
M_TICK (CSR 0x3002)	75
M_SEED (CSR 0x3003).....	75
M_BADADDR (CSR 0x3007)	75
M_BAD_INSTR (CSR 0x300B)	75
M_SEMA (CSR 0x300C)	75
M_KVEC – Kernel Vectors (CSR 0x3030 to 0x3034).....	76
M_SR_STACK (CSR 0x3080 to CSR 0x3087)	76
M_IOS – IO Select Register (CSR 0x3100)	76
M_CFGS – Configuration Space Register (CSR 0x3101).....	76
M_EIP (CSR 0x3108 to 0x310F).....	76
U_VLEN – (CSR 0x0200) - Vector Length	78
U_VELSZ – (CSR 0x0201) - Vector Element Size.....	78
U_VSTART – (CSR 0x0202) - Vector Start	78
U_NANCR – (CSR 0x0210) – NaN Control Register.....	78
Operating Modes	80
Exceptions	81
External Interrupts.....	81
Effect on Machine Status	82
Exception Stack.....	87
Vector Table.....	83
Breakpoint Fault (0).....	85
Single Step Fault (1)	85
Bus Error Fault (2).....	85
Address Error Fault (3)	85
Unimplemented Instruction Fault (4).....	85
Privilege Violation Fault (5)	85
Page Fault (6).....	85
Stack Canary Fault (8)	85

Abort (9)	85
Interrupt (10).....	85
Non-Maskable Interrupt (11).....	85
Reset Vector (12).....	85
Alternate Cause (13)	86
Reset.....	87
Precision.....	87
Hardware Description.....	88
Caches	88
Overview.....	88
Instructions	88
L1 Instruction Cache.....	89
Data Cache	90
Capabilities Tag Cache	90
Cache Enables.....	90
Cache Validation.....	90
Un-cached Data Area.....	90
Fetch Rate	90
Return Address Stack Predictor (RSB)	91
Branch Predictor.....	92
Branch Target Buffer (BTB)	92
Decode Logic	92
Instruction Queue (ROB)	93
Queue Rate.....	94
Sequence Numbers	94
Input / Output Management.....	95
Device Configuration Blocks	95
Reset.....	95
Devices Built into the CPU / MPU	95
Memory Management.....	96
Bank Swapping	96
The Page Map	96
Regions.....	96
Region Table Location.....	96
Region Table Description	97

PMA - Physical Memory Attributes Checker	98
Overview.....	98
Page Management Table - PMT.....	99
Overview.....	99
Location	99
PMTE Description	99
Access Control List.....	99
Share Count.....	99
Access Count	99
Key.....	100
Privilege Level.....	100
N	100
M.....	100
E.....	100
AL.....	100
C.....	100
urwx, srwx, hrwx, mrwx	100
Page Tables	101
Intro.....	101
Hierarchical Page Tables	101
Inverted Page Tables.....	101
The Simple Inverted Page Table.....	102
Hashed Page Tables	102
Shared Memory.....	103
Specifics: Qupl Page Tables.....	103
Qupl Hash Page Table Setup.....	103
Qupl Hierarchical Page Table Setup	106
TLB – Translation Lookaside Buffer	108
Overview.....	108
Size / Organization.....	108
TLB Entries - TLBE	108
Small TLB Entries - TLBE	108
What is Translated?.....	109
Page Size.....	109
Ways	109

Management.....	109
?RWX ₃	109
CACHE ₄	109
TLB Entry Replacement Policies.....	109
Flushing the TLB.....	110
Reset	110
PTW - Page Table Walker	110
Page Table Base Register.....	110
Page Table Attributes Register	111
Card Table.....	112
Overview.....	112
Organization.....	112
Location	112
Operation	112
Sample Write Barrier	113
Instruction Set.....	114
Overview.....	114
Code Alignment	114
Root Opcode.....	114
Register Specs	115
The Register Type Field - V	115
Destination register Spec	115
Source Register Spec	116
Secondary Opcode	116
Primary Function Code	116
Precision.....	116
Constant Field Spec.....	117
Clock Cycles	117
Execution Units.....	117
Instruction Descriptions.....	119
Basic Instruction Formats.....	119
Arithmetic Operations	120
Representations.....	120
Arithmetic Operations.....	121
ABS – Absolute Value.....	122

ADC – Add With Carry	124
ADD – Add Register-Register	125
ADD.xP – Add Parallel Register-Register.....	127
ADDI - Add Immediate	129
ADDJO – Add, Jump on Overflow.....	130
ADDIP - Add Immediate to Instruction Pointer	132
BYTENDX – Character Index.....	133
CHARNDX – Character Index	134
CHK – Check Register Against Bounds	135
CHKCPL – Check Code Privilege Level.....	137
CNTxx – Count.....	138
CPUINFO – Get CPU Info	140
CSR – Control and Special Registers Operations	142
DIV – Signed Division.....	143
DIVI – Signed Immediate Division	144
DIVU – Unsigned Division	145
DIVUI – Unsigned Immediate Division	146
LOADA – Load Address	147
MAJ – Majority Logic	148
PTRDIF – Difference Between Pointers.....	149
REM – Signed Remainder	151
REMU – Unsigned Remainder	152
REVBIT – Reverse Bit Order	153
SATADD – Saturating Add Register-Register	154
SBC – Subtract With Carry.....	156
SQRT – Square Root	157
SUB – Subtract Register-Register.....	158
SUBFI – Subtract from Immediate	160
TETRANDX – Character Index	161
WYDENDX – Character Index	162
Vector Arithmetic Operations	163
VABS – Absolute Value	163
VADD – Add Register-Register	164
VCNTxx – Count.....	165
VDIV – Signed Division.....	166

VDIVU – Unsigned Division.....	167
VMASK – Mask Result	168
VREM – Signed Remainder	169
VSATADD – Saturating Add Register-Register	170
VSUB – Subtract Register-Register.....	171
Examples:	172
Multiply.....	175
BMM – Bit Matrix Multiply	176
CLMUL – Carry-less Multiply	177
MUL_xx – Multiply Register-Register.....	178
MULW – Multiply Widening	180
MULI - Multiply Immediate.....	181
MULSU_xx – Multiply Signed Unsigned	182
MULSUW – Multiply Signed Unsigned WIDENING	183
MULU_xx – Unsigned Multiply Register-Register.....	184
MULUW – Unsigned Multiply WIDENING	185
MULUI - Multiply Unsigned Immediate	186
Vector Multiply.....	187
VCLMUL – Carry-less Multiply	187
VMUL_xx – Multiply Register-Register.....	188
VMULU_xx – Unsigned Multiply Register-Register.....	189
Data Movement.....	190
Scalar Data Movement.....	190
Vector Data Movement.....	210
Logical Operations.....	222
Scalar Operations.....	222
Vector Logical Operations	234
Vector Reduction Operations	243
Overview.....	243
Arithmetic Operations.....	243
Logical Operations.....	247
Data Movement Operations	250
Comparison Operations.....	255
Overview.....	255
CMP - Comparison	255

CMPI – Compare Immediate	257
CMPU – Unsigned Comparison	258
CMPUI – Compare Unsigned Immediate	260
SEQ – Set if Equal	261
SEQM – Masked Set if Equal	263
SLE – Set if Less Than or Equal	264
SLEU – Set if Unsigned Less Than or Equal	266
SLT – Set if Less Than	268
SLTU – Set if Unsigned Less Than	270
SNE – Set if Not Equal	272
Vector Compare Operations	274
VCMP - Comparison	274
VSEQ – Set if Equal	277
VSLT – Set if Less Than	280
VSLTU – Set if Less Than Unsigned	282
Shift and Rotate Operations	283
Overview	283
Scalar Shift Instructions	283
Vector Shift Instructions	292
Bit-field Manipulation Operations	297
CLR – Clear Bit Field	300
COM – Complement Bit Field	301
DEP – Deposit Bit Field	302
EXT – Extract Bit Field	303
EXTU – Extract Unsigned Bit Field	304
SET – Set Bit Field	305
Cryptographic Accelerator Instructions	306
AES64DS – Final Round Decryption	306
AES64DSM – Middle Round Decryption	306
AES64ES – Final Round Encryption	307
AES64ESM – Middle Round Encryption	307
SHA256SIG0	308
SHA256SIG1	309
SHA256SUM0	310
SHA256SUM1	311

SHA512SIG0	312
SHA512SIG1	312
SHA512SUM0	313
SHA512SUM1	313
SM3P0	314
SM3P1	315
Neural Network Accelerator Instructions	316
Overview	316
NNA_MFACT – Move from Output Activation	317
NNA_MTBC – Move to Base Count	318
NNA_MTBIAS – Move to Bias	319
NNA_MTFB – Move to Feedback	320
NNA_MTIN – Move to Input	321
NNA_MTMC – Move to Max Count	322
NNA_MTWT – Move to Weights	323
NNA_STAT – Get Status	324
NNA_TRIG – Trigger Calc	325
Floating-Point Operations	327
Precision	327
Representations	327
NaN Boxing	329
Rounding Modes	329
FABS – Absolute Value	331
FADD – Float Addition	332
FCLASS – Classify Value	333
FCMP - Comparison	334
FCONST – Load Float Constant	336
FCVTD2Q – Convert Double to Quad Precision	338
FCVTH2D – Convert Half to Double Precision	339
FCVTQ2D – Round Quad to Double Precision	340
FCVTQ2H – Round Quad to Half Precision	341
FCVTQ2S – Round Quad to Single Precision	342
FCVTS2D – Convert Single to Double Precision	343
FCVTS2Q – Convert Single to Quad Precision	344
FCX – Clear Floating-Point Exceptions	345

FDIV –Float Division	346
FDP –Float Dot Product.....	347
FDX – Disable Floating Point Exceptions	348
FEX – Enable Floating Point Exceptions.....	349
FINITE – Number is Finite.....	350
FMA –Float Multiply and Add	351
FMAX –Float Maximum Value.....	352
FMIN –Float Minimum Value.....	353
FMS –Float Multiply and Subtract	354
FMUL –Float Multiplication	355
FNEG – Negative Value	356
FNMA –Float Negate Multiply and Add	357
FNMS –Float Negate Multiply and Subtract	358
FRES – Floating point Reciprocal Estimate	359
FRM – Set Floating Point Rounding Mode	360
FRSQRT – Float Reciprocal Square Root Estimate	361
FSCALEB –Scale Exponent	362
FSEQ – Float Set if Equal.....	363
FSGNJ – Float Sign Inject	364
FSGNJP – Float Negative Sign Inject	365
FSGNJPX – Float Sign Inject Xor	366
FSIG – Get Significand of Number	367
FSIGMOID – Sigmoid Approximate.....	368
FSIGN – Sign of Number	369
FSINCOS – Float Sine And Cosine	370
FSLE – Float Set if Less Than or Equal	371
FSLT – Float Set if Less Than	372
FSNE – Float Set if Not Equal.....	373
FSQRT – Floating point square root.....	374
FSUB –Float Subtraction.....	375
FTOI – Float to Integer	376
FTRUNC – Truncate Value	377
FTX – Trigger Floating Point Exceptions.....	378
ISNAN – Is Not a Number.....	379
ITOF – Integer to Float	381

Vector Floating-Point Operations	382
VFABS – Absolute Value.....	382
VFADD –Float Addition	383
VFCMP - Comparison	384
VFCVTS2D – Convert Single to Double Precision.....	386
VFDP – Float Dot Product.....	387
VFMA –Float Multiply and Add	388
VFMAX –Float Maximum Value.....	389
VFMASK –Float MASK	390
VFMIN – Float Minimum Value	391
VFMUL –Float Multiplication.....	392
VFNEG – Negative Value	393
VFSEQ – Float Set if Equal.....	395
VFSGNJ – Float Sign Inject	396
VFSGNJN – Float Negative Sign Inject	397
VFSGNJX – Float Sign Inject Xor	398
Decimal Floating-Point Instructions	402
Overview.....	402
Precision	402
Representations.....	402
DFADD – Add Register-Register	404
DFMUL – Multiply Register-Register.....	405
DFNEG – Negate Register.....	406
DFSEQ – Decimal Float Set if Equal	406
DFSIG – Get Significand of Number	407
DFSUB – Add Register-Register.....	408
DFTOI – Decimal Float to Integer.....	408
ITODF – Integer to Decimal Float.....	409
Memory Operation Instructions	410
Overview.....	410
Addressing Modes	410
Atomic Memory Operations	411
Load Operations.....	420
Store Operations	432
Miscellaneous Memory Operations	437

Vector Memory Operations	443
Block Instructions	451
Overview.....	451
BCOMPARE – Block Compare	452
BCANCEL – Cancel Block Operation	454
BCOUNT – Get Block Count	455
BFIND – Block Find.....	456
BMOVE –Block Move	457
BSTATUS – Block Operation Status.....	458
BSTORE – Block Store	459
Vector Specific Instructions	460
Overview.....	460
VGMASK – Generate Vector Mask	461
VSHLV – Shift Vector Left (Vector Slide Up).....	297
VSHRV – Shift Vector Right (Vector Slide Down)	298
Vector Specific Instructions	462
VEINS / VMOVSV – Vector Element Insert	462
VEX / VMOVS – Vector Element Extract	463
VGNDX – Generate Index.....	464
VMFILL – Vector Mask Fill	465
VMFIRST – Find First Set Bit.....	465
VMLAST – Find Last Set Bit.....	466
Branch / Flow Control Instructions	468
Overview.....	468
Conditional Branch Format.....	468
Branch Conditions	469
Branch Target	471
Unconditional Branches.....	472
BAND –Branch if Logical And True.....	473
BANDB –Branch if Bitwise And True	474
BBC – Branch if Bit Clear	475
BBS – Branch if Bit Set	476
BENOR –Branch if Equal.....	477
BEOR –Branch if Not Equal.....	478
BEQ –Branch if Equal	479

BGE –Branch if Greater Than or Equal	480
BGEU –Branch if Unsigned Greater Than or Equal	481
BGT –Branch if Greater Than	482
BGTU –Branch if Unsigned Greater Than	482
BHI –Branch if Higher.....	483
BLE –Branch if Less Than or Equal.....	483
BLEU –Branch if Unsigned Less Than or Equal	484
BLT –Branch if Less Than	485
BLTU –Branch if Unsigned Less Than	485
BNAND –Branch if Logical And False	486
BNE –Branch if Not Equal	487
BNOR –Branch if Logical Or False.....	487
BOR –Branch if Logical Or True	488
BORB –Branch if Bitwise Or True.....	488
BTW –Branch Three-Way	489
BRA – Branch Always.....	490
BSR – Branch to Subroutine.....	491
FBEQ –Branch if Equal	492
FBGE –Branch if Greater Than Or Equal.....	493
FBGT –Branch if Greater Than	493
FBLE –Branch if Less Than Or Equal.....	494
FBLT –Branch if Less Than	495
FBNE –Branch if Not Equal	496
BUGT –Branch if Unordered or Greater Than	497
JMP – Jump to Target.....	498
JSR – Jump to Subroutine.....	501
JTT – Jump Through Table.....	502
NOP – No Operation.....	503
RET – Return from Subroutine	504
RTD – Return from Subroutine and Deallocate.....	505
RTS – Return from Subroutine	505
Graphics Instructions.....	507
BLEND – Blend Colors	507
TRANSFORM – Transform Point.....	508
Processor Queue Management Instructions	511

Overview.....	511
PEEKQ – Peek at Queue / Stack.....	512
POPQ – Pop from Queue / Stack	512
PUSHQ – Push on Queue / Stack	513
READQ – Read From Queue / Stack.....	513
RESETQ – Reset Queue / Stack	514
STATQ – Get Status of Queue / Stack.....	514
WRITEQ – Write to Queue / Stack	515
System Instructions	516
BRK – Break.....	516
ESC – Escalate Exception.....	517
FENCE – Synchronization Fence	518
IRQ – Generate Interrupt	519
JMPX – Jump to Exception Handler.....	520
MEMDB – Memory Data Barrier.....	521
MEMSB – Memory Synchronization Barrier	521
PFI – Poll for Interrupt.....	522
REX – Redirect Exception.....	523
RTE – Return from Exception	525
SYS – System Call.....	526
STOP – STOP Processor	527
TRAP – Trap.....	528
Subroutine Instructions	529
ENTER – Enter Routine	529
EXIT – Exit Routine	530
Constant Support.....	531
CZ – Constant Zone	531
Modifiers / Postfixes	532
ATOM.....	532
PRED	533
REXT – Extended Register Selection.....	535
ROUND	536
Opcode Maps.....	537
Qupl Root Opcode	537
{CAP} Operations	538

{R1} Operations	539
{R3, R3P, R3SV} Operations.....	540
{EXTD} Extended Precision Instructions	541
{CMOV} Conditional Move Instructions.....	541
{FLT, FLTP} Operations.....	541
{DFLT3} Operations	542
{DFLT} Operations	542
{AMO} – Atomic Memory Ops	543
{EX} Exception Instructions	543
MPU Hardware.....	544
PIC – Programmable Interrupt Controller.....	544
Overview.....	Error! Bookmark not defined.
System Usage.....	Error! Bookmark not defined.
Priority Resolution.....	Error! Bookmark not defined.
Config Space.....	Error! Bookmark not defined.
Registers	Error! Bookmark not defined.
Control Register.....	Error! Bookmark not defined.
PIT – Programmable Interval Timer	548
Overview.....	548
System Usage.....	548
Config Space.....	548
Parameters.....	549
Registers	550
Programming	552
Interrupts.....	552
Glossary	553
ABI.....	553
AMO	553
Assembler.....	553
ATC.....	553
Base Pointer	553
Burst Access.....	553
BTB.....	553
Card Memory	554
Commit.....	554

Decimal Floating Point.....	554
Decode	554
Diadic	554
Endian	554
FIFO	554
FPGA	555
Floating Point.....	555
Frame Pointer.....	555
HDL	555
HLL.....	555
Instruction Bundle.....	555
Instruction Pointers	555
Instruction Prefix.....	556
Instruction Modifier	556
ISA	556
JIT	556
Keyed Memory.....	556
Linear Address	556
Machine Code	556
Milli-code.....	556
Monadic	556
Opcode	557
Operand.....	557
Physical Address	557
Physical Memory Attributes (PMA)	557
Posits	557
Program Counter	557
RAT.....	557
Retire	557
ROB	558
RSB	558
SIMD.....	558
Stack Pointer	558
Telescopic Memory.....	558
TLB	558

Trace Memory	559
Triadic	559
Vector Chaining	559
Vector Length (VL register).....	559
Vector Mask (VM).....	559
Virtual Address	559
Writeback	559
Miscellaneous	560
Reference Material	560
Trademarks.....	560
WISHBONE Compatibility Datasheet	562
FTA Bus	564
Overview.....	564
Bus Tags.....	564
Single Cycle	564
Retry.....	564
Signal Description	564
Requests	564
Responses	565
Om	565
Cmd.....	565
BTE.....	566
CTI.....	566
Blen.....	567
Sz	567
Segment	567
TID.....	568
Cache	568

PREFACE

WHO THIS BOOK IS FOR

This book is for the FPGA enthusiast who's looking to do a more complex project. It's advisable that one have a good background in digital electronics and computer systems before attempting a read. Examples are provided in the SystemVerilog language, it would be helpful to have some understanding of HDL languages. Finally, a lot about computer architecture is contained within these pages, some previous knowledge would also be helpful. If you're into electronics and computers as a hobby FPGA's can be a lot of fun. This book primarily describes the Qupls ISA. It is for anyone interested in instruction set architectures.

ABOUT THE AUTHOR

First a warning: I'm an enthusiastic hobbyist like yourself, with a ton of experience. I've spent a lot of time at home doing research and implementing several soft-core processors, almost maniacally. One of the first cores I worked on was a 6502 emulation. I then went on to develop the Butterfly32 core. Later the Raptor64. I have progressed slowly from the simple to the complex. I have about 25 years professional experience working on banking applications at a variety of language levels including assembler. So, I have some real-world experience developing complex applications. I also have a diploma in electronics engineering technology. Some of the cores I work on these days are too complex and too large to do at home on an inexpensive FPGA. I await bigger, better, faster boards yet to come. To some extent larger boards have arrived. The author is a bit wary of larger boards. Larger FPGAs increase build times by their nature.

MOTIVATION

The author desired a CPU core supporting 128-bit floating-point operations for the precision. He also wanted a core he could develop himself. The simplest approach to supporting 128-bit floats is to use 128-bit wide registers, which leads to 128-bit wide busses in the CPU and just generally a 128-bit design. It was not the author's original goal to develop a 128-bit machine. There are good ways of obtaining 128-bit floating-point precision on 64-bit or even 32-bit machines, but it adds some complexity. Complexity is something the author must manage to get the project done and a flat 128-bit design is simpler.

Good single thread performance is also a goal. To achieve good single thread performance multiple instructions must be able to execute simultaneously. A superscalar processor of some sort may be required.

Having worked on Thor2023 for several months, the author finally 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. Thor2022 was better in that regard. So, Thor2024 arrived and is a mix of the best from previous designs. Thor2024 aimed to improve code density over earlier versions. Qupls code density is slightly worse than Thor2024 but offers potentially better performance.

Some efficiency is being traded off for design simplicity. Some of the most efficient designs are 32-bit.

The processor presented here isn't the smallest, most efficient, and fastest RISC processor. It's also not a simple beginner's example. Those weren't my goals. Instead, it offers reasonable performance and hopefully design simplicity. It's 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. I chose a 64-bit design supporting 128-bit ops 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

Qupls4 is a work in progress beginning November 2025. It sprang from Qupls November 2023. 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.

Qupls's graphics engine originate from the ORSoC GFX accelerator core posted at opencores.org by Per Lenander, and Anton Fosselius.

FEATURES OF QUPLS4

- Fixed 48-bit instructions.
 - *The design has gone through several iterations of variable length instructions. A fixed length instruction set makes the design simpler and seems to require less hardware.*
- Four way out-of-order superscalar operation.
 - *One of the goals was high performance single thread execution.*
- Four operating modes, secure (machine), hypervisor, supervisor, and user/app.
 - *Multiple operating modes are a requirement for modern software.*
- 64-bit data path, support for 128-bit floats.
 - *64-bit values are needed for 64-bit addressing.*
- 16 (or more) entry re-order buffer
- 32 general purpose registers.
 - *This is standard in many new architectures.*
 - *Performance often is not improved even if there are more registers available. The physical register file size needs to be large enough to support the number of architectural registers. The physical register file has 512 registers. Using three per architectural register for renaming purposes means there should not be more than 170 architectural registers. 128 of those are to support the vector register file. That leaves 42 registers. Selecting 32 registers fits into a five bit register code.*
- The register file is unified; it may contain either integer or float data.
- Independent control of sign for each register.
 - *Sign control removes the need for NEG or NOT instructions and helps reduce the dynamic instruction count, improving performance.*
- Constants may be substituted for register values in most instructions.
 - *Large constants are embedded in the instruction stream following the instruction. An instruction may have multiple constants.*
- Register renaming to remove dependencies, vector chunks are also renamed.
 - *A chunk is 64-bits and may contain multiple elements depending on the size.*
- Dual operation instructions, $Rd = Rs1 \text{ op1 } Rs2 \text{ op2 } Rs3$
 - *Dual operation instructions are not commonly used. They came about as there were a few extra bits available in the instruction. They have the potential to double performance if they can be used; they may allow up to eight operations per clock cycle. The issue is finding uses.*

- Standard suite of ALU operations, add subtract, compare, multiply and divide.
- Simple ALU ops performed on the FPU.
 - *Allowing simple ALU ops to be performed on the FPU increases the level of parallelism in the machine which should increase performance.*
- Arithmetic right shift with rounding.
- Bitfield operations.
- Conditional branches with 21 effective displacement bits.
- Vector Operations
- 32 x 256-bit or 16 x 512-bit vector registers build option.
 - *This is a build option. As the design is pretty much limited to 170 architectural registers there are not many choices about the vector format. If a wider data-path were available, the size of the vector register would increase without requiring more architectural registers.*
- 128 Entry Two-way TLB shared between data and code.
- Paged memory management.
- Message signaled interrupt handling.

GETTING STARTED

To get started designing an ISA or CPU core some basic tools are required.

CHOOSING AN IMPLEMENTATION LANGUAGE

You will need a high-level hardware description language (HDL) of some sort to develop a processor. For a complex project it is not uncommon for there to be several different languages involved.

It is a good idea to become accustomed to any number of languages. It helps to review the work of others and a lot can be learned by studying code from existing projects.

Choosing a language is somewhat of a personal choice. One should choose whatever works best for themselves. There are two popular HDL languages (Verilog, and VHDL) and number of others. I encourage you to search the web for HDL languages and find something you're comfortable with. Additional languages include things like Java or C++ classes that people have developed to output HDL. Or language translators such as a 'C' to Verilog translator, for people who wish to work in 'C'. Not everybody speaks the same language as easily as everybody else, and it does have a little bit to do with linguistics. I know some people who will only work with schematics. My personal favorite is System Verilog. VHDL is more verbose than Verilog and has tighter control of types. Qupls is implemented in the System Verilog HDL language.

SUPPORT TOOLS

One wouldn't be able to achieve anything without the appropriate supporting toolsets. If you can't get your hands on the tools required to do the work you may have to roll some of your own. It can be quite an investment and it's up to you to decide. You have the power and control over your hobby. Many thanks to the vendors who supply free toolsets for use with their FPGA's. One may have to develop one's own tools to some extent. It's almost like a circus performance to get one's own toolsets working well. Is it the processor that's broken? or the toolset? That program didn't work because the assembler didn't assemble it correctly, it wasn't a bug in the processor. Keeping everything 'in sync' is like a dance, one goes around and around in circles. The author has had to develop his own assembler, disassembler, compiler, glyph editing program and other things. It's more involved than one might anticipate to begin with. For instance, to get character display on-screen a glyph editor was needed. The author looked at a couple of free ones

available on the net, but they didn't quite do what was needed. Something was needed that could output FPGA vendor compatible files, and the free glyph editors were geared towards graphics files formats. After spending about a day trying to modify an existing editor, the author gave up and decided to roll his own. The author first developed a simple assembler about 30 years ago for use at school; it is still using the same source code with many, many updates. The assembler has become quite powerful now.

DOCUMENTING THE DESIGN

Any processor design is likely to have a few documents associated with it. One needs to be able to refer to things like what opcode does what, outside of the implementation code itself. For general tasks the author is using MS Office. Word for word processing, and Excel for spreadsheets. OpenOffice is another toolset that may be used. A spreadsheet is handy for representing tables like opcode tables. One will likely need some sort of word processor that supports tables for documentation purposes. A simple text editor probably isn't enough.

BUILDING THE SYSTEM

To produce an implementation some sort of FPGA developer tools will be required. The FPGA devices typically must be programmed with a bit file generated by tools supplied by the FPGA vendor. It's the vendors who know the requirements for programming their devices; the author does not know of any third-party software that can generate bitstreams from source code. The author has used both free toolsets from Altera and Xilinx.

SOFTWARE FOR THE TARGET ARCHITECTURE

The problem with an original home-grown processor is that there's no software for it. Fortunately, there is a lot of free software with source code available on the internet. One of the first things one will need is an assembler for the target architecture. One can assemble opcodes by hand with a reference chart handy, but it gets boring quickly. The author usually ends up doing some hand assembly to do some simple tests on the processor before the assembler is working. Then he takes an existing assembler and modifies it for the new processor. One assembler found on the net is for the 6809 (listed in the resources section) was modified for a 6809-enhancement core. The author has two assemblers one written in C++ the other in Visual Basic. Visual Basic's a little easier to work in for string handling. Some sort of text scripting language is a good place to start with a simple assembler. Some projects use Python. Much (older) software is written in C. It's a good language to know.

Another approach for a simple assembler is to use a spreadsheet program. If the instruction set is simple enough, it may be possible to have the spreadsheet calculate them as needed.

Once an assembler is working there are other languages that may be useful and easy to adapt. The author has adapted a version of Tiny Basic to several different homebrew projects now. Forth is another language popular with small systems. Once some of the simpler pieces of software are working, one may want to try one's hand at a toolset.

There are several toolsets available that can be utilized during development of soft-core processors like Qupls. One of these is the LCC compiler. The LCC compiler was used for the Butterfly32 project. It's straightforward to implement the compiler for a new ISA especially if your ISA is similar to an existing one. Another toolset is the gcc compiler. The author has not put this toolset to use yet, but has had a look at it. It seems somewhat daunting. GCC is very general in nature and supports a lot of target architectures. People have put a lot of work into making this compiler available for any architecture. The author knows a number of people have been turned off by the complexity, however. The compiler the author uses a fair bit

is a modified 68000 ‘C’ compiler that was found on the net a while ago. One may have to study compilers for a while before being able to modify one or create one oneself. Compilers tend to be complex, and if you want good results for an original ISA you will have to write a good part of a compiler yourself. Not to worry, many homebrew projects get by without a compiler.

TESTING AND DEBUGGING

A lot of testing is required to get something working. This section seems short for the amount of testing the author does. 90% of the work is in the testing. But this is a book about implementing or developing a processor, not a book about testing. Whole books could easily be written about testing. The key to avoiding backtracking and wasted time down the road is lots of testing along the way. Every bug fix is a test. When one bug is fixed, the next one shows up. Sometimes they seem like a two-headed hydra. Good testing skills are a requirement for developing and debugging a processor. Once you’ve managed to get such a thing working you’re probably an ace at testing. Sometimes the processor and programming cannot help you to find a bug in the processor itself. You must be able to think in terms of ‘what test can I do ?’ to fix the bug. There are usually at least several wow-zzy bugs. For example, the author found a bug where a register exchange instruction only failed on a cache miss, when the instruction was at the end of a cache line. Many programs worked fine, and the processor seemed not to work intermittently. It took quite a while to find. The author finally noticed the instruction failed when the cache was turned off. So, one thing to try for testing is turning the cache on or off.

TEST BENCHES

If you’re going to build it there must be some way to perform testing. The author recommends writing a test-bench first and trying the code in a simulator before trying out the code in an FPGA. A test bench is an artificial environment setup specifically to test a component. Inputs simulating a real environment are sent to the component then the output of the component is monitored for correctness. In the test bench usually so-called corner cases are tested, which are cases testing the extremes to which the component should work. If the component works in the extremes of the test bench it’ll certainly work when it’s put to real use is the general idea. A simulator is a tool built specifically for running test benches. The simulator has features to aid in debugging logic. One may set breakpoints, points which force the logic to stop at a particular place, and view the outputs of a component.

A simple test bench for the Thor divider circuit is shown below. Note that most test bench files don’t have any input or output ports. Instead, signals are selected in the simulator for viewing.

In this case arguments for the divider were manually altered in the test bench to check for specific cases.

```
module Thor_divider_tb();
parameter WID=64;
reg rst;
reg clk;
reg ld;
wire done;
wire [WID-1:0] qo,ro;

initial begin
```

```

clk = 1;
rst = 0;
#100 rst = 1;
#100 rst = 0;
#100 ld = 1;
#150 ld = 0;
end

always #10 clk = ~clk; // 50 MHz

```

```

Thor_divider #(WID) u1
(
    .rst(rst),
    .clk(clk),
    .ld(ld),
    .sgn(1'b1),
    .isDivi(1'b0),
    .a(64'd10005),
    .b(64'd27),
    .imm(64'd123),
    .qo(qo),
    .ro(ro),
    .dvByZr(),
    .done(done)
);

```

endmodule

Note that it is possible to automate test cases and even use file I/O in some tools. Test benches can become quite complex. The author feels that one should not lose sight of the goal while developing test benches. The test bench is just for testing; it is not the project itself. Test benches for the float components often use a test input file containing the operands for the design under test, DUT, and output the results along with the input operands in a results output file. The output file can then be studied at leisure for issues to correct. Having a file output allows different revisions of the core to be compared and may make regression testing easier.

It is extremely unlikely that one would get the HDL code perfect the first time. The processor is not likely to be working, so how do you fix it up? One needs debugging dumps of course, and those are only

available from a simulator. Judiciously placed debug output can be real aid to getting the CPU working. Unless a fix-up is minor and well-known, the author runs simulator traces before attempting to run the code in an FPGA.

As a first test running software code in the FPGA try something simple like turning an LED on or off. One of the first lines of code Table888 executes is:

```
start
    sei                                ; disable interrupts
    ld       r1, #$FF
    st       r1, LEDS
```

which turns on all the LEDs on the board.

This idea is popular for debugging hardware. The IBM PC had a “post-code” which was a byte value periodically written to an I/O port during startup for debugging. Depending on the display of the byte one could tell where in start-up it failed. Something like a missing or bad display adapter would end up with a specific code.

Another suggestion for test-benches is to use the actual system being loaded into the FPGA device as a component of the test-bench. If one keeps the system simple enough to start with then it’s possible to debug using the test-bench.

USING EMULATORS

An invaluable tool for debugging software prior to the processor being finished is the software emulator. A software emulator is an emulation of the device or system written as a software program to run on a workstation. Software emulators are often significantly slower than the real hardware. It’s also a tool where events applied to the system can be generated by user input. The code for the software emulation of a system mirrors the code for processor implementation itself. The code is just written in a different language. Having an emulator available allows for consistency checks between the emulation and the “real” device. Ideally the emulator should produce the same results as the real device would, except that it’s in a virtual environment of the emulator. The emulator can help resolve software problems that would be too difficult to do using the logic simulator.

Emulators can be cycle-exact, meaning they emulate what happens during each cycle of the processor’s clock. Cycle-exact emulators are often slower than non-cycle exact ones. An emulator that is not cycle exact may only emulate running software, interpreting object code, rather than performing all the internal operations that the CPU does.

BOOTSTRAP CODE VS THE “REAL CODE”

The next thing to do after getting simpler I/O tests working is more complex I/O like a video display. Being able to display things on-screen can be invaluable (a character LCD display or LED display works well too). Many low-cost FPGA boards come with numeric LED displays for output and buttons for input. It’s slightly more challenging to drive a numeric display and may make a good second test. Also being able to get a keystroke can be valuable too. One of the first routines my processors execute is the clear-screen routine. If it can’t clear the screen, I know something’s seriously wrong in the start-up. While the blue screen-of-death may be a bad sign, it’s a good sign at least the processor is working that much. When setting the processor software up (bootstrapping) don’t go for the most complex algorithms to begin with.

Go with simple things. I have two versions of keyboard routines. The one that ‘works the right way’ and the one I use for bootstrapping. The bootstrapping routine goes directly to the keyboard port to read a character. It’s very simple and pauses the whole machine waiting for a character.

DATA ALIGNMENT

Are your variables mysteriously getting over-written? There could be a problem with address generation in the processor, or perhaps a problem with the external address decoding.

One approach to aligning data structures in memory is to ensure that the structures don’t have partially overlapping addresses. This may help if there are memory addressing problems. For instance, if data structure addresses all end in xxx000, then if there is an address decoding problem, all the structures may get overwritten by values intended for other variables. If the variable addresses are somewhat mangled for example 0xxxx004, xx1018, xx2036 (ending in different LSB’s) then it may be less likely for data to be corrupted. This is a temporary debugging approach. One would want to have the var’s properly listed in a program.

GET RID OF COMPLEXITY

One of the best ways to be able to debug something is to get rid of all the extra complexities involved with it. Many is the time that the author has backtracked on a project and removed features in favor of getting something to work. Add one feature at a time, make it a component that can be easily disabled or removed from the design. Disable the complex features of the design. It’s great to be able to do a complex design. But all the complicated stuff started out small and simple. One doesn’t need caches, interrupts, branch predictors, and so on to have a working design. It’s very rewarding to have even the simplest design working.

DISABLING INTERRUPTS

This bit only applies if you’ve managed to get some sort of interrupt facility working. Several smaller, simpler systems don’t make use of interrupts. The original Apple computer did not use interrupts. Interrupts aren’t something that one must get working right away. They would be part of a longer-term project goal (if at all). Start small and simple and expand from there. There are alternatives to interrupts the main one being polling in a loop.

When working with the real hardware having a set of switches available can be invaluable. The switches can be wired to key signals in the design to offer a manual override option. There may be times when one desires to disable a feature under development while other aspects of the project are taking place. For instance, eventually at some point in time one might want to venture into the world of interrupt processing. Interrupts are a challenge to get working. It’s nice to be able to disable interrupts using an external switch. Also, there are times when one wants to know if the processor is capable of executing a linear sequence of instructions, without the interference of interrupts. Debugging the processor with interrupts enabled can be tricky. Development of an interrupt system is something for a later stage of development. Get the processor running longer sequences of code successfully first before trying to deal with interrupts.

THE IRQ LIVE INDICATOR

The IRQ live indicator is one of the first debug techniques the author uses once the core can run some code. An indicator that IRQ’s are happening seems like a friendly image. It can be useful to see that IRQ’s are happening on a regular basis. An IRQ indicator can let one know if the machine is just busy, or really, really stuck. This can be accomplished by incrementing a character at a fixed location on-screen. If that

character stops flipping around, one knows there's real trouble. Another common approach is to use an LED to indicate the presence of IRQ's. Turning a LED on and off at a low frequency can be handy to visually detect the presence of IRQs.

DISABLE CACHING

This tip applies only if a cache is present. Implementing a cache isn't priority number one. The first few projects the author did, did not include any caching. It was too complex to add a cache to begin with. As mentioned before, it sometimes necessary to disable the cache. Nice-to-have instructions are a cache-on and cache-off instruction. The processor should end up with the same results regardless of whether caching is enabled. If results seem flaky try disabling the cache.

CLOCK FREQUENCY

Be conservative when choosing a clock frequency. Don't try to run at the fastest possible frequency until the design is thoroughly debugged. Sometimes changing the clock frequency will provide clues to timing or synchronization problems. If the problem varies with a change in clock frequency, then maybe it's a timing problem. If the problem is consistent regardless of the clock frequency, it's likely some other problem. Note we are dealing with debugging probabilities here. Just because a problem is consistent at different clock frequencies doesn't mean it's not a timing problem.

Another nice aspect of a conservative clock frequency is that the tools used for building the system often work much faster if it's easy for the tools to meet the timing requirements. A conservative clock frequency is a way to speed up the development cycle.

MORE ADVANCED DEBUGGING OPTIONS

The following debugging mechanisms fall under the category of being more sophisticated in nature and more difficult to do, but they can sometime prove invaluable. They require interrupts or exceptions.

DEBUG REGISTERS

One option that aids primarily software debugging is the presence and use of debug registers. Adding debug registers to the core may make software debugging easier to do. Typically, there are one or more address matching registers that cause an interrupt or exception when the processor's program counter or data address matches the one in the debug register. One must have a working interrupt system for this to be usable.

TRACE / PROGRAM COUNTER HISTORY

One of the debug facilities that the author has added to cores is the capability to capture the history of the program counter. While the processor is running at full speed, the program counter is stored in a small history table which is usually some sort of shift register. When an exceptional condition occurs in the processor core the history capture is turned off. In the exception processing routine, the program counter history can then be dumped to the screen showing where the program went awry.

The technique is called "trace". A good trace history will often be able to be triggered perhaps at a specific address or via debug match register. The trace may record all instructions, but it is common to record only the branch history, and then a few of the instruction addresses for synchronization purposes. Since branches are either taken or not taken a single bit can be used to record the history making trace very compact. With only a couple of block RAMs a trace history of thousands of instructions is possible.

STUCK ON A BUG?

This is a brain trick. Try changing the code around the bug. Sometimes just by changing the code, refactoring without really changing operation, you will be able to spot a bug that wasn't readily apparent. It's a bit like moving your eyes around on the horizon to try and spot an enemy. The action of changing or simply moving the code causes a bug to pop out, out of the shadows.

THE RARE CHANCE

There is a rare chance that it's a problem in the toolset. A problem like this can make things really difficult, especially if it's a free toolset with no technical support. In about 20 years or so, of using toolsets the author has found a few bugs. The toolsets generally speaking are superb, so the chance of it being a bug in a toolset is extremely remote but not impossible. The one bug run into was in extending a complement of a single bit value. The toolset returned a binary "10" the value two when a single bit was being inverted. It should have returned a zero. The author was able to work around this problem by zero extending the value manually. The author found the bug by tracking the location of it down and dumping values using debug outputs.

Bugs in toolsets are often obvious. The most recent one caused the toolset to crash and quit running depending on how simulation was started. There was a work-around by restarting the simulation fresh every time which takes longer than the usual restart.

If you suspect a bug in the toolset try searching the web for information on it. If it's a common problem it's bound to be posted on the web somewhere. There are also usually forums on the web where one can post about problems, and even sometimes get replies.

NOMENCLATURE

There has been some mix-up in the naming of load and store instructions as computer systems have evolved. A while ago, a "word" referred to a 16-bit quantity. This is reflected in the mnemonics of instructions where move instructions are qualified with a ".w" for a 16-bit move. Some machines referred to 32-bits as a word. Times have changed and 64-bit workstations are now more common. In the author's parlance a word refers to the word size of a machine, which may be 16, 32, 64 bits or some other size. What does ".w" or ".d", and ".l" refer to? To some extent it depends on the architecture.

The ISA refers to primitive object sizes following the convention suggested by Knuth of using Greek.

Number of Bits		Instructions	Comment
8	byte	LDB, STB	UTF8 usage
16	wyde	LDW, STW	
32	tetra	LDT, STT	
64	octa	LDO, STO	

128	hexi	LDH, STH	
-----	------	----------	--

The register used to address instructions is referred to as the instruction pointer or IP register. The instruction pointer is a synonym for program counter or PC register.

DESIGN CHOICES

For something as complex as a CPU there are many design choices to be made. The hard part is not deciding what to include, but what to leave out. Almost anything could be included.

RISC VS CISC

No computer book would be complete without mentioning the RISC vs CISC paradigms.

There are two extremes to processor architecture. Most machines fall somewhere in-between. Qupls is somewhere in-between, leaning towards being a RISC machine, but it hardly has a reduced instruction set. At the extreme end of RISC the architecture may support as little as single instruction, or just a handful like eight or sixteen. At the other extreme a CISC architecture may support thousands of instruction variants. RISC architectures are typically load/store, large register array, and few instructions of a fixed format size. CISC architectures tend to have memory operands, varying register array sizes, lots of instructions of varying formats and sizes. The goal behind a RISC architecture is high performance by using a simple processor that operates at a high clock frequency. The goal behind a CISC architecture is high performance by providing a more customized instruction set. CISC architectures may combine multiple operations into a single instruction attempting to increase performance. Examples include stack linkage instructions, looping constructs, and complex memory addressing modes. Qupls has some of the better features from a CISC style machine.

LITTLE ENDIAN VS BIG ENDIAN

One choice to make is whether the architecture is little endian or big endian. There's a never-ending argument by computer folks as to which endian is better. It looks to the author that little endian architectures have won. There are more little-endian architectures in widespread use. In reality they are about the same or there wouldn't be an argument. In a little-endian architecture, the least significant byte is stored at the lowest memory address. In a big-endian architecture the most significant byte is stored at the lowest memory address. The author is partial to little endian machines; it just seems more natural to him although he knows people who swear by the opposite. Whichever endian is chosen, often the machine has instructions(s) for converting from one endian to the other. The author does not bother with endian conversion; it's a feature that he probably wouldn't use. Some implementations even allow the endian of the machine to be set by the user. This seems like overkill to the author. The endian of data is important because some file types depend on data being in little or big-endian format. Qupls is a little-endian machine.

ENDIAN

Qupls is a little-endian machine. The difference between big endian and little endian is in the ordering of bytes in memory. Bits are also numbered from lowest to highest for little endian and from highest to lowest for big endian.

Shown is an example of a 32-bit word in memory.

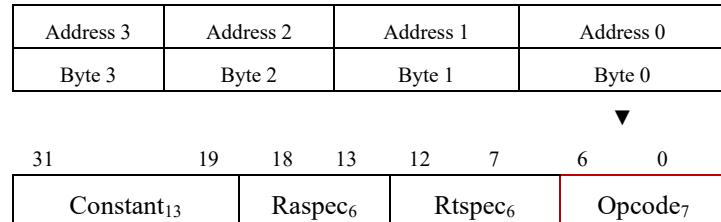
Little Endian:

Address	3	2	1	0
Byte	3	2	1	0

Big Endian:

Address	3	2	1	0
Byte	0	1	2	3

For Qupls the root opcode is in byte zero of the instruction and bytes are shown from right to left in increasing order. As the following table shows.



DECIDING ON THE DEGREE OF PIPELINING

How much pipelining is going to be done can impact the instruction set architecture (ISA). Some things are easier or harder to do depending on the pipelining present. For instance, handling large constants in an overlapped-pipelined design can be tricky, so one may want to stick with specific approaches. If one wants to support complex addressing mode such as memory indirect indexed it may be a lot easier to implement with a non-overlapped pipeline. The pipeline for Table888 is basically a non-overlapped pipeline, a couple of goals for the processor were a high clock frequency and complex instructions. The author wanted to be able to implement complex instructions easily using state machines. He has found non-pipelined designs easier to debug as well. The following chart shows the relationship ship between pipelining, clock frequency, and design complexity. It's based on the author's own experiences developing processor for FPGA's. It's a little bit of an Apple's to Orange's comparison, but it may be good for a general sense.

CPU	Max Clock Frequency	Clocks per Instruction	MIPS	Logic Cells	Processor Architecture
	100 MHz	3	33	2000	Sequential, non-overlapped
Raptor ⁶⁴	60 MHz	1.5	40	10000	Overlapped pipeline
Thor	40 MHz	0.75	53	100000	Superscalar 2 way
Qupls	40 MHz	<0.75	>53	160000	Superscalar 4-way

Note that what one chooses to do can depend on resource budgets of the whole system. If the CPU is going to stall waiting for a shared memory access most of the time, then it might as well be using multiple clock cycles to accomplish tasks. It doesn't matter how fast the CPU is if memory access is limited.

CHOOSING A BUS STANDARD

The processor interacts with the outside world using a bus. The author encourages one to use one of the commonly known bus standards. A well-known bus standard makes it possible to use peripheral cores developed by others.

As an example, Table888 uses a WISHBONE compatible bus to communicate with the outside world. Specs for the WISHBONE bus can be found at OpenCores.org. WISHBONE bus is straightforward and easy to understand and free. It is used by other projects. The external bus used by Table888 is a 32-bit bus. This is the size of the system's data bus. All the peripherals in the test system use a 32-bit data bus. The ROM's and RAM's in the system are all 32 bits wide. Also, the interface to the dynamic RAM memory is only 32 bits. Table888 makes use of two word burst memory accesses to load the instruction cache. A burst access is several accesses that occur rapidly in a row in a sequence. Since instructions are only 40 bits it works okay with a 32-bit bus. Loading or storing a word to memory requires two bus accesses.

Having mentioned the use of a standard bus, for Qupls the author decided to use his own bus standard with the goal of achieving higher performance. Qupls uses a bus called 'FTA bus' standing for Finitron Asynchronous bus.

CHOOSING AN ISA

The author would suggest as a first project to use an existing ISA and pick something simple. Designing one's own processor tends to be project N rather than project #1. It can be quite daunting to have to develop all the tools necessary to support one's own ISA, and an existing ISA is likely to have ready-made tools on the web. There are many projects that implement existing ISAs. MIPS must have been done about 100 times. An existing ISA is also likely to have examples of implementations in various languages. If you want to roll your own ISA, it's a lot of fun. There are many things that factor into the choice of an ISA. What is the processor geared towards? Is it to be designed for a specific task? What kind of resources will be available to the processor? Is there lots of memory available, or is the amount significantly limited? It is said that one of the pitfalls of ISA design is not allowing for growth in memory requirements.

READABILITY

One of the first issues to consider is readability. This is a human factor. Believe it or not, sometimes people read machine code. Having an instruction set that contains odd sized bit fields is difficult to read (at least for the author). Byte code instruction sets were partly done the way they were to facilitate reading the machine code, so that it would be easier for developers to write software. These days most software is written in high-level languages. As such, there is less emphasis on producing human readable machine code and more emphasis on performance. For this processor the author chose to forego to a byte-oriented design because he was interested in performance and planning to program in high level languages.

PLANNING FOR THE FUTURE

If one leaves no room for future instructions, it'll be difficult to upgrade the processor later. This has been a problem for several commercial processors. Table888's instruction set has a base of 256 opcodes available; most of the opcode space is unused and reserved for future expansion. Future expansion includes things

like floating point, vector operations, and SIMD operations. While working on the instruction set for the Raptor64, which is another 64-bit processor, the author found the seven-bit opcode somewhat cramped. The instruction set for that processor just fit with little room left over. If possible, leave several open opcodes for future expansion; that way it'll be possible to at least use them as prefix instructions for subsequent pages of opcodes. For an example of using page prefixes see the 6809 processor. The 65C816 processor has just a single opcode left, wisely reserved for future expansions.

Qupls uses a seven-bit primary opcode. Of the 128 possible primaries there are about 25 open codes left that could be assigned.

Part of the reason to develop a 64-bit processor isn't that it's really required right now, but that it has some room to grow over the next 20 years. The typical "small" FPGA board has megabytes of RAM available. To address that much memory, one needs an ISA that supports the address range. A question the author has heard from time to time is "How do I get my micro-controller to access more memory?". Needing to access more memory is a common problem. What might be needed is a processor with greater memory accessing capacity. One can only shoehorn so much before the shoe splits.

OPCODE / INSTRUCTION SIZE:

What works the best? For implementing the CPU in a small FPGA device the ISA must be relatively simple. Some of the first microprocessors (6800, 6502, Z80, 8085 and others) were byte code oriented. They would fetch the first byte of an instruction and begin processing from there, fetching additional opcode bytes as needed. For simplicity the ISA the author has chosen to implement has a fixed instruction size of 48 bits. He would not recommend using an oddball sized instruction set; it can be done, but one would need to put a lot of work into building a toolset that understood the ISA. The author is speaking from experience. A CPU was developed with 36-bit instructions. It was about 10x the ordinary amount of work to get the tools to deal with nibbles instead of bytes. The instruction size should at least be a multiple of eight bits. I've chosen 48 bits because a lot of bits are required to represent the number of registers available in the design. The instruction size is fixed to keep the instruction fetch simple otherwise it would be necessary to implement a table containing the size for each instruction.

VARIABLE LENGTH INSTRUCTION SETS

One of the goals of a variable length instruction set is to minimize the number of bytes required to represent a program. Shorter code can sometimes execute faster because it makes better use of the cache. For embedded systems a shorter code may allow the use of smaller less expensive ROMs. Implementing a variable length instruction set adds some hurdles to the project. Instruction cache design for instructions varying in width is a challenge as well. A sample of a processor with varying sized instructions is the RTF65003 which makes use of a table to track instruction sizes. If choosing a variable length instruction set the author would advise setting up the instruction set so that the first few bits of the opcode can be used to determine the instruction size. RISC-V processing core uses this approach. For the Thor core the size of the instruction can be determined primarily by looking at the opcode byte.

The author has found that decoding the length of an instruction for a variable sized instruction set can be on the critical timing path, at least for an FPGA based processor. He decided to use a fixed length encoding for Qupls to help improve timing.

INSTRUCTION BUNDLES

Qupls originally used 40-bit instructions. 40 bits might sound okay but a 40-bit instruction size doesn't work well with an instruction cache, because it results in an oddball cache line length. For simplicity, typically cache lines are a power of two in length, otherwise a fast division would be required to find out which cache line to load. 128 is a power of two and it's close to the size of three instructions (120 bits). So, one solution in addition to having an instruction size of 40 bits would be to pack the 40-bit instructions into a 128-bit instruction bundle. A suggested bundle format:

127	120	119	80	79	40	39	0
Debug		Slot2		Slot1		Slot0	

The extra bits in the bundle would be used for debugging information. In some processors the extra bits serve a different function. For instance, in the IA64-Itanium architecture these are template bits which control the classes of instructions in the instruction slots.

For Qupls using an instruction bundle was seriously considered but ultimately discarded. Instead, a more complex instruction cache is used which allows instructions spanning cache-lines to be fetched. An issue with using a bundled format is that one may be restricted to processing instructions in specific groups. A non-bundled format can vary the number of instructions fetched and processing more easily.

DATA SIZE

While the size of instructions in an instruction set may vary, typically data does not. I would strongly recommend against using unusual data sizes. One would be incompatible with everything else if an unusual data size is used. It becomes a nightmare to transport and convert data files. Primitive data types should be a multiple of two of the size of a byte (eight bits). That is 8, 16, 32, or 64 bits. There are a great many well-known file formats in existence. They all rely on common data sizes. If one were to choose a nine-bit byte for instance they would have trouble packing it into the eight bits that everybody else uses. Make an effort to find out what existing data formats are. If your application uses a specific type of data object, it's likely that someone else has already run into the same type of object. They may have encountered issues with using the object that one hasn't thought about yet.

REGISTERS

NUMBER OF GENERAL-PURPOSE REGISTERS

Some research reveals that typically somewhere around 24 registers is a sweet spot for performance when dealing with high-level compiled languages. Machines with fewer registers start to suffer ill effects of moving data between registers and memory. Machines with more registers don't improve very much in performance over having 20 or so registers. Having more registers impacts the task switch time because they must be swapped to memory during a task switch. Some common examples are the ARM processor which has a working set of the sixteen registers. Also, the latest processors from INTEL support sixteen registers. The original INTEL 80x88 processor sported a register set of eight registers. Later more registers were added to the design. SPARC uses a register windowing scheme where there are eight global registers and twenty-four local registers which rotate around using a circular register buffer. If starting out small, it might be advisable to leave some means to extend the architecture with more registers.

A sixteen-register machine is a good choice for performance reasons. Why aren't there twenty-four registers if it's a sweet spot? It's a trade-off between using bits in the instruction set to represent the registers and performance impacts. The choice is really between 32 and 16 registers because either four or five bits must be used in an instruction to represent the register number. For my design I've chosen to use

64 registers. The reason is to support 128-bit values. If the machine were 128-bits wide I would have used 32 registers. Another consideration is that within the FPGA memory resources are allocated in blocks. These blocks are typically 512 or 4096 bytes in size.

Yet another consideration is technical. Use of register renaming in the processor requires more registers than what appear to the programmer. About three times as many registers is a good number of rename registers. So, 96 registers are required for a 32-register machine.

REGISTER ACCESS

Are registers going to be accessed in parallel or in sequence? Some instructions require more than a single register. It may be desirable for performance reasons to be able to access more than one register at a time. To do this the register file must have multiple register read ‘ports’. On the other hand, multiple read ports increase the size and cost of a register file. If one wants to keep a smaller register file, then the registers will have to be accessed in sequence. Many instructions require only a single register read access, for example the typical add immediate or compare immediate instructions. The most frequently used memory operation, load a register, usually only needs to read a single register. With so many instructions requiring only a single register (or even no registers) accessing the register file sequentially across several clock cycles is a consideration for when multiple registers need to be read. Table888 uses three register read ports, mainly for simplicity, a few instructions read three registers (stores with indexed addressing for example); accessing registers sequentially can add complexity to the state machine and register read file path.

Qupls, as a four-way superscalar processor, has lots of register read ports. There are about 20 read ports in a smaller configuration of Qupls. This is to support executing multiple instructions at the same time. For instance, two ALUs require eight read ports, three for source operands and one for the target operand. With two ALUs, two FPUS, a flow control unit, and two data memory units a lot of ports are needed. Even more ports are required to support multiple write ports. To write multiple results at the same time requires multiplying the number of reads ports by the number of write ports.

SEGMENT REGISTERS

As part of the memory management portion of a CPU segment registers are often provided. There are usually multiple segment registers to support multiple segments which are typically part of a program. Common program segments are the code segment, the data segment, the uninitialized data segment and the stack segment. There are often other segments as well. 80x86 is famous for its segment registers, but other processors like IBM’s PowerPC also use them as well. Segment registers are an easy to understand and a low cost, low overhead memory management approach. The memory address from an instruction is added to a value from a segment register to form a final address. The segment register is often shifted left as it is added to allow a greater physical memory range than the range directly supported by the architecture. Segment registers allow programs to be written as if they had specific memory addresses available to them, such as starting at location zero, while the actual physical address of the program is much different. Once a design seems to be working well, the author tends to add segment registers to the design as a first step at providing memory management features. Table888 does not include segment registers at this point.

For Qupls the author decided not to include segment registers. Qupls uses a paged memory management unit and segment registers would be largely redundant.

OTHER REGISTERS

There are often other registers that are not general purpose in nature associated with a design. A common register is the status register, or machine control register as it is sometimes called. The status register often contains flags, and interrupt masks. It may contain other mode controlling bits like the decimal flag on the 6502 or the up/down flag on the 80x88. Many designs support additional registers such as an interrupt table base address register, a tick count register, debug registers, memory management control registers, cache control registers and others. Usually, these other registers are handled with a simple move instruction between the register and a general-purpose register. Table888 has a handful of special registers that are accessed with the ‘mtspr’ (move to special register) and ‘mfspr’. Qupls follows the convention of calling these registers CSRs for control and status registers. The CSRs are accessible with CSR register read and write instructions. The author encourages the reader to review the Qupls instruction set for these instructions, found later in the book.

MOVING REGISTER VALUES

A common operation is transferring data from one register to another. This operation is commonly done with a move instruction of some sort (MOV). Some simpler processors don’t supply a register-to-register move operation. Instead, they rely on using another instruction that doesn’t affect the data transfer, such as a register ‘or’ instruction. For example, or r1, r2, r0 effectively moves r2 to r1 because r1 is or’d with zero. It can be confusing looking at an assembly language dump, because it looks like there is an ‘or’ instruction. Another puzzle piece is that an explicit register move instruction uses only a single register read port. This is sometimes important in more advanced processors. Another related instruction that is less often used is the exchange registers instruction. Exchanging two registers can be tricky to implement because two register updates must take place. Exchanging registers is not always offered in processor architectures, when it is supported, it is often a multi-cycle operation. Qupls supports an explicit register move instruction because the move operation can move between more registers than what is possible with other instructions like OR.

REGISTER USAGE

While the general-purpose register array may be considered general in nature, and any register may be used for any purpose, registers are often given specific usages by convention for software purposes. As far as hardware is concerned it doesn’t care how general registers are used. But from a software perspective it is beneficial to assign specific registers to some tasks. For instance, often a general register is reserved for use by the operating system, meaning that application programs should not use it. This is a convention enforced by a compiler, not the hardware itself. Qupls has some basic register usage constraints.

HANDLING IMMEDIATE VALUES

First some background information. A significant proportion of instructions (for example 40%) use immediate or constant values. Immediate values or constants vary widely in the number of bits required for representation, although most constants are small. Placing small constants using a field in the instruction works not too badly. The problem to solve is how to place and use large constants in the instruction stream. There are a few goals to achieve here. 1) Minimizing processor complexity. 2) Minimizing code and data size bloat. 3) Maximizing performance. There are five basic methods of handling immediate constants that I know of besides including the constant directly in the instruction stream.

- 1) SETHI / LUI – is an instruction to set the high order bits of a register
- 2) IMMxx – is an immediate prefix for the following instruction
- 3) IMMxx – is an immediate postfix for the previous instruction

- 4) LW table – placing constants in a table
- 5) Half-operand or shifted operand instructions – instructions operating on only part of a register

Qupls architecture uses constants embedded in the cache line. This is like placing constants inline with instructions. Using inline constants is by far the best approach in the author's opinion. It minimizes the code space and reduces the dynamic instruction count which helps performance. An instruction with the constant is just one instruction executing. Other solutions add instructions and decrease the code density.

SETHI

No, this is not the search for extra-terrestrials. The author likes the moniker because it reminds him of the existence of other things. SETHI is often called LUI which stands for ‘load upper immediate’. One solution is to load an immediate value into a register using a pair of “set” instructions, then perform a register-register operation rather than a register-immediate operation. It looks like this:

ALU op used only to set the low order bits of a register ->	OR Rs2,R0,#Low ; load low order
SETHI Instruction ->	SETHI Rs2,#High ; load high order
Instruction Needing Large Immediate- translated into register operand ->	ADD Rd, Rs1, Rs2

Disadvantages of this approach:

- 1) It often requires more memory than other solutions would. Using a large immediate requires three instructions rather than the two that a prefix would require. A 64-bit processor may also require more set instructions to handle middle bits of a constant.
- 2) It uses up a register(s).

Advantages of this approach:

- 1) It's simple.
- 2) It doesn't require processor interlocks, or re-execution of the prefix when interrupts occur. Allows instructions to execute as independent units.

IMMXX - PREFIX

Second solution: use an immediate prefix instruction. The constant prefix instruction simply contains the bits of the constant that wouldn't fit in the following instruction. It looks like the following:

Immediate prefix Instruction ->	IMM16 #HighBits
Instruction Needing Large Immediate ->	ADD Rd,Rs1,#Lowbits

Advantages:

It requires less memory space as the prefix needs only to contain bits to specify an immediate. Often the prefix can be arranged to contain sufficient information so that only a single instruction is needed, rather than the two that would be required for other solutions.

Disadvantages:

It can be complicated. It may require processor interlocks or re-execution of instructions when an interrupt occurs.

IMMXX - POSTFIX

Third solution: use an immediate postfix instruction. This is one of the author's favorites. For a long time, the Qupls design used postfix immediates. The constant postfix instruction simply contains the bits of the constant that wouldn't fit in the previous instruction. It looks like the following:

Instruction Needing Large Immediate ->	ADD Rd,Rs1,#Lowbits
Immediate postfix Instruction ->	IMM16 #HighBits

Advantages:

It requires less memory space as the postfix needs only to contain bits to specify an immediate. Often the postfix can be arranged to contain sufficient information so that only a single instruction is needed, rather than the two that would be required for other solutions. The postfix also has the advantage that it can be read from the cache-line as if it were part of the current instruction. It is a little bit easier to process a postfix instead of a prefix.

Disadvantages:

It can be complicated. It may require processor interlocks or re-execution of instructions when an interrupt occurs.

LW TABLE

Fourth solution: place the large constants in a table in memory, then use regular load and store operations to load the constant into a register. This is the author's least favorite method. He is of the opinion that constants belong in the instruction stream and are better stored in the instruction cache rather than polluting the data cache. However, many systems use the load from table method.

Load Instruction – retrieves value from table ->	LW Rs2, constantAddress
Instruction Needing Large Immediate – translated into a register operand ->	ADD Rd,Rs1,Rs2

Advantage:

It's simple. It doesn't require a special means (instructions) to handle constants. Uses a means already present in the processor. This may be useful when the size and complexity of a processor is an issue.

Disadvantages:

- 1) It's often slow. Load / store operations generally occur through the data port of the processor rather than the instruction port. There may be delays for memory access.

It uses a register.

HALF OR SHIFTED OPERAND INSTRUCTIONS

Fifth solution: provide instructions that can operate on part of a register. This looks like the following:

Instruction Needing Large Immediate (operates on lower half of register) ->	ADD Rd,Rs1,#Low
---	-----------------

Instruction operating on upper half of registers ->

ADDHI Rd,Rs1,#High

Advantages:

- 1) Minimizes code size.
- 2) It often doesn't require the use of extra registers.
- 3) Does not require instruction interlocks

Disadvantages:

- 1) The number of instructions in the instruction set is increased. This may cause problems with the representation of instructions.
- 2) Increases the complexity of the processor.

THE BRANCH SET

One of the first things the author looks at when evaluating an ISA is the branch set. Is it semi-sensible or non-sense? Branches may represent up to one quarter of instruction executed. Branches are one item that must be well done in an architecture. What conditions will the processor branch on? Is it a simple branch on zero / non-zero test or are there more complex conditions available? What the branch set supports impacts what other instructions need to be available in the architecture. If branching only supports a zero / non-zero test, then other instructions must be present to setup the branch test. In the DLX architecture for instance, there are a set of 'set' instructions that set a register to a one or zero based on a condition. After a set instruction is done, then a conditional branch may occur. Many architectures include a compare instruction(s). For instance, the MMIX architecture includes both signed (CMP) and unsigned compare (CMPPU) instructions that set the value of a register to -1, 0, or 1 for less than, equal, or greater than another register. The same paradigm was used for the Raptor64 processor. For the Table888 processor there is a standard set of branches that act like they are branching on a flag register value. If you're used to the 6800 / 68x00 / 6502 series processor, these branches will look familiar.

Qupls uses combined compare-and-branch instructions to help reduce the dynamic instruction count. Effectively a compare operation and a branch operation are fused together.

BRANCH TARGETS

Branches which change program flow conditionally are usually implemented as relative branches. One reason to implement using relative addresses is that it takes fewer bits to represent the target address of the branch. In many designs, typically 16 bits are allowed for, for a branch displacement even though only 12 bits are what is necessary. It has to do with keeping the format of instructions simple and there is usually room in a branch instruction for sixteen bits. Even in byte-code architectures that use eight-bit branch displacements by default, there is often a longer form for branches supported (for example the 6809). A lot of software expects at least a sixteen-bit branch displacement. Eighteen effective bits is recommended. Qupls has effectively 21 bits of displacement. A second reason to use relative branching is that it allows code to be relocated in memory. Changing the location of the code in memory often does not require updating relative addresses associated with branch instructions. Note that if some form of memory management is present, it is possible to move a program in memory without having to worry about fixing up non-relative addresses, so the value of relative branches for this reason is limited.

A relative branch branches relative to the address of the branch instruction or the address of the next instruction (do not make it otherwise). The author strongly recommends using the address of the next

instruction as the reference point for branches. It just makes it a bit more readable in machine code. A branch with a zero displacement arrives at the next instruction. As a ground rule, the displacement field should be at least 12 bits.

As mentioned previously, Qupls has effectively a 21-bit displacement. The displacement constant is encoded as an 20-bit value, but it is in terms of wydes opposed to the number of bytes. The number of bits may seem like overkill, but it's trying to look into the future of branches. When people write structured subroutines, they typically don't create a routine more than a few pages long. This results in branching that branches within a few kilobytes of the branch location because branches are located within a subroutine. Hence the reason 12 bits is adequate most of the time. However, if one is using an automated code generator, the code generator may generate larger subroutines.

Unconditional branches in Qupls use a wyde displacement value rather than an instruction displacement because subroutines could be located at any wyde address. They also have larger displacements which is needed to hit subroutine targets. Unconditional branches are often used to enter or exit routines.

BRANCH PREDICTION

Branch prediction enhances performance by predicting which direction a conditional branch instruction will take. It is often used in overlapped or superscalar pipeline designs. Branch prediction can turn branches into a single cycle operation rather than a multi-cycle one which is what happens when a branch is taken in an overlapped pipeline design. Branch prediction has little value for the Table888 processor as it's a non-overlapped pipeline. It takes multiple cycles to execute a branch whether prediction is present. Branch prediction adds additional complexity to the processor. The Raptor64 includes a (2,2) correlating branch predictor, for an example of a branch predictor.

Qupls includes two branch predictors, one called a branch-target-buffer operating at the fetch stage of the processor, and a second predictor called a gselect predictor operating at the decode stage of the processor.

LOOPING CONSTRUCTS

Sometimes processors support looping constructs directly. 680x0 has a decrement and branch instruction. 80x88 has loop instructions which decrement the CX register and branch. Decrementing a register then branching if it is non-zero is a common operation, so some processors implement these two operations together with a single instruction. It's really like executing two instructions at once. Table888 supports a decrement and branch instruction for loop constructs.

OTHER CONTROL FLOW INSTRUCTIONS

SUBROUTINE CALLS

Subroutine calls represent about 1% of instructions executed, but it's an important 1%. Some architectures store the return address for a subroutine call in a processor register, typically a general-purpose register. These architectures may make use of a jump-and-link (JAL) instruction to both call a subroutine and return from it (for example xr16 – Grey Research). PowerPC architecture makes use of a dedicated link register (LR). This works only for a single level of subroutine call, and the register must be saved onto the stack before calling a nested subroutine. Table888 automatically stores the return address on the stack for a subroutine call. Using a JAL instruction to return from a subroutine allows a return to a point past the original calling address. This is occasionally useful to skip over inline parameters passed to a subroutine. What's more useful is removing parameters from the stack during a return operation. This is useful enough

that a number of architectures incorporate it as part of a return instruction (680x0, 80x88). While Table888 doesn't directly support returning past the calling point, it does support adding onto the stack pointer to remove parameters.

Qupls supports branch-to-subroutine, [BSR](#), with a large displacement which should be sufficient for most software. It also supports jumping to a subroutine at an absolute address, [JSR](#). To use the full address range the target address of the subroutine must be loaded into a register before using JSR. Otherwise, the address range is limited.

MEMORY INDIRECT JUMPS

A common operation that must be performed by a running program is to jump to address from a table of addresses. This can be done using a load instruction then a jump to register instruction, but it often has hardware supporting the operation, called a memory indirect jump. Qupls has a memory indirect jump instruction which may set part of the instruction pointer, so that the jump table size may be limited to smaller entries.

RETURNING FROM SUBROUTINES

Returning from a subroutine is the reverse operation to calling one. In a machine that uses registers this can be as simple as loading the PC with the register value. Some RISC architectures store the return address in a register. Table888 like many architectures, loads the return address off the stack. Qupls return-and-deallocate instruction, [RTD](#), returns from a subroutine, deallocates the stack, and allows a return a few instructions past the calling point.

SYSTEM CALLS

System calls are used to call the system. They are often called software interrupts or traps. The 80x88 uses the name 'int'. 6809 calls this a 'SWI' for software interrupt. In 6502 parlance it's the BRK instruction. They are called TRAPs on the 680x0 series. All these instructions do much the same thing. They are almost like a jump to subroutine instruction with an implied address. The system call instruction usually saves more machine state on the stack than a subroutine call would. These instructions may also switch the processor operating mode into a more protected level. Table888 calls this a break (BRK) instruction. Qupls uses the [CHK](#) instruction that always fails to perform system calls. CHK accepts a cause code argument encoded in the instruction which determines which exception vector will be invoked.

Qupls uses an internal state stack to store CPU state during a system call. This is much faster than using external memory. Multiple items such as the instruction pointer, and status register are stored on the state stack in a single clock cycle. This stack is small, only eight entries deep.

RETURNING FROM INTERRUPT ROUTINES

Like a subroutine, interrupt routines also require a method of return. Typically returning from an interrupt routine requires loading some of the machine state from the stack in addition to the return address.

Hardware interrupts are not normally invoked with parameters, so there are no parameters to pop off the stack at the end of an interrupt routine. Qupls uses the [RTE](#) instruction to return from an interrupt or system call. This instruction loads both the instruction pointer and status register from the internal stack. A feature of the Qupls RTE instruction is the ability to perform a two-up return, returning twice from a system call which would otherwise be difficult to do since an internal state stack is used.

JUMPS

Strange as it may seem, unconditional (absolute address) jumps are very rarely used. Usually, one wants the program to branch conditionally or call a subroutine. An unconditional relative branch is usually used for jumping within a program. Jumps are sometimes used to handle exceptional conditions, where the normal subroutine return is circumvented. For instance, a jump may be used to implement a program abort. Another place where jumps are used sometimes is with jump tables. Addresses of subroutines are stored in a table in memory. Functions in the table are called by loading a register with an index number, loading the address from the table using the index into the table and jumping to it. This operation can be done with registers and a jump-to-register value instruction. Table888 implements this complex operation directly as an indexed memory indirect jump.

CONDITIONAL MOVES

Conditional moves are available in many modern architectures. The idea behind conditional moves is to avoid branches which are usually timely to execute. So, a conditional move is a performance enhancing instruction. A conditional move ‘conditionally’ moves a value into a register based upon whether the condition is true. It’s like having a branch instruction combined with a load instruction. Table888 does not currently have any conditional move instructions. Qupls has conditional moves in the form of the [CMOVZ](#) and [CMOVNZ](#) instructions. It also supports zero-or-set, or just plain set instructions which are also a form of conditional move.

PREDICATED INSTRUCTION EXECUTION

Some processors include the ability to execute virtually any instruction conditionally, for example the ARM processor or INTEL Itanium IA64. It’s a powerful means of removing branches from the instruction stream. Sequences of instructions executed with predicates rather than branching around the instructions should be kept short. The issue is the amount of time spent fetching the instructions and treating them as NOPs versus the time it would take to branch around the instructions. A compiler can optimize this and choose the best means. One of the problems of predicates is that they use up bits in the instruction regardless of whether they’re useful. For instance, the Itanium has a six-bit field in virtually every instruction. The result is that a wider instruction format of 41 bits is used. A second problem with predicates is that they act like a second instruction being executed at the same time as the instruction they are associated with. The predicate operation requires a predicate register read, and a predicate evaluation operation. This adds complexity to the processor. Predicate registers are another form of register that must be present and bypassed in an overlapped or superscalar design.

The Thor processing core features uses a whole byte for predicates but gains back some of the opcode space by using redundant forms of the predicates as single byte instructions.

Qupls uses the [PRED](#) instruction modifier to perform predicated operations. The modifier may be applied before a group of instructions to be predicated. The author got the idea for the PRED modifier by browsing the comp.arch newsgroup and learning about its use in the My 66000 CPU. Use of a modifier partly solves the issue of wasting instruction bits specifying predicate registers, and the issue of accessing another register for predicate operation. In Qupls any general-purpose register may be used as a predicate. Predicates may also be applied for vector instruction.

COMPARISON RESULTS

Another issue to resolve is whether to use a flag register(s) or a result stored in a general-purpose register to determine when to branch conditionally. Avoiding the use of a flags register makes it easier to implement an overlapped pipelined or superscalar design. However, most processors in large scale use, use explicit flags registers (80x88, SPARC, ARM, PowerPC uses eight flag registers). It is somewhat simpler architecturally just to use a general-purpose register and branch based on the value in the register. The most common form of branching is branching on whether a register is zero, so a simpler architecture just uses the register directly (for example the DLX). The architecture presented here stores the flag result from a compare operation in a general-purpose register. That register can then be tested using a branch instruction. Part of the benefit of having so many general-purpose registers in the design is that they can act as a substitute for other forms of registers, in this case a flags register. Several of the general-purpose registers in Table888 are designated as ‘flags registers’ by convention. For Qupls any register may be used to store flag results.

DUAL OPERATION INSTRUCTIONS

Dual operation instructions are not commonly done, not many compilers support them. Qupls uses them because there are instruction bits available to represent them and they are occasionally useful to reduce register usage and instruction counts. A dual operation instruction performs two operations on data instead of one. The first operation is performed between two source registers followed by a second operation on the result of the first and an additional source register. An example of a dual-operation instruction is the [AND OR](#) instruction.

ARITHMETIC OPERATIONS

In the simplest RISC machines one can get away with just an ADD instruction. It’s possible to synthesize other operations like multiply from an ADD instruction. So, instructions beyond the ADD instruction are provided for performance enhancement and programmer convenience. In some instruction sets multiply and divide operations are not supported as they consume hardware resources. Multiply and divide require multiple clock cycles to complete and have several states of their own.

Arithmetic operations include addition, subtraction, multiplication and division. These are available in Qupls with the ADD, SUB, MUL, and DIV instructions.

There are both signed and unsigned versions of the arithmetic operations.

LOGICAL OPERATIONS

In the simplest of RISC machines one can get away with just a single inverting logical operation like NAND, or NOR. Other logical operations can be synthesized from the aforementioned ones. Once again additional instructions are supported for performance and programmer convenience.

Qupls logic operations include logical ‘and’, logic ‘or’ and logical exclusive ‘or’ and others. The mnemonics are as follows: AND, OR, EOR, ANDN, NAND, NOR, ENOR, and ORN. Note there are no immediate forms for the following: NAND, NOR, ENOR, and ORN. The instructions formats for logical operations are the same as those for arithmetic ones.

SHIFT INSTRUCTIONS

Shift instructions can take the place of some multiplication and division instructions. Some architectures provide shifts that shift only by a single bit. Others use counted shifts, the original 80x88 used multiple clock cycles to shift by an amount stored in the CX register. Table888 uses a barrel shifter to allow shifting

by an arbitrary amount in a single clock cycle. Shifts are infrequently used, and a barrel (or funnel) shifter is relatively expensive in terms of hardware resources.

In Table888 the shift immediate instructions are implemented as a subset of the RR (register to register) instruction group because the immediate value only needs to be six bits. This small value fits nicely into what is normally the register field for the instruction. It would be wasteful to implement these immediate mode instructions in the major opcode grouping.

Qupls shift instructions have their own instruction group, more opcode bits are used as the shift instructions may shift pairs of registers. This is done in some architectures as it allows the implementation of rotate operations. The Qupls arithmetic shift right instruction, [ASR](#), features rounding options on the result.

MYSTERY OPERATIONS

There is a class of instructions available on some processors that the author likes to call ‘Mystery Operations’. For a mystery operation the operation to be performed isn’t known until runtime, and hence is a mystery. This class of instructions is present to aid in the avoidance of writing self-modifying code. In some cases, it’s desirable to control the code itself without resorting to complex (and slow) branching. For instance, in a graphics plot routine, it may be desirable to control the raster operation (AND, OR, XOR, COPY, etc.). Rather than use a case statement with many branches, instead the raster operation code is loaded into a register. The program then executes the code from the register rather than branching. Code executed with mystery op’s can run substantially faster than code using branches.

However, mystery operations are not typically available in modern CPUs they can wreck-havoc on a pipeline depending on how they are implemented. Instead, other means like just-in-time, JIT, compilation are used.

OTHER INSTRUCTIONS

Branching to registers. Some higher performance designs include the capacity to conditionally branch to a location contained in a register. Supporting this functionality significantly increases the number of branch instructions. The benefit to being able to branch to a register is that the register value doesn't have to be calculated like a branch displacement does. Therefore, the target address of the branch can be known sooner.

Bit-field instructions. Bit-field instructions are nice-to-have but one can get by without them. Compilers can easily synthesize extract and insert of bit-fields using shift and ‘and’ or ‘or’ masking operations, at some performance cost. Many high-level languages do not support bit-fields. The ‘C’ language does support bitfields. The arpl compiler directly supports bitfield operations on primitive data types.

Bitmap instructions. Bitmap instructions used to manipulate bitmaps are nice-to-have but once again they are instructions that can be synthesized by a compiler at some performance cost.

SIMD instructions. SIMD instructions are straightforward to implement, however they take up a lot of room because of the parallel hardware. They also may require additional registers to implement. SIMD instructions are often done with wide registers (for example 128 bits or more). SIMD instructions can considerably enhance performance for some applications because they operate on multiple data items at the same time using a single instruction. The Qupls ISA supports SIMD instructions. Most instructions have a precision field that indicates how to treat values in registers.

String instructions. String type operations include block moving, block set, and block compare operations. The 80x88 has some string operations. Once again these operations can be performed using existing instructions at some performance cost. String operations can considerably enhance performance for some applications.

EXCEPTION HANDLING

Software exceptions are just a special form of branching. When an exception occurs during an instruction, there is an automatic call to an exception handler which is located at an implied address. Almost the same thing can be done without software exceptions by using existing instructions to test for exceptional conditions, then branching if an exceptional condition is found. The reason to do things automatically is to improve performance and reduce code size. When exception handling is present, there's no need to explicitly test for exceptional conditions in program code, the processor does it internally. There are fewer instructions fetched and executed and hence code runs faster.

HARDWARE INTERRUPTS

Hardware interrupts are in some ways like software exceptions and many processors use the same hardware resources to implement both. The difference between a software exception and a hardware interrupt is that a software exception occurs as the result of executing an instruction and a hardware interrupt may occur at any time being triggered by an external event. Software exceptions are usually *synchronous*, occurring when a specific instruction is executed. Hardware interrupts are *asynchronous* events. Hardware interrupts are such a powerful mechanism and so useful that virtually all processors have support of some kind for them. A hardware interrupt allows the processor to respond to external events. The external event directly triggers a jump to hardware interrupt handling routine, rather than having the processor poll for the external event. The hardware interrupt ‘interrupts’ whatever the processor happens to be doing. Table888 supports hardware interrupts and uses the break (BRK) instruction in the implementation of hardware interrupts. Qupls also supports hardware interrupts and software interrupts with the CHK instruction. Having made a

big boo about hardware interrupts it should be noted that it's possible to get by without them. The original Apple machine didn't make use of interrupts. Even something as sophisticated as the Apple Macintosh used a system of co-operative multi-tasking rather than interrupt driven tasking. It's entirely possible to setup a decent polling system, many embedded systems work this way.

INTERRUPT VECTORING

A design question to answer is "how does the processor know where to go when an interrupt occurs "? About the simplest mechanism to use is to have the processor vector to code at fixed addresses when an interrupt occurs. This mechanism is used by many RISC processors. This is like "when hardware interrupt #1 occurs, go to address \$100, when hardware interrupt #2 occurs go to address \$200, and so on. The original Table888 used a variation of this method, where the upper address bits were determined by another register in the processor. The z80 uses a similar mechanism.

A slightly more sophisticated method of determining the vector address is to use an interrupt vector table. The vector table contains a list of addresses of where to vector to for a given interrupt. This mechanism is used on a lot of processors including but not limited to the x86 series, the 68x00 series, SPARC and even many 8-bit machines like the 6502, 6800, and 6809.

INTERRUPT VECTOR TABLE

The interrupt vector table is a table full of addresses of the interrupt routines. The vector table may be located at a fixed memory address meaning, usually that's where the system ROM would be placed. Many eight-bit machines have a fixed address for this table. In a slightly more advanced, and more expensive system, the location of the interrupt vector table is relocatable in memory via an interrupt base address register. This is one piece in providing the capability of writing hyper-visors for the machine. Table888 calls this register the VBR (vector base register).

In Table888 the interrupt vector number supplied by the BRK instruction indexes into the interrupt vector table to determine which vector to load. The BRK instruction acts like a memory indirect jump then. There are 512 interrupt vectors allowed for by Table888.

Qupls uses a separate vector table for each operating mode of the processor. The location of vector tables is stored in one of the TVEC CSRs. The exception vector table and [exceptions](#) in general for Qupls are outline in the Qupls specific section of the document.

GETTING AND PUTTING DATA

To have data to work on some means must be present to transfer it to or from memory or an I/O device. Are there going to be explicit I/O instructions or is I/O memory mapped (called MMIO)? There is some appeal to having explicit I/O instructions. I/O typically does not require the same range of addressing that general memory does. I/O devices may be limited to a 64k page of memory as on for example the 80x88. In the test system the author built, all the I/O is within a single megabyte address range even though there are gigabytes available. This would allow the use of shorter instructions to access the I/O. Another appealing aspect of explicit I/O instructions is that it makes it easy to indicate when data caching should not be used. One way to think of I/O instructions is as if they were un-cached memory load / store instructions. Some designs have explicit un-cached memory load / store operations, this is almost another way of saying I/O.

Transferring data to / from memory is what the load and store instructions are for.

Data doesn't all come in the same size. Data size for different structures varies widely. Examples of large data structures are video frame buffers or a movie clip. A smaller structure may be a name such as a person's name or place. About the best we can do here is load or store a portion of a data structure at a time. The processor handles the most primitive data types directly, these include bytes (8 bit), characters (16 bit), half-words (32 bit) and words (64 bit). Note that as a convention the author calls a 16-bit quantity a wyde. To him, a word is the word size of the machine, a half-word is half that size, and a byte is always eight bits. These quantities are called a byte (8 bits) a wyde (16 bits), a tetra(32 bits) and an octet (64 bits) by Knuth. The RISC paradigm is that the only instructions accessing memory are load or store instructions. This design doesn't quite follow the paradigm. It also supports explicit stack push and stack pop operations in addition to load and store instructions. Pushing values onto the stack is a common way argument passing is implemented in high-level languages. RISC machines synthesize this quite nicely using load and store instructions. The author finds push / pop instructions easier to read and understand while reading code. Is that store for a subroutine push? or a general memory op? Explicit push and pop instructions may also have better code density if they can support pushing and popping multiple registers with a single instruction.

ALIGNED AND UNALIGNED MEMORY ACCESS

Memory access alignment is an issue that crops up on a machine supporting multiple data sizes larger than a byte. On a machine that's byte addressable with varying data sizes, one must decide how to support unaligned memory accesses. If the data-bus size of the machine is eight bytes wide, a word access could potentially start at any one of those bytes. If the access starts at any byte other than zero, then it would wrap around into the next memory word. This is called an unaligned access. Directly supporting unaligned memory accesses requires multiple bus accesses for unaligned data. This isn't too bad to do in a state machine driven design, but it's difficult to do in an overlapped pipelined design. Many machines simply stipulate that unaligned memory access is not allowed. Other machines implement unaligned accesses using traps where software can implement the unaligned access (this makes unaligned memory access quite slow). Table888 does not currently support unaligned memory accesses. Because the bus size is only 32 bits in the current implementation there is a potential to easily allow words to be half-word aligned in memory. Qupls ISA supports unaligned memory access.

LOAD / STORE MULTIPLE

With a machine with a lot of registers there is often a means to load or store more than a single register at a time. For instance, the PowerPC has a LMW instruction standing for 'load multiple words'. Table888 supports loading and storing multiple registers at the same time with the LMR (load multiple registers) and SMR (store multiple registers) instructions. The range of registers between Rs1 and Rs2 is loaded or stored to an address identified by Rs3. Part of the value of the LMR and SMR instruction is that they can save considerable cache space over having many independent load / store instructions. For example, one might use 16 SMR instructions rather than 256 SW instructions to save the register state during a task switch. Qupls supports loading and storing multiple registers with the PUSH, POP, PUSHA, POPA, LDCTX and STCTX instructions.

THE STACK

This architecture has an explicitly defined stack. Oftentimes with RISC machines there is no explicit stack pointer. Instead, one chooses a general register to use and uses regular load and store instructions. It's a little bit less intuitive a way of doing things.

In this architecture register R31 is used as the stack pointer. The stack may be used to pass arguments to functions. There are instructions supporting stack operations which include [PUSH](#) and [POP](#).

Note that while some machines allow pushing or popping the entire register set with a single instruction, that is deemed to be not a good idea for a machine with 256 registers like Table888. It would create too much latency when other processing like interrupts is going on. The only other option is to be able to push or pop a subset of registers, which is allowed. In Table888 the push / pop instructions push or pop any four of 256 registers. Qupls allows up to five registers to be pushed or popped. Note that the same register may be pushed or popped multiple times with the instruction.

A push tends to be used more often than pops. Instead of popping arguments off the stack after a subroutine call, usually the stack pointer is simply incremented because we don't care about getting the argument values back. Adding onto the stack pointer turns the pop operation into a single instruction, and often a single cycle operation, rather than a series of memory operations.

One consideration for the POP instruction is whether to support popping multiple items with one instruction. It adds a level of complexity to the processor to pop more than one item per instruction because most other instructions only update one register. Register write ports are expensive so often there's only a single write port to the register array. That means doing multiple updates to the register array requires multiple clock cycles. In an overlapped pipelined design, it's desirable to stick to a single register update per instruction. Providing for multiple register updates makes the design really complex. If the author were going to turn Table888 into an overlapped pipelined design one thing he would seriously consider is limiting the pop instruction to a single register.

Qupls handles popping multiple values off the stack by implementing pop as a macro instruction. The pop instruction invokes micro-code which uses a separate load instruction for each value popped from the stack.

DATA CACHING

Qupls has a 64kB data cache. While a data cache can improve performance it adds complexity and can be tricky to debug. Store operations which typically write to memory are effectively un-cached anyway. Also, I/O operations should not be cached. For some critical applications data isn't even allowed to be cached.

There are several policies associated with caches. A given cache may implement multiple policies as options. I mention the most prominent ones here.

A write-back cache policy delays the writing of data back to main memory until the dirty cache line is dumped. This may be when the data cache controller decides it's a good time to update memory, also when a new incoming cache data would replace the dirty line. The cache is updated immediately on a data store operation, but main memory is not.

A write-through cache policy updates the data in both the cache (if it is in the cache) and main memory immediately.

A write-allocate cache policy is a cache that loads the cache line from memory when a write cycle takes place.

ADDRESS MODES

A point of sale from a marketing perspective in the past has been the number and type of address modes available in the processor. "Use any address mode with any instruction." was a statement about the simplicity of the processor when coding in assembly language. Symmetry of address modes for instructions was a selling point. These days load / store architectures are popular and in these architectures address modes really apply to only the load and store instructions. The author follows this paradigm. While it is

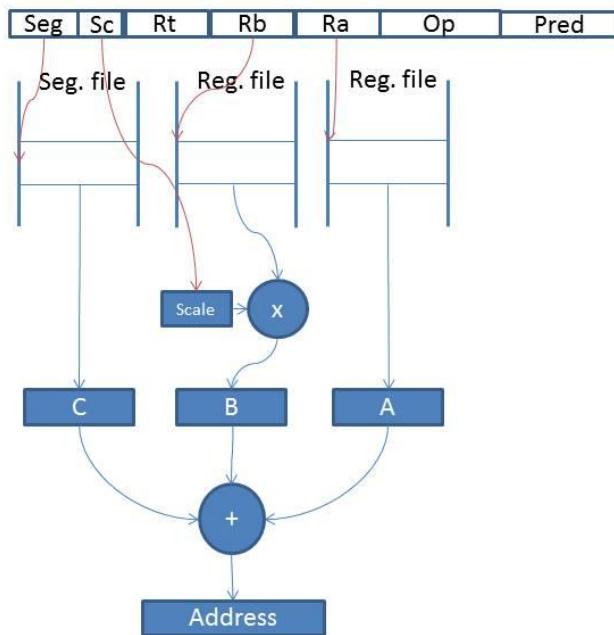
possible to have quite a general set of address modes including things like memory indirect addressing and automatic incrementing or decrementing of registers (see the 680x0 architecture for example), complex address mode can be synthesized from simpler ones and the synthesized address modes execute just as fast as built in ones. Complex addressing modes were just an attempt at programmer convenience while programming in assembly language. Unless the language compiler is really sophisticated it's unlikely to even be able to use some of the more complex address modes. Many RISC designs include only a single addressing mode – register indirect with displacement or sometimes only register indirect. They then rely on a compiler to synthesize other address modes if required. Table888 implements two address modes for load and store instructions. The modes are register indirect with displacement, and indexed addressing with a scaled index register. The author happens to like the scaled indexed address mode. It's sometimes convenient to use the scaling. Qupls uses scaled indexed addressing exclusively.

SCALED INDEX ADDRESSING

Indexed addressing with a scaled index register works by adding two registers together with an offset to form the address of the data. The second index register may be optionally multiplied by 2, 4, or 8, this is called scaling. The idea behind scaling is that data may be accessed by an ordinal number, incrementing a register by one unit at a time to access the next data item. The scale factor accounts for the size of the data which may be one, two, four, or eight bytes in size. Without scaling it is necessary to use another register and perform a multiplication or shift operation prior to the load / store. A compiler will output the necessary multiply and add instructions to do indexing. It's a bit of a trick to get the compiler to use scaled indexing which only applies when the object size is 2,4, or 8 bytes in size. Note that scaled indexed addressing mode uses an offset and not a displacement. The difference between an offset and a displacement is that an offset is always positive, and a displacement may be either positive or negative. The offset is limited to eleven bits. If a larger offset or displacement is required it will have to be managed using registers.

The following diagram shows how scaled indexed addresses are formed. The diagram is pertinent to the Thor processing core and so shows a predicate field in the instruction, but illustrates the address formation.

Scaled Indexed Addressing



REGISTER INDIRECT WITH DISPLACEMENT ADDRESSING

The other addressing mode that is highly useful is register indirect with displacement. In this address mode a register is added to a displacement to form the data address. Several other address modes may be emulated using this one. Setting the register to zero results in a displacement only mode, and setting the displacement to zero results in a register indirect mode.

SUPPORT FOR SEMAPHORES

While semaphores can be implemented using software only, it is an extremely expensive operation and slow to perform only with software. Ideally there is some support for semaphore operations supported by the processor itself. Instructions that support semaphores include instructions that atomically read-modify-write memory. A compare and swap instruction has been implemented on many processors to support semaphore operations. Other instructions include test-and-set bit, or increment, decrement or rotate memory.

An alternative to atomic memory instructions are instructions that perform a load and then a conditional store. These are called a locked or linked load and store. The load operation sets a flag in the processor that a semaphore access is desired. A following store operation checks this flag and aborts the store if the flag isn't set. The flag may be reset when another processor accesses the memory region identified by the load.

Table888 did not support a compare-and-swap or other atomic memory operations. Instead, semaphores will have to be implemented with software or external hardware. The test system has a set of 1024 hardware semaphore registers available to use which can be accessed like a memory device.

Qupls supports compare-and-swap and other atomic memory operations.

For Qupls, atomic memory operations are performed at the memory controller. The memory controller accepts an atomic memory opcode from the CPU and performs it.

MEMORY MANAGEMENT

SEGMENTATION AND PAGING

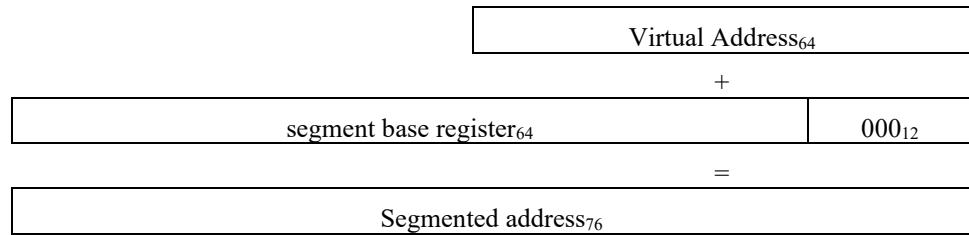
Segmentation and paging are the two main choices for memory management beyond some simpler mechanisms like bank switching. The goal for Table888 was to implement both. Several commercial processors implement both segmentation and paging. Although segmentation has fallen out of favor somewhat it is still used. Typically, the segmentation part of the CPU has a handful of segment registers loaded with a flat memory model, then is for the most part ignored. One place where segmentation is still used in a modern OS is in establishing the global storage area, and the thread local storage area.

SEGMENTATION OVERVIEW

As part of the memory management portion of a CPU segment registers are often provided. There are usually multiple segment registers to support multiple segments which are typically part of a program. Common program segments are the code segment, the data segment, the uninitialized data segment and the stack segment. There are often other segments as well. 80x86 is famous for its segment registers, but other processors like IBM's PowerPC or the PA-RISC machine also use them as well. Segment registers are an easy to understand, and a low-cost memory management approach. The memory address from an instruction is combined (added) to a value from a segment register to form a final address. The segment register is often shifted left as it is added to allow a greater physical memory range than the range directly supported by the architecture. Segment registers allow programs to be written as if they had specific memory addresses available to them, such as starting at location zero, while the actual physical address of the program is much different. Once a design seems to be working well, the author tends to add segment registers to the design as a first step at providing memory management features. Table888mmu uses sixteen segment registers. Table888mmu's segmentation system is modelled after the INTEL 80286 segmentation model. The basic concepts are the same, but the layout and size of fields has been altered. Qupls does not use segmentation.

ADDRESS FORMATION

The virtual address is added to a segment base register to form a final address.



The Table888mmu sample shifts the segment base register left by 12 bits before adding it to the virtual address. The resulting segmented address could be 76 bit in size, however fewer bits are implemented in the sample.

The number of bits shifted to the left is referred to as the paragraph size.

NUMBER OF REGISTERS

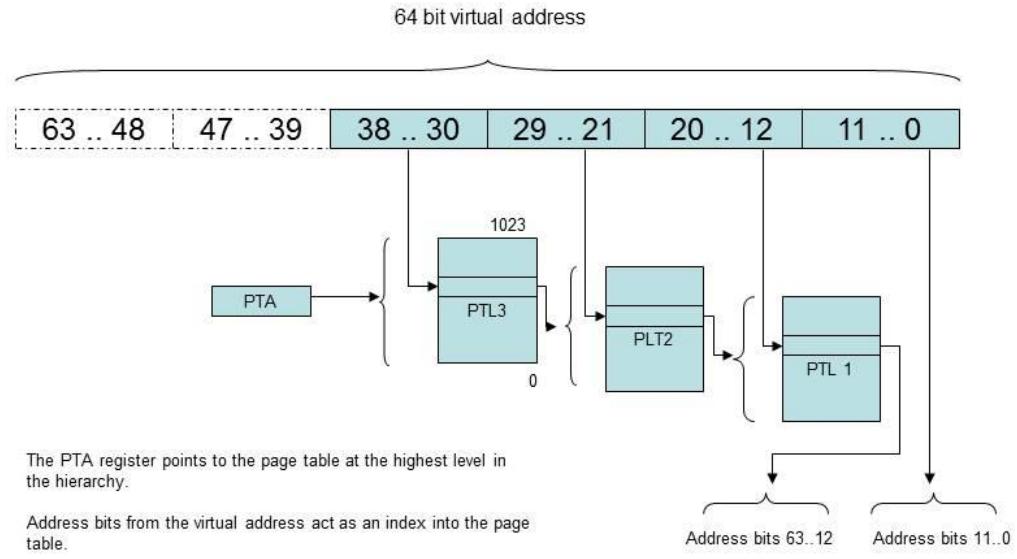
The number of segment registers that are useful seems not to be quantified as closely as the number of general-purpose registers. However, four registers were deemed not enough for the 80x86 architecture and two more segment registers were added. Also, a couple of additional registers in the 80x86 design were added to support the segmentation architecture and they act a lot like segment registers. These include the task register and the local descriptor table register. So, we have about eight segment registers in the 80x86 architecture. PA-RISC uses eight “space” registers that act a bit like segment base registers. PowerPC uses an array of sixteen registers. Table888 uses sixteen segment registers. Segments registers are typically initialized to a flat memory model then forgotten about.

The segment registers in Table888mmu contain a selector which is 24 bits in size. This size was chosen as they are used as an index into a segment descriptor table. The segment descriptor table in this case contains a maximum of 512k entries.

PAGING OVERVIEW

Paging uses a set of tables to perform mapping of virtual addresses to physical ones. Unlike segmentation, paging cannot resolve maps right down to individual bytes. Instead, memory is broken up into pages and managed on that basis. A typical page size is 4kB. The virtual address is divided up and each part of the virtual address is used to index into a table.

Virtual to Physical Translation



* Translations apply only in application mode.

The table at the highest level of the hierarchy is usually permanently resident in the computer's memory for performance reasons. Because there is a fair amount of work to be done in mapping addresses, address mappings are usually cached in an additional unit called a translation look-aside buffer (or TLB). This unit is also sometimes called an Address Translation Cache (ATC). A paging system tends to have more overhead associated with it compared to a segmented system.

One of the nice features of paging is that it is almost invisible from a software perspective. There aren't any registers like segment registers, to worry about when paging is active. Paging simply works behind the scenes.

The QN9000 paging system can map the entire 64-bit address space. A multi-level system of page directories and subdirectories is used. In most cases the address space mapped will be less than a full 64-bit address space. The paging system accommodates this by using a smaller directory hierarchy. For instance, if an application is less than 512B in size, a single level page system is used. If the application can fit within 4TB only two levels are required. The depth of the directory system is controllable on an application basis. The page memory management unit takes care of walking the page tables in hardware to find a translation. Translations are stored in a translation look-aside buffer (TLB) which is a translation cache, so that the page tables don't have to be walked for every translation.

REGISTERS

The primary register that controls paging is the page table address register (PTA), page table base register, PTBR or as it is alternately called control register number three. (CR3). This register contains the base

address of the root page table in memory. Once the PTBR is set, the processor knows where to begin looking up virtual to physical address translations. The PTBR is a register that needs to be saved and restored when the context changes.

PAGE TABLES

Page tables are the central piece of a paging system. There are two types of page tables, page directory tables and page leaf tables. For Qupls page directory tables are 64kB in size and contain 8192, 8-byte entries. Page Leaf tables are also 64kB in size and contain 8192, 8-byte entries. Details of the Qupls [paged memory management](#) system are available later in the book. The paged mmu knows where in the hierarchy the page table is, so it can determine if the table is a directory table or a leaf table. Page tables at the bottom of the hierarchy are always leaf tables.

NOP RAMPS

A brief word about NOP ramps used with the Tabl888 architecture. It shows that sometimes software can aid resolving hardware issues.

In a paged memory management unit, address translations take place continuously, not just at segment load time as would be for a segmentation unit. One nice feature about segmentation is that one can be sure that if the segment is loaded it is a continuously available memory range.

With paging on the other hand, page faults may occur at 4kB boundaries. When the page isn't present in memory, it must be loaded then the instruction can be executed. In Table888 most instructions are a single instruction word in length, so they won't cross a page boundary. However, there are several prefix instructions, when combined with a prefix an instruction might cross a page boundary. This is bad news. The problem is that the prefixing would get lost in the shuffle to move the missing page into memory. There are two solutions, one is to go back a page in memory and re-execute the prefix after the missing page is loaded. This is complicated by the fact that both pages are required to be present in memory, otherwise the processor would thrash back and forth trying to execute the instruction. In Table888 it is safe to re-execute the prefix instructions as they don't modify the processor's state until the following instruction is executed. That means going back a page and re-executing the prefixed instruction is simple to do. The second solution is to force the instruction stream to output the prefixes so that they don't cross page boundaries. This can be handled by the assembler. The assembler handles the occasional case where a prefix instruction would cause an instruction to span a page boundary by outputting a series of NOPs to force the instruction onto the next memory page. The following example shows that the prefix instruction to a store byte operation is forced onto the next page of memory.

00008FF0	41 F8 2A 90 00	bne	f10, kbdi2
00008FF5	16 01 24 00 00	ldi	r1, #36
00008FFA	EA EA EA EA EA	; imm	
00009000	EA EA EA EA EA		
00009005	FD 70 FF 03 10		
0000900A	A0 00 01 00 18	sb	r1, LEDS

Note that the NOP ramp's won't work if address space defined by a segment is paged out of memory and the segment isn't aligned on a 4kB boundary. This shouldn't be the case in Table888 as the segment paragraph size is the same as the paged mmu page size.

PROTECTION MECHANISMS

Table888's protection mechanism is like that of the 80x86 series. If you are comfortable with the 80x86 protection mechanism then Table888's mechanism will seem familiar. The format of descriptors is different, but the fields are similar. Table888 has sixteen protection ring levels.

Qupls reduces the number of rings to four, called operating modes.

Often there are hardware supported protection mechanisms for a given architecture. Many architectures are bi-modal, with kernel / system / supervisor modes and user/application modes. Even processors supporting more modes are often used in a bi-modal fashion by the OS. The x86 series processor has four levels of privilege. Usually, all the features of the processor are available in kernel mode, while a subset of features are available in application mode. Limiting application code to a subset of the processor features is one way of protecting the system. Qupls supports 256 privilege levels in addition to having four operating modes.

The segment limit in a segmented system protects against memory access outside of the segment. If an access is attempted beyond the limit a bounds violation exception occurs. If the segment descriptors are only accessible from kernel mode, then application code can't modify them to gain access to data outside of the segment.

PROTECTION RULES

Code cannot access data at more protected level. This is checked by comparing the processor's current privilege level to the privilege level of the segment that is pending access. The check is performed when a segment register is loaded using the mtspr instruction. In paged MMU system the check is performed when the page is accessed.

Code at a high protection level cannot call code at a lower level. In Table888's case this is checked when a jump to subroutine or jump instruction is executed with a jump selector prefix. For Qupls, a CHK instruction is used to switch protection levels.

The code segment may only be loaded with code type descriptors. If an attempt is made to load a data descriptor into a code segment a segment type violation exception occurs.

The data segments may only be loaded with data type descriptors. If an attempt is made to load a code segment into a data segment a segment type violation exception occurs.

The protection rules refer to a privilege level. The current privilege level of the processor is maintained in the status register, this is called the CPL.

TRIPLE MODE REDUNDANCY (TMR)

Triple mode redundancy is a feature to improve the reliability of the system. Operations are performed in triplicate and majority logic used to determine the correct results.

The Table888 MMU features a triple redundancy mode that was spawned when it was believed that problems due to bad memory were cropping up. In triple redundant mode memory loads and stores are performed in triples. Ideally each load/store operation of the triple goes to a different memory bank. For example, a store operation stores the data to an address in memory bank#1, then repeats the store to memory bank#2 and #3. Part of the output address is supplied by a two-bit counter which selects the DRAM bank. This reduces the effective amount of memory that the processor sees by a factor of four. The triple redundant memory loads and stores also use three times as many memory cycles, and so slow the program down considerably.

During a load operation, the operation is repeated three times, each time loading into a separate load buffer. When all three loads are complete the values are compared using majority logic. Whichever values are the most common (2 or all 3 bits the same) are deemed the correct values.

There is currently no provision to correct values in memory if a bit error occurs.

Table888's CR0 bits 6, 7 and 8 enable triple mode redundancy for reads, writes, and instruction fetches (execute) respectively.

Note that triple mode redundancy should be turned off while performing I/O operations. Some I/O devices reset internal values automatically on the first read of a register. These devices would not return valid data if triple mode redundancy were on. Additionally, performing three writes to an I/O device likely wouldn't have the desired effect. For instance, one wants to transmit a single character using a UART device, not a character repeated three times which is what happens with triple redundant stores.

Triple mode redundancy on the register file is also available as a build option.

PERFORMANCE MEASUREMENT / COUNTERS

In some processors there are performance measurement registers present. These are particularly present in machines designed for research purposes. A common register is the tick count register which simply increments every clock cycle. The tick count register may be settable, or it may simply count beginning at zero after a reset.

There may be other performance registers available such as a breakdown of counts for various types of instructions. For example, the number of instruction fetches, the number of load or store operations.

It's also common in embedded systems to have built in counters. The counters are used with comparators to provide periodic interrupt capabilities. A periodic interrupt source is a commonly present hardware component, often integrated with the processor. Many simple software task switchers use a periodic interrupt in their operation.

Table888 uses an external interrupt to provide periodic interrupts. The Qupls MPU has an interval timer component to it, which connects to the interrupt controller.

TICK COUNT

The Qupls tick count register begins counting from zero after a reset and is not alterable. This register may be used to measure things like the number of clock cycles per instruction. It may also be used as a micro-delay timer. It is not recommended to use the tick count for precise timing measurement. The frequency of the tick count varies with processor frequency. A wall-clock time register is better suited to many forms of measurement.

POWER MANAGEMENT

Related to performance is power management. Decreasing the amount of power consumed often means decreasing the performance. One means to decrease power consumption is to limit the clock frequency. Power consumed is proportional to frequency, so lowering the operating frequency lowers the power consumption. Another means to reduce power consumption is to gate off functional units that are not in use. Table888 or Qupls doesn't provide this capability.

Some cpu's provide instructions that allow the clock to be stopped until an external event occurs. This often called a stop (STP) or halt (HLT) instruction.

FLOATING POINT

The author must confess his knowledge of floating point is somewhat limited, but constantly growing. The author's experience is limited mainly to using double precision numbers with banking applications. One can get by without hardware floating point. Most early micro-processors did not include floating point support. Floating point support is often implemented in software. External floating-point units were later offered as an optional co-processor. Finally floating-point operations were incorporated directly into the cpu chip. Floating-point support has become more common as transistor budgets have increased. Floating point almost exclusively follows the IEEE standard. Very few floating-point units are non-standard these days.

PRECISION

One of the issues with floating point is the available precision. There are a number of standard precisions possible. Sometimes software is used in lieu of greater precision than that available with hardware. Higher precision numbers may be built up out of lower precision representations by using multiple values but it gets to be complex. Try having a look at the double-double precision library. There is movement lately towards support for quadruple precision numbers in hardware. It seems that people like their precision, especially in financial, scientific and engineering applications. Achieving higher precision is often slower and lower performance than lower precision. For some applications where performance is critical, and precision is not required low precision floating point may be in use. The precision required is application dependent. Table888 only supports double precision operations. The Qupls ISA has support for four precisions, half, single, double, and quad precision. Precision can also be applied to integer operations.

OPERATIONS

Not too long ago, floating-point coprocessors typically supported a wide range of operations including trigonometric and exponential / logarithmic functions all in hardware. A more recent trend is to provide only basic operations in hardware.

Qupls includes floating point operations as part of the instruction set. Only basic operations such as FADD, FSUB, FMUL, FDIV and multiply-add are supported. Fortunately, the floating-point operations don't take many cycles to complete (they are faster than an integer divide for instance).

The floating-point operations unit itself is a module separate from the processor, but incorporated within it. The unit is capable of much higher performance than achieved in the processor implementation. Most operations can be pipelined and a new operation can start every clock cycle (excepting divide). However, this feature is not used when implemented in Qupls.

FLOATING POINT NUMBER FORMAT

The floating-point number format used by Qupls is the IEEE standard double precision format:

1	1	10	52 + 1 hidden bit
S _m	S _e	EEEEEEEEE	.MMMMMM.....MMMMMM

S_m = sign of mantissa

S_e = sign of exponent

The exponent and mantissa are both represented as two's complement numbers, however the sign bit of the exponent is inverted.

S_e EEEEEEEEE	
1111111111	Maximum exponent
....	
0111111111	exponent of zero
....	
0000000000	Minimum exponent

The exponent ranges from -1024 to +1023

Half, single, and quad precision are also supported.

Qupls also supports quad precision *decimal* floating-point.

FLOATING POINT REGISTERS

Many architectures keep separate register files for floating-point and general-purpose registers. This helps improve floating-point performance which often relies on a lot of registers. It also compartmentalizes the floating-point making it possible to configure without floating-point support.

For Qupls floating-point values may be stored in the general-purpose register array.

PIPELINE DESIGN

Qupls is a four-way superscalar out-of-order design. The pipeline is very complex and deep with about nine stages to it.

Table 888 is a non-overlapped pipelined design. A pipelined design implements the processor with a number of pipeline stages that data and instructions pass through. Most processors are pipelined in one fashion or another. In an overlapped pipeline design, there can be multiple instructions and multiple data items in the pipeline at the same time. Each instruction and data item can be present in each stage of the pipeline. Data and instruction dependencies between pipeline stages are resolved by hardware. An overlapped pipeline design is like a bucket-brigade where every person in the line has a bucket of water. A non-overlapped design is like a bucket brigade where there is only a single bucket of water available to be handled. The Table 888 design does not use an overlapped pipeline, an overlapped pipeline is (a) more complex to implement, trickier to debug, harder to understand and (b) results in a slightly lower clock frequency for the design. However, the overall performance of an overlapped pipelined design is much greater than that of a non-overlapped design (for example by about a factor of two or more). The Raptor64 is an example of an overlapped-pipelined design. It has a CPI of around 1.5. The RTF65003 is a non-overlapped design, it has a CPI of about 3.0. The clock frequencies of the designs are comparable, although the RTF65003 has a slightly higher clock frequency achievable.

A superscalar pipeline design has parallel pipeline lanes associated with it. It is like multiple lines of bucket-brigade where everyone has a bucket.

PROCESSOR STAGES / STATES

This section gives a general overview of what is done during each pipeline stage. The description of these stages is not particular to this design. These stages are commonly found in many designs. The author seems to intermix the term ‘stage’ with ‘state’. The two are similar. However, a stage may contain multiple states. For instance, an often-identified stage is the memory stage. This stage often contains multiple states for interfacing to memory. A stage is a higher level of looking at the design.

RESET

Long running reset operations, like invalidating the cache, are done by this state. There are usually registers that need to be reset before the processor can begin operations. For instance, in Qupls the TLB registers must be preset to allow access to the ROM boot memory.

IFETCH

Instruction fetch – This is often called a stage because sometimes multiple states are present. At this stage instructions are fetched from memory or a cache and made ready to be decoded. Register file access may also begin at this stage depending on the instruction. This stage transitions to the DECODE stage (or the ICACHE stage if there is a cache miss). This stage may have a branch-target-buffer predictor associated with it.

IALIGN

Instruction align – in a CPU instructions are often not where they need to be for subsequent processing. This stage typically shifts the instruction into a better position. Hardware decoders may be present only at specific positions relative to the beginning of an instruction.

EXTRACT / PARSE

This stage extracts the instruction(s) from the output of the align stage.

DECODE

Decode / Register access – at this stage the instruction is decoded, in parallel registers may be accessed from the register file. Constant values are also setup at this stage.

Decoding instructions is done with a big case statement. All instructions are processed by the instruction decoder. Some of the simpler instructions may also be executed at this stage depending on the pipeline. Instructions that don’t require register values right away may begin execution. This stage transitions into the EXECUTE stage or back to the IFETCH stage for some instructions. This stage also transitions in the memory load and store stages.

REGISTER FILE ACCESS

During the decode stage, register file access may begin as well.

In the Table888 ISA the destination register field “floats around” while the Rs1, Rs2, and Rs3 register read ports are always located at the same positions in the instruction set. For Qupls all the register ports are always located in the same position of the instruction. This allows the incoming instruction to feed the register port number directly to the register file to begin reading registers right away. The destination register field can “float” because it isn’t needed until the register file is updated during the next IFETCH

cycle. This means that the destination register can be set in the decode stage. Shown in the code below, the register specs are taken directly from the IR (instruction register) while the Rd field is another register waiting to be loaded in the DECODE stage.

```
wire [7:0] Rs1 = ir[15:8];
wire [7:0] Rs2 = ir[23:16];
wire [7:0] Rs3 = ir[31:24];
reg [7:0] Rd;
```

RENAME

Registers are renamed at this stage to remove false dependencies. The rename stage is present only in higher performance designs.

ENQUEUE

This stage places a decoded and renamed instruction into a queue for further processing.

SCHEDULE

Instructions are scheduled for execution. This stage may not be present depending on the pipelining. For a simpler pipeline the instructions simply execute in sequence, there is no reason to schedule them. For a more complex machine where instructions are in a queue the scheduler will attempt to pick the best instruction to execute that has ready arguments. The scheduler in a superscalar design can pick multiple instructions to execute at the same time.

EXECUTE

At this stage instructions are “executed”. Results are calculated based on the decoding of the previous stage.

This is the last stage for many instructions. Branches and other control flow instructions are executed during this stage. Memory loads and stores are also begun. It is possible to execute any instructions now because the register values from the register fetch or decode stage are stable.

By the time the EXECUTE stage is reached, all instructions will have been setup for execution, or already executed in the DECODE stage. Once again, like the DECODE stage, the EXECUTE stage uses a big case statement. At the end of the case list there is a default case. This is the place that unimplemented instructions would be handled. The normal procedure would be to invoke an unimplemented instruction exception. However, for simplicity this processor just treats the unimplemented instruction like a NOP operation.

Table888 approaches ALU operations by using an inline ALU to keep things simple. The ALU is incorporated directly into the EXECUTE state. Usually, the ALU is a separate distinct unit. Having a distinct ALU unit would probably allow for better optimization.

Qupls has two ALUs.

MEMORY STAGE

During this stage data is loaded from or stored to memory. This stage often contains multiple load and store states.

WRITEBACK

At this stage results are written to the register file.

COMMIT

At this stage the result of instruction execution is committed to the machine's architectural state.

INSTRUCTION CACHE

It's almost pointless to try to execute instructions at a high clock frequency without an instruction cache present. An instruction cache adds much to the performance of a machine. As much as 75% of memory accesses can be for instruction fetches. Loading of the instruction cache can make use of burst memory transactions, which further increases performance. Without an instruction cache, performance is limited by the speed of external memory. External memory tends to be quite slow compared to processor speeds. Without a cache there can be no overlapping of instruction fetches when another device is accessing memory, and the CPU must wait while the device does its memory access. If one anticipates operating without an instruction cache, and with long memory cycle times, one can develop a processor that uses lots of clock cycles to execute instructions. Perhaps a bit-serial resource scrounging processor could be developed.

If one wants instructions to fetch from an instruction cache that must be accounted for during the instruction fetch stage. It is sometimes desirable to bypass an instruction cache during instruction fetches. That means there must be a multiplexer somewhere to switch between cached and un-cached instructions.

NICE-TO-HAVE HARDWARE FEATURES

Clock stopping. Ideally the processor should be able to stop the clock under certain conditions. This is often done with a stop (STP) instruction. The Stop instruction often puts the processor in a lower power mode to conserve energy.

As it is now, the processor only implements 32 address bits for external addressing. 32 bits is enough to support 4GB of memory. The board has only a 128MB of memory, so it would be wasteful to implement a 64-bit addressing scheme.

Checking for unaligned memory access. Currently the processor does not validate that the address for data is properly aligned. It'll go ahead and try to load data from unaligned addresses if they are specified that way; but it won't work properly.

Check for unimplemented instructions. Unimplemented instructions should exception to a handler routine. This isn't present in the processor, and it just treats an unimplemented instruction like a NOP.

Additional arithmetic operations such as square root, minimum and maximum functions.

Compare-and-swap and other instructions supporting semaphore operations.

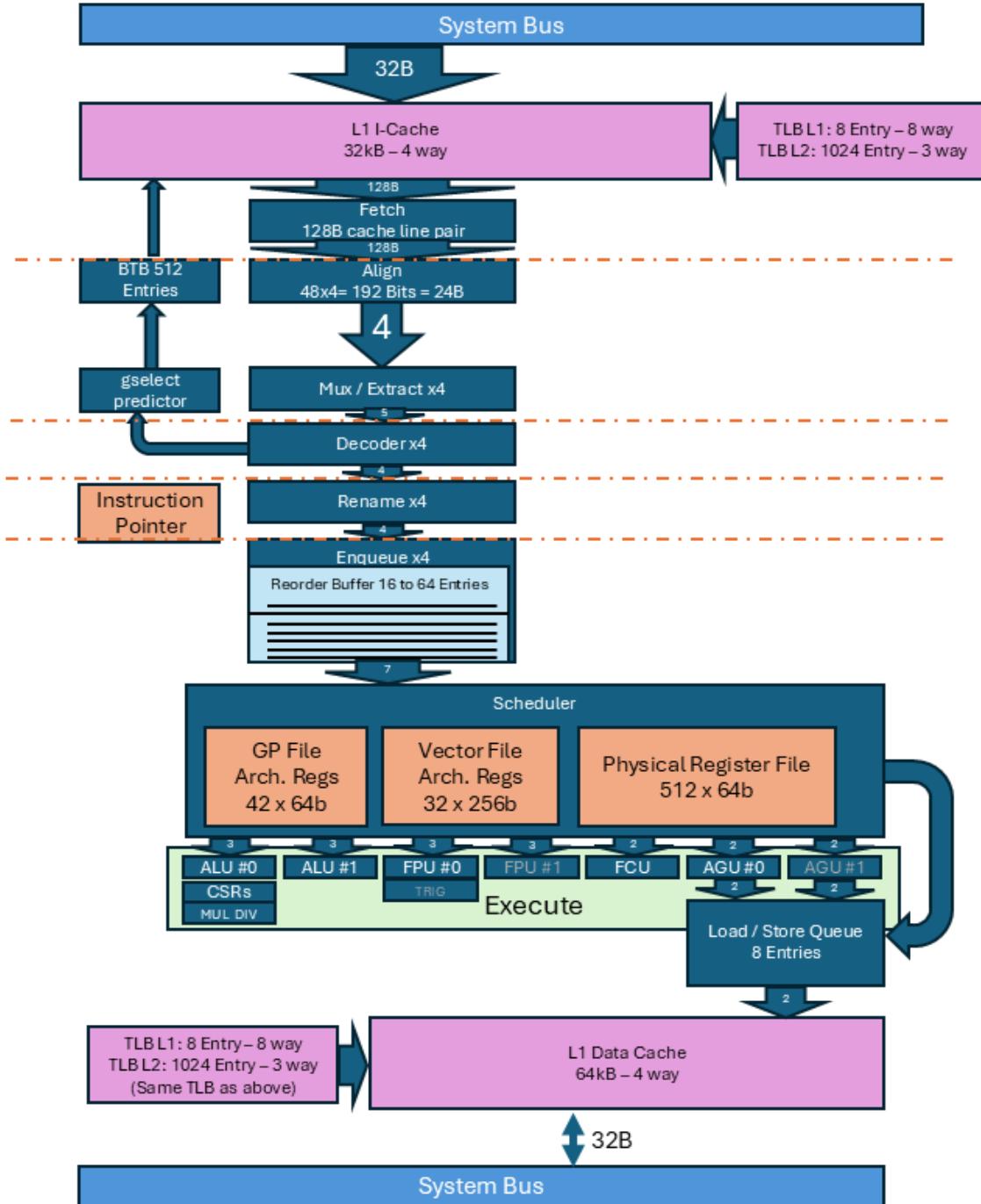
Protection mechanism.

Q+

CPU BLOCK DIAGRAM

Please refer to Slides for more detail.

Q+ Block Diagram



PROGRAMMING MODEL

REGISTER FILE

GENERAL PURPOSE REGISTERS

The register file contains 32 64-bit general purpose registers.

The first 32 registers may be assigned any use by the application software (compiler).

The register file is *unified* and may hold either integer or floating-point values.

Register code 0 is special in that it always reads as a zero.

REGISTER USAGES

Regno	Moniker	Usage
Variable assignments		
0	0	Always zero – read only
1 to 31	~	Compiler assigned usage
28	CP	Suggested use: context pointer
29	LR	Suggested use: link register
30	FP	Suggested use: frame pointer
31	SP	Suggested use: stack pointer

HIDDEN REGISTERS

	Fixed assignments	
32	SSP	Safe stack pointer (hidden from app)
33 to 34	MO0 to MO1	Micro-op temporaries
35 to 63		Unimplemented register selection

LOGICAL REGISTERS

There are 170 logical registers in the design.

FLOATING-POINT REGISTERS

All registers may hold either integer or floating-point values. Pairs of registers may be used to 128-bit values or capabilities.

PHYSICAL REGISTERS

There are 512 general purpose physical registers in the CPU. This provides rename coverage for the 170 logical registers in the design. On average there are 3.0 renamed registers available for every logical register.

REGISTER FLAGS

Each register has a value part and a flags part. The value is 64-bit and the flags are 8-bit.

	7	6	5	4	3	2	1	0
	Cap	Ptr	Ovf	Res	~	~	~	~

Cap is the capabilities tag bit, required to identify when the register contains a capability.

The Ptr bit is set if the register contains a pointer.

Ovf is set if a signed integer result overflows, or if float-point overflow occurs.

Res is a reserved bit

The use of the remaining four bits is unassigned.

PREDICATE REGISTERS

Predicate registers are part of the general-purpose register file and may be manipulated using the same instructions as for other registers. Each bit of a predicate value corresponds to a byte in the destination register. If the bit in the predicate register is set, then the corresponding byte of the destination register is updated. Otherwise, the byte retains its value.

LINK REGISTERS

First note that any register may be used as a subroutine linkage register. However, there are three registers in the Qupls ABI designated for subroutine linkage. These registers are used to store the address after the calling instruction. They may be used to implement fast returns for three levels of subroutines or to be used to call milli-code routines. The jump to subroutine, [JSR](#), and branch to subroutine, [BSR](#), instructions update a link register. The return from subroutine, [RTS](#), instruction is used to return to the next instruction.

LOOP COUNTER

The loop count register is used in counted loops for some instructions like [BSET](#), [BMOV](#), [BFND](#), and [BCMP](#). Any general-purpose register may be assigned as a loop counter.

INSTRUCTION POINTER

This register points to the currently executing instruction. The instruction pointer increments as instructions are fetched, unless overridden by another flow control instruction. The instruction pointer is 16-bit aligned. Instructions may begin at any wyde address. It is possible to write position independent code, PIC, using IP relative addressing.

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
1	sie
2	hie
3	mie
4	die
5 to 10	ipl
11	ssm
12	te
13 to 14	om
15 to 16	ps
	capsz
17	dbg
18 to 20	mprv
21 to 23	Swstk
24 to 31	cpl
32 to 33	fx
34 to 35	vx
36 to 39	Reserved - state

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.

CAPSZ: indicates the size of a capability (32 or 64 bits).

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

IPL is the interrupt mask level

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

vx	Status	
0	Off	Any vector instruction will cause an exception.
1	Initial	State has initial constant value
2	Clean	State is consistent with last state stored in context
3	Dirty	State has been modified since last context save.

VECTOR PROGRAMMING MODEL

VECTOR REGISTERS

The machine has 32 x 256-bit vector registers.

For an out-of-order superscalar processor registers are renamed. To support vector processing each 64-bit chunk of a vector register needs to be renamed separately. Ideally the register file should be less than about 128 registers, after that performance starts to suffer due to the size of the register file. With the number of 64-bit chunks limited to 128 it puts a constraint on the size of a vector register.

Since each chunk of a vector register needs to be renamed anyway, the general register file renaming logic is used as an efficient solution.

The vector length is specified in bytes.

TAIL ELEMENTS

Tail elements of a vector register are not normally operated on and are not updated. A tail element is an element that comes after the number of elements determined by the vector length. For instance, if the vector length is 20 bytes, the remaining 12 bytes of the register contain tail elements.

The micro-op translator will elide translations of instructions which would only update the tail elements. This may improve performance as only relevant instructions get executed.

VECTOR MASKING

Vector operations may be masked. The vector mask is contained in a general-purpose scalar register. Each bit of the register corresponds to a vector element to be masked. There is a maximum of 64-elements (or lanes) in a vector.

The current vector register width of 256 bits allows only 32 lanes of bytes as a max. If the register width is widened, then more lanes may be supported. If the register width is widened it will likely apply to GPRs as well meaning more mask bits will be available.

For each set bit in the mask register, the result element is enabled in the destination register. For each zero bit in the mask register, the destination element remains unchanged.

Masked off elements do not cause exceptions.

The vector mask register is specified by the Rs3 field of many instructions. In that case the second operation of a dual-operation instruction is not allowed. The mask itself is the second operation.

The SET instructions may be used to generate a mask in a register. A true result will set a bit corresponding to the element in the mask register to one. A false result will set a bit corresponding to the element in the mask register to zero.

Since GPRs contain the mask the full instruction set is available to manipulate masks.

VECTOR OPERANDS

Vector instructions support both vector and scalar operands in most cases. Whether the operand is a vector or a scalar is determined by the VN field of the instruction. In some cases, the instruction itself requires specific operand types. The VN field is not present for all instructions.

VN₄ field: 1 = vector operand, 0 = scalar operand

VN₄ field bit:

3	2	1	0
Rs3	Rs2	Rs1	Rd

If the VN field indicates a vector operand, the sign control for the vector operand comes from the sixth bit of the register spec. as only five bits are needed to specify 32 registers.

UNSUPPORTED FUNCTIONALITY

There are some vector operations available in other CPUs that are not supported in Qupls4. These include widening and narrowing operations. The issue with supporting these is how to propagate the vector information into a different sized vector. Currently the core processes vectors in chunks of 64-bits. The core's capacity to deal with values spanning 64-bits is limited. If the core had a wide data-path for instance 256-bits it would be able to support the operations.

SPECIAL PURPOSE REGISTERS

SC - STACK CANARY (GPR 53)

This special purpose register is available in the general register file as register 53. The stack canary register is used to alleviate issues resulting from buffer overflows on the stack. The canary register contains a random value which remains consistent throughout the run-time of a program. In the right conditions, the canary register is written to the stack during the function's prolog code. In the function's epilog code, the value of the canary on stack is checked to ensure it is correct, if not a check exception occurs.

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

See status register.

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

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 8 bits are implemented. The high order bits read as zero and are not updateable.

U_FPCSR (CSR – 0x?014)

U_FPCSR - FLOATING POINT STATUS AND CONTROL REGISTER (CSR 0X0014)

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

Bit		Symbol	Description
63:53			reserved
52		inexact	inexact
51		dbz	divide by zero
50		under	underflow
49		over	overflow
48		invop	invalid operation
47		~	reserved
46:44	RM	rm	rounding mode
43	E5	inexe	- inexact exception enable
42	E4	dbzxe	- divide by zero exception enable
41	E3	underxe	- underflow exception enable
40	E2	overxe	- overflow exception enable

39	E1	invopxe	- invalid operation exception enable
38	NS	ns	- non standard floating point indicator

Result Status

32		fractie	- the last instruction (arithmetic or conversion) rounded intermediate result (or caused a disabled overflow exception)
31	RA	rawayz	rounded away from zero (fraction incremented)
30	SC	C	denormalized, negative zero, or quiet NaN
29	SL	neg <	the result is negative (and not zero)
28	SG	pos >	the result is positive (and not zero)
27	SE	zero =	the result is zero (negative or positive)
26	SI	inf ?	the result is infinite or quiet NaN

Exception Occurrence

21 to 25			reserved
20	X6	swt	{reserved} - set this bit using software to trigger an invalid operation
19	X5	inerx	- inexact result exception occurred (sticky)
18	X4	dbzx	- divide by zero exception occurred
17	X3	underx	- underflow exception occurred
16	X2	overx	- overflow exception occurred
15	X1	giopx	- global invalid operation exception – set if any invalid operation exception has occurred
14	GX	gx	- global exception indicator – set if any enabled exception has happened
13	SX	sumx	- summary exception – set if any exception could occur if it was enabled - can only be cleared by software

Exception Type Resolution

8 to 12			reserved
7	X1T	cvt	- attempt to convert NaN or too large to integer
6	X1T	sqrtx	- square root of non-zero negative
5	X1T	NaNCmp	- comparison of NaN not using unordered comparison instructions
4	X1T	infzero	- multiply infinity by zero
3	X1T	zerozero	- division of zero by zero
2	X1T	infdiv	- division of infinities
1	X1T	subinfx	- subtraction of infinities
0	X1T	snanx	- signaling NaN

U_FXCSR – FIXED POINT CONTROL AND STATUS (CSR – 0X0015)

This register contains control bits for fixed point arithmetic.

Bit			Description
0 to 2		Rm	rounding mode
3 to 7			reserved
8 to 15		Dp	Decimal position

[U/S/H/M]_SCRATCH – CSR 0X?041

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

S_ASID (CSR 0X101F)

This register contains the address space identifier (ASID) or memory map index (MMI). The ASID is used in this design to select (index into) a memory map in the paging tables. Only the low order sixteen bits of the register are implemented.

S_KEYS (CSR 0X1020 TO 0X1027)

These eight registers contain the collection of keys associated with the process for the memory lot system. Each key is twenty-four bits in size. All eight registers are searched in parallel for keys matching the one associated with the memory page. Keyed memory enhances the security and reliability of the system.

			23	0
1020				key0
1021				key1
...				...
1027				key7

M_CORENO (CSR 0X3001)

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

M_TICK (CSR 0X3002)

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

M_SEED (CSR 0X3003)

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

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_BADADDR (CSR 0X3007)

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

M_BAD_INSTR (CSR 0X300B)

This register contains a copy of the exceptioned instruction.

M_SEMA (CSR 0X300C)

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

M_KVEC – KERNEL VECTORS (CSR 0X3030 TO 0X3034)

This set of registers contains the exception handler address for each operating mode, including a vector for debug.

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

Reg #	
0x3030	User mode vector
0x3031	supervisor mode vector
0x3032	Hypervisor mode vector
0x3033	Secure mode vector
0x3034	Debug vector

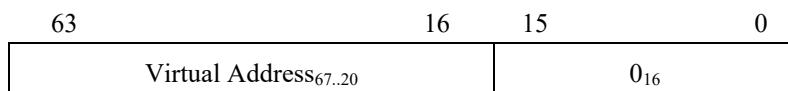
These registers contain the address of the exception handler table for a given operating mode. TVEC[0] to TVEC[2] are used by the REX instruction.

M_SR_STACK (CSR 0X3080 TO CSR 0X3087)

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

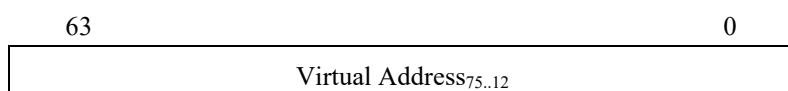
M_IOS – IO SELECT REGISTER (CSR 0X3100)

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 16 bits of this register should be zero and are ignored.



M_CFGS – CONFIGURATION SPACE REGISTER (CSR 0X3101)

The location of configuration space is determined by the contents of the CFGS control register. The select is for a 256MB region. This address is a virtual address. The low order 12 bits of this address are assumed to be zero. The default value of this register is \$FF...FD0000



M_EIP (CSR 0X3108 TO 0X310F)

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

Reg #	Name
0x3108	EIP0
...	
0x310F	EIP7

U_VLEN – (CSR 0X0200) - VECTOR LENGTH

This register contains the number of bytes per vector for a given type of vector. The size should be a multiple of the register size (REGSZ from CPUINFO). This register aids in coding vector routines in a generic fashion. Valid values are REGSZ to REGSZ*8 (8 to 64) in multiples of REGSZ. A value of zero will result in a zero-length vector; vector instructions will perform NOPs instead.

The vector length comes into play when translating vector instructions into micro-ops. Only as many micro-ops as needed for the vector length are added to the micro-op buffer. A consequence of the translation is that the tail of the vector register may not be modified.

63	40	39	32	31	24	23	16	15	8	7	0
~		Addresses		Character		Fixed		Floats		Integers	

U_VELSZ – (CSR 0X0201) - VECTOR ELEMENT SIZE

This register contains the size of a vector element for a given type of vector. Vector operations may use this size to determine the size of elements processed by an instruction. This register aids in coding vector routines in a generic fashion. Both the integer and floating-point size are specified by this register. Integer size is specified in the lower eight bits; floating-point size is specified in bits eight to fifteen.

63	40	39	32	31	24	23	16	15	8	7	0
~		Addresses		Character		Fixed		Floats		Integers	
		1,2,4 or 8		1 to 64		1,2,4,8 or 16		2,4,8 or 16		1,2,4,8 or 16	

Note that there are both sized and unsized instructions. The sized instructions will treat the vector as the size of the instruction indicates. The unsized instructions will treat the vector according to the size specified by the U_VELSZ register.

U_VSTART – (CSR 0X0202) - VECTOR START

This register contains the number of the element to begin vector processing at, used after exception processing. Normally this register is zero indicating to start at element zero, but exception processing may set it to another value.

U_NANCR – (CSR 0X0210) – NAN CONTROL REGISTER

This register contains the number of the element to begin vector processing at, used after exception processing. Normally this register is zero indicating to start at element zero, but exception processing may set it to another value.

OPERATING MODES

The core operates in one of four basic modes: application/user mode, supervisor mode, hypervisor mode or secure (machine) mode. When an interrupt or exception occurs, or when debugging is triggered, a jump is made to the appropriate vector for the cause. The exception processing routine at the given operating level may choose to escalate the exception to the next higher level.

On power-up the core is running in secure mode. An RTE instruction must be executed to leave secure 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 secure mode.

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

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

EXCEPTIONS

OVERVIEW

There are two types of exceptions: external asynchronous events caused by hardware, and internal synchronous events caused by software. Both of these exception types share logic in the CPU.

EXTERNAL INTERRUPTS

OVERVIEW

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 flag is set in the pipeline, so the CPU knows to perform interrupt processing when the instruction commits.

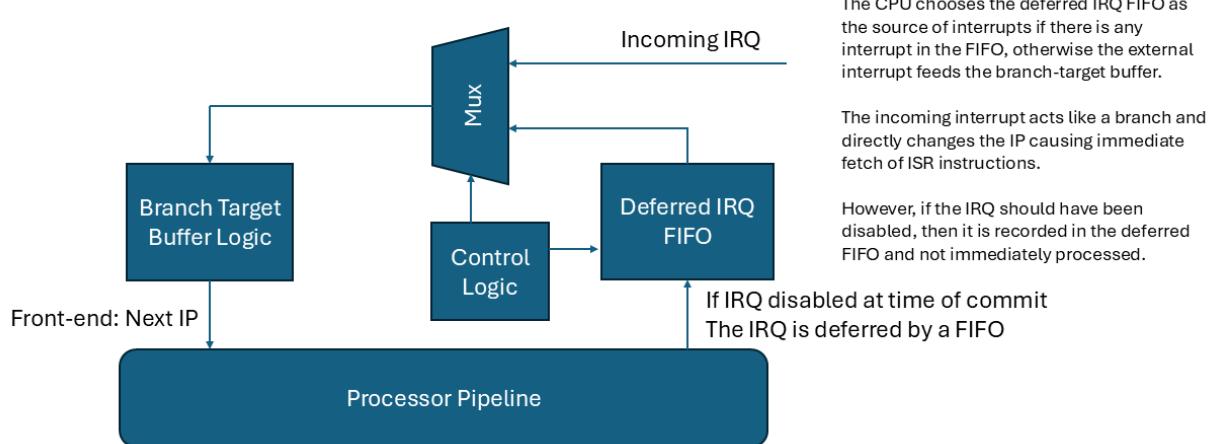
External interrupts are asynchronous events caused by hardware.

FLOW CONTROL TRANSFER

On detection of an unmasked external interrupt, instruction fetch will be redirected immediately to the interrupt subroutine. The address is supplied by an external interrupt controller. This minimizes interrupt latency.

IRQ CONNECTION DIAGRAM

Qupls4: External Interrupt Handling



PRIORITY LEVELS

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. The global interrupt enable must also be set.

IRQ PROTOCOL

The CPU follows a protocol for communicating with the external MSI (message signalled interrupt) controller.

When an MSI interrupt occurs, the controller signals an interrupt to the CPU and places message information on a bus to the CPU. The message information includes the priority level, address vector, software stack required and operating mode of the interrupt.

When the CPU sees the interrupt signal it sends an ACK to the MSI controller, so the controller knows it can place the next interrupt on the bus.

The interrupt request is treated like a branch by the CPU and overrides the branch target buffer logic to supply the next instruction pointer address. The processor will then begin fetching instructions for the IRQ handler. This mechanism provides low latency for the interrupt as instructions begin fetching the same cycle the interrupt is recognized by the CPU.

THE IRQ FIFO

There is a complication that the interrupt may have been disabled by an instruction already in the CPU's pipeline when the CPU goes to process the interrupt. The CPU has already acknowledged the interrupt to the MSI controller, and the interrupt would be lost if it were not for the FIFO. When the interrupt reaches the end of the pipeline a check is made to ensure that the interrupt is still enabled. If the interrupt is no longer enabled, then the interrupt is deferred by placing it in an internal FIFO, otherwise the interrupt is processed, and exception handing begins.

The CPU checks the internal FIFO for deferred interrupts before acknowledging a request made by the MSI controller.

Thus, interrupts are serviced in the order they occur.

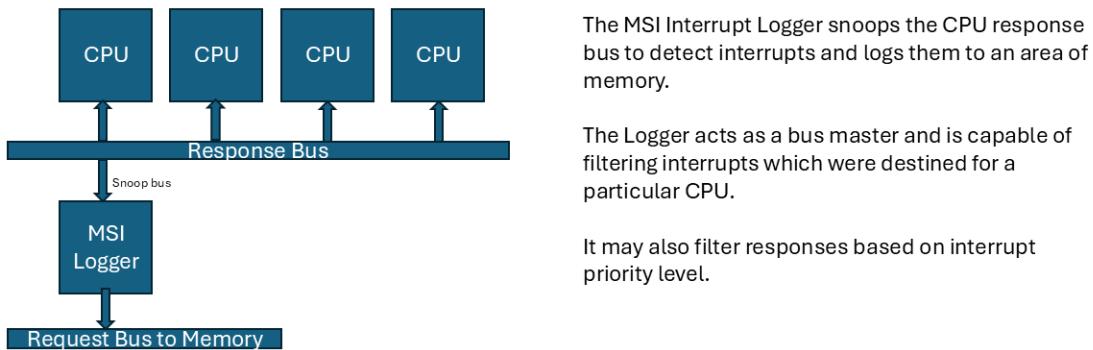
IRQ MESSAGE INFORMATION

The information required for an interrupt is stored in the [MSI controller](#). This device is described later in the document. The controller may store information for up to 2048 vectors for each operating mode of the CPU. The controller is parameterized to allow system customization.

IRQ LOGGING

Interrupts may be logged to a table in memory by the system MSI logging device which logs interrupts appearing on the response bus. The logger monitors the response bus and may filter interrupts for particular cores.

Qupls4: MSI Interrupt Logging



IRQ DISABLED EXCEPTION

If the CPU detects that IRQs have been disabled for too long of a time, and exception will be generated.

EFFECT ON OPERATING MODE

For external interrupts the CPU is switched to the operating mode specified by the interrupt message. This is set in the interrupt vector table of the MSI interrupt controller.

INTERNAL EXCEPTIONS

OVERVIEW

Internal exceptions are synchronous and occur as a result of executing software.

EFFECT ON OPERATING MODE

For an internal exception, the operating mode is always switched to the next higher mode. It is up to the mode code to redirect the exception to a lower operating mode when desired.

FLOW CONTROL TRANSFER

An internal exception does not cause an immediate transfer of flow control as the exception may later be found not valid due to speculative execution of code. Flow control only changes at the end of the pipeline when the instruction commits. The address is supplied from one of the kernel vectors.

VECTOR TABLE

The kernel vector table is always used to locate the exception routine. The [kernel vector table](#) is a small table built directly into the CPU for performance. It is accessible as a set of CSR registers. The table has entries for each operating mode plus debug.

When an exception occurs, the CPU loads the instruction pointer with the proper vector from the kernel vector table. The entry should point to the exception handler. It is suggested that the exception handler should lookup a further vector for the corresponding cause from a table.

The exception routine may then redirect the exception to a lower operating mode (de-escalate) using the DESC instruction.

The exception handler must process the cause code, possibly loading a handler address from a table.

Cause	Usage
0	Debug Breakpoint (BRK)
1	Debug breakpoint – single step
2	Bus Error
3	Address Error
4	Unimplemented Instruction
5	Privilege Violation
6	Page fault
7	Instruction trace
8	Stack Canary
9	Abort
10	Interrupt
11	Non-maskable interrupt
12	Reset
13	Alternate Cause
14, 15	Reserved

32	
33	
34	Instruction Address
33 to 63	Trap #1 to 31
	Applications Usage
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
----------	----------

DESCRIPTION OF CAUSE CODES

BREAKPOINT FAULT (0)

The breakpoint instruction, 0, was encountered.

SINGLE STEP FAULT (1)

This vector is invoked after the execution of an instruction if single step mode is active.

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 FAULT (3)

This exception fault occurs if there is an attempt to set the instruction pointer to an odd address.

UNIMPLEMENTED INSTRUCTION FAULT (4)

An unimplemented instruction causes this fault.

PRIVILEGE VIOLATION FAULT (5)

An attempt was made to access memory that is not authorized at the current privilege level. It may also be triggered by an attempt to execute a privileged instruction from a non-privileged mode.

PAGE FAULT (6)

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

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)

Software may redirect interrupts here. The external interrupt signal was asserted, and the interrupt level was greater than the current mask level.

NON-MASKABLE INTERRUPT (11)

The external NMI interrupt signal was asserted. This interrupt is always recognized.

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 15.

BOTH EXTERNAL AND INTERNAL EXCEPTIONS OR INTERRUPTS

EXCEPTION STACK

The status register and instruction pointer are pushed onto an internal stack when an exception or interrupt occurs. This stack is at least eight entries deep to allow for nested interrupts and multiply nested traps and exceptions.

PRECISION

Exceptions in Qupls4 are precise. They are processed according to program order of the instructions. External interrupts are processed in order of occurrence. 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.

RESET

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

The core begins executing instructions at the address defined for the reset vector. The reset vector address is a core parameter. By default, reset jumps to \$FF...FFC00. All registers are in an undefined state.

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 48-bit parcels format. Specific formats are listed under the instruction set description section of this book.

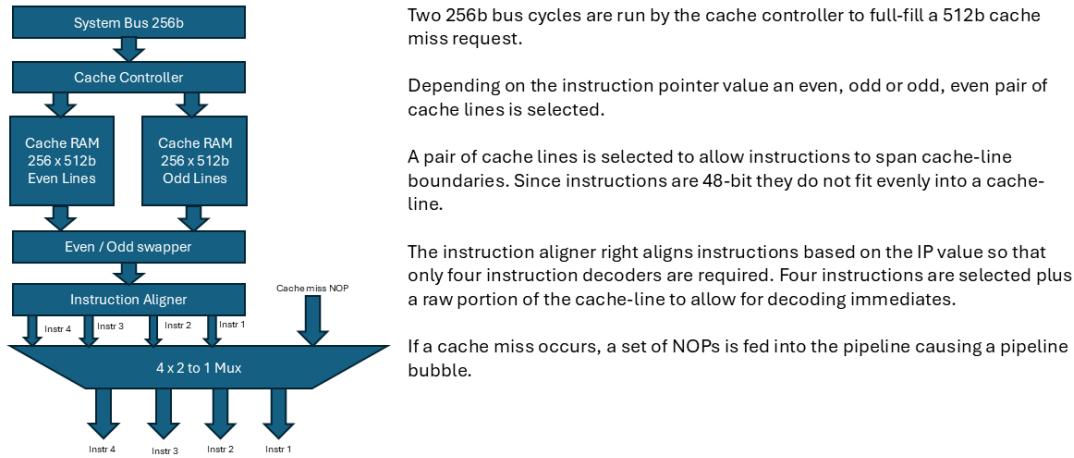
A 48-bit parcel was chosen because it's simpler for a hobbyist design and to limit the amount of multiplexing taking place for an instruction read. The larger number of registers required to support vector processing requires large register specifiers in instructions. To get all the register specifiers to fit with additional opcode information a larger instruction is needed.

The author found that determining the length of an instruction and selecting the next instruction to fetch based on a varying length affected the timing of the core. A simpler design makes it easier to achieve a higher clock rate.

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 Qupls4.

Qupls2026: Instruction Cache



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.

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.

CAPABILITIES TAG CACHE

The capabilities tag cache supports the capability system. Every sixteen bytes of memory has a capabilities tag bit associated with it. If there is a valid capability stored at the address the tag bit will be set, otherwise it will be clear. The tag cache is 512 lines of 16 bytes of tag bits for a capacity of 64k tags. It is a direct mapped 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...FDxxxx 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. There is also field in the load instructions that allows bypassing the data cache.

FETCH RATE

The fetch rate is four instructions per clock cycle.

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 RTD 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 RTD against the address fetched for the RTD 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.

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.

The author has attempted to add logic to fetch along alternate branch paths when the instruction cache is idle. This logic is not implemented yet, but appears in RTL code.

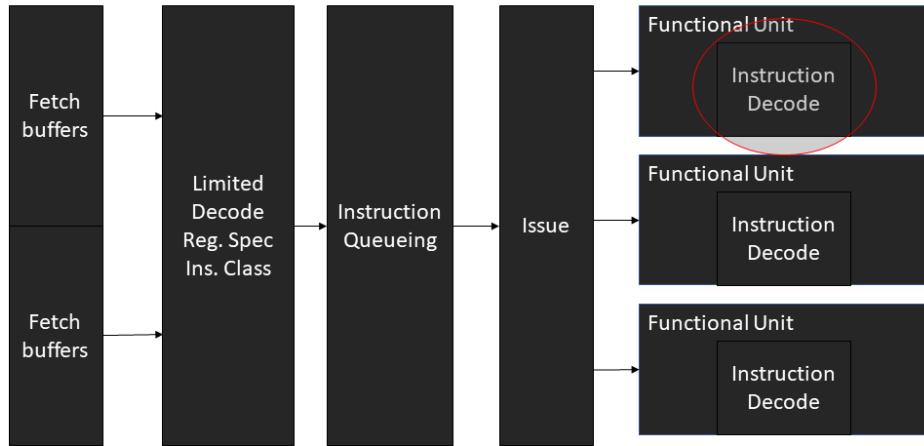
BRANCH TARGET BUFFER (BTB)

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

DECODE LOGIC

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

Placement of Instruction Decode

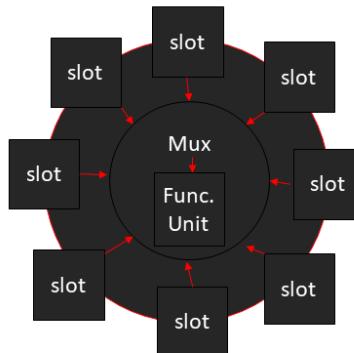


Limited decode takes place between fetch and queue. Between fetch and queue register specifications are decoded along with general instruction classes for the benefit of issue. A handful of additional signals (like sync) that control the overall operation of the core are also decoded. Much of the instruction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

INSTRUCTION QUEUE (ROB)

The instruction queue is a 32-entry re-ordering buffer (ROB). The instruction queue tracks an instruction's 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 guise 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 Qupls4 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 256MB region of memory in the address range \$FF...D0000000 to \$FF...DFFFFFFF. Each device configuration block is aligned on a 4kB boundary. There is thus a maximum of 64k 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	Config Block Address	Default Address
Interrupt Controller	0	6	0	~	\$FF..FD0030000	\$FF..FEE2xxxx
Interval Timers	0	4	0	29	\$FF..FD0020000	\$FF..FEE4xxxx
Memory Region Table	0	12	0	~	\$FF..FD0060000	\$FF..FEEFxxxx
Page Table Walker	0	14	0	~	\$FF..FD0070000	\$FF..FFF4xxxx
Hardware Card Table	0		0	~		

Function is mapped to address bits 12 to 14

Device is mapped to address bits 15 to 19

Bus is mapped to address bits 20 to 27

MEMORY MANAGEMENT

This section is somewhat pedantic and reviews technical approaches before getting into Qupls details.

BANK SWAPPING

About the simplest form of memory management is a single bank register that selects the active memory bank. This is the mechanism used on many early microcomputers. The bank register may be an eight bit I/O port supplying control over some number of upper address bits used to access memory.

THE PAGE MAP

The next simplest form of memory management is a single table map of virtual to physical addresses. The page map is often located in a high-speed dedicated memory. An example of a mapping table is the 74LS612 chip. It may map four address bits on the input side to twelve address bits on the output side. This allows a physical address range eight bits greater than the virtual address range. A more complicated page map is something like the MC6829 MMU. It may map 2kB pages in a 2MB physical address space for up to four different tasks.

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. Qupls 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+ is a memory mapped I/O device and has a device configuration block associated with it. The default address of the device is \$FF...FEEF0000.

REGION TABLE DESCRIPTION

Reg	Bits	Field	Description
0000	128	Pmt	associated PMT address
0010	128	cta	Card table address
0020	128	at	Four groups of 32-bit memory attributes, 1 group for each of user, supervisor, hypervisor and machine.
0030	128	...	Not used
0040 to 01F0		...	7 more register sets

PMT ADDRESS

The PMT address specifies the location of the associated PMT.

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

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 <table border="1"> <tr><td>G</td><td></td></tr> <tr><td>0</td><td>byte accessible</td></tr> <tr><td>1</td><td>wyde accessible</td></tr> <tr><td>2</td><td>tetra accessible</td></tr> <tr><td>3</td><td>octa accessible</td></tr> <tr><td>4</td><td>hexi accessible</td></tr> <tr><td>5 to 7</td><td>reserved</td></tr> </table>	G		0	byte accessible	1	wyde accessible	2	tetra accessible	3	octa accessible	4	hexi accessible	5 to 7	reserved
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														

OVERVIEW

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

Physical memory attributes are stored in an eight-entry region table. Three bits in the PTE select an entry from this table. The operating mode of the CPU also determines which 32-bit set of attributes to apply for the memory region.

Most of the entries in the table are hard-coded and configured when the system is built. However, they may be modified.

Physical memory attributes checking is applied in all operating modes.

The region table is accessible as a memory mapped IO, MMIO, device.

PAGE MANAGEMENT TABLE - PMT

OVERVIEW

For the first translation of a virtual to physical address, after the physical page number is retrieved from the TLB, the region is determined, and the page management table is referenced to obtain the access rights to the page. PMT information is loaded into the TLB entry for the page translation. The PMT contains an assortment of information most of which is managed by software. 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 access rights table entries (PMTEs). There are as many PMTEs as there are pages of memory in the region.

For subsequent virtual to physical address translations PMT information is retrieved from the TLB.

As the page is accessed in the TLB, the TLB may update the PMT.

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	~9	C	E	AL ₂	~4	mrwx	hrwx	srxw	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.

PAGE TABLES

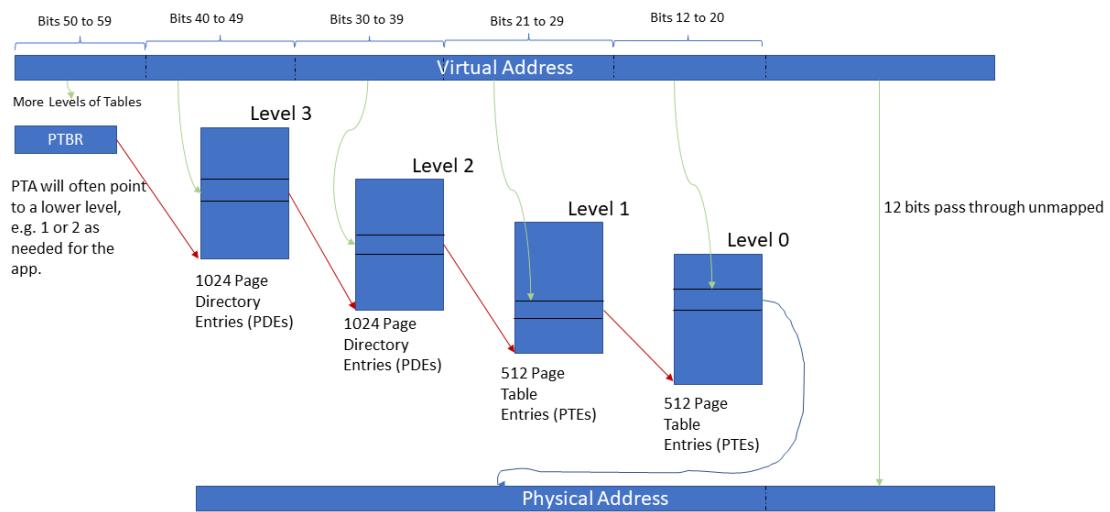
INTRO

Page tables are part of the memory management system used map virtual addresses to real physical addresses. There are several types of page tables. Hierarchical page tables are probably the most common. Almost all page tables map only the upper bits of a virtual address, called a page. The lower bits of the virtual address are passed through without being altered. The page size often 4kB which means the low order 12-bits of a virtual address will be mapped to the same 12-bits for the physical address.

HIERARCHICAL PAGE TABLES

Hierarchical page tables organize page tables in a multi-level hierarchy. They can map the entire virtual address range but often only a subrange of the full virtual address space is mapped. This can be determined on an application basis. At the topmost level a register points to a page directory, that page directory points to a page directory at a lower level until finally a page directory points to a page containing page table entries. To map an entire 64-bit virtual address range approximately five levels of tables are required.

Paged MMU Mapping



INVERTED PAGE TABLES

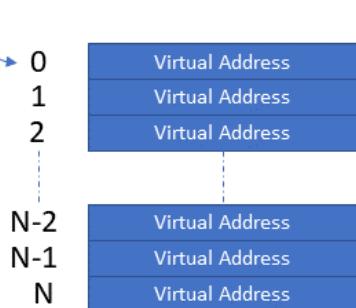
An inverted page table is a table used to store address translations for memory management. The idea behind an inverted page table is that there are a fixed number of pages of memory no matter how it is mapped. It should not be necessary to provide for a map of every possible address, which is what the hierarchical table does, only addresses that correspond to real pages of memory need be mapped. Each page of memory can be allocated only once. It is either allocated or it is not. Compared to a non-inverted paged memory management system where tables are used to map potentially the entire address space an inverted page table uses less memory. There is typically only a single inverted page table supporting all applications in the system. This is a different approach than a non-inverted page table which may provide separate page tables for each process.

THE SIMPLE INVERTED PAGE TABLE

The simplest inverted page table contains only a record of the virtual address mapped to the page, and the index into the table is used as the physical page number. There are only as many entries in the inverted page table as there are physical pages of memory. A translation can be made by scanning the table for a matching virtual address, then reading off the value of the table index. The attraction of an inverted page table is its small size compared to the typical hierarchical page table. Unfortunately, the simplest inverted page table is not practical when there are thousands or millions of pages of memory. It simply takes too long to scan the table. The alternative solution to scanning the table is to hash the virtual address to get a table index directly.

Inverted Page Table

Entry number identifies physical page number



HASHED PAGE TABLES

HASHED TABLE ACCESS

Hashes are great for providing an index value immediately. The issue with hash functions is that they are just a hash. It is possible that two different virtual address will hash to the same value. What is then needed is a way to deal with these hash collisions. There are a couple of different methods of dealing with collisions. One is to use a chain of links. The chain has each link in the chain pointing to the next page table entry to use in the event of a collision. The hash page table is slightly more complicated than as it needs to store links for hash chains. The second method is to use open addressing. Open addressing calculates the next page table entry to use in the event of a collision. The calculation may be linear, quadratic or some other function dreamed up. A linear probe simply chooses the next page table entry in succession from the previous one if no match occurred. Quadratic probing calculates the next page table entry to use based on squaring the count of misses.

CLUSTERED HASH TABLES

A clustered hash table works in the same manner as a hashed page table except that the hash is used to access a cluster of entries rather than a single entry. Hashed values may map to the same cluster which can store multiple translations. Once the cluster is identified, all the entries are searched in parallel for the correct one. A clustered hash table may be faster than a simple hash table as it makes use of parallel searches. Often accessing memory returns a cache line regardless of whether a single byte or the whole cached line is referenced. By using a cache line to store a cluster of entries it can turn what might be multiple memory accesses into a single access. For example, an ordinary hash table with open addressing

may take up to 10 memory accesses to find the correct translation. With a clustered table that turns into 1.25 memory accesses on average.

SHARED MEMORY

Another memory management issue to deal with is shared memory. Sometimes applications share memory with other apps for communication purposes, and to conserve memory space where there are common elements. The same shared library may be used by many apps running in the system. With a hierarchical paged memory management system, it is easy to share memory, just modify the page table entry to point to the same physical memory as is used by another process. With an inverted page table having only a single entry for each physical page is not sufficient to support shared memory. There needs to be multiple page table entries available for some physical pages but not others because multiple virtual addresses might map to the same physical address. One solution would be to have multiple buckets to store virtual addresses in for each physical address. However, this would waste a lot of memory because much of the time only a single mapped address is needed. There must be a better solution. Rather than reading off the table index as the physical page number, the association of the virtual and physical address can be stored. Since we now need to record the physical address multiple times the simple mechanism of using the table index as the physical page number cannot be used. Instead, the physical page number needs to be stored in the table in addition to the virtual page number.

That means a table larger than the minimum is required. A minimally sized table would contain only one entry for each physical page of memory. So, to allow for shared memory the size of the table is doubled. This smells like a system configuration parameter.

SPECIFICS: QUPLS PAGE TABLES

QUPLS HASH PAGE TABLE SETUP

HASH PAGE TABLE ENTRIES - HPTE

We have determined that a page table entry needs to store both the physical page number and the virtual page number for the translations. To keep things simple, the page table stores only the information needed to perform an address translation. Other bits of information are stored in a secondary table called the page management table, PMT. The author did a significant amount of juggling around the sizes of various fields, mainly the size of the physical and virtual page numbers. Finally, the author decided on a 192-bit HPTE format.

V	LVL/BC ₅	RGN ₃	M	A	T	S	G	SW ₂	CACHE ₄	MRWX ₃	HRWX ₃	SRWX ₃	URWX ₃	
PPN _{31..0}														
PPN _{63..32}														
VPN _{37..6}														
VPN _{69..38}														
~4	ASID _{11..0}					~2	VPN _{83..70}							

Fields Description

V	1	translation Valid
G	1	global translation
RGN	3	region
PPN	64	Physical page number
VPN	84	Virtual page number
RWX	3	readable, writeable, executable
ASID	12	address space identifier
LVL/BC	5	bounce count
M	1	modified
A	1	accessed
T	1	PTE type (not used)
S	1	Shared page indicator
SW	3	OS usage

The page table does not include everything needed to manage pages of memory. There is additional information such as share counts and privilege levels to take care of, but this information is better managed in a separate table.

SMALL HASH PAGE TABLE ENTRIES - SHPTE

The small HPTE is used for the test system which contains only 512MB of physical RAM to conserve hardware resources. The SHPTE is 72-bits in size. A 32-bit physical address is probably sufficient for this system. So, the physical page number could be 18-bits or less depending on the page size.

V	LVL/BC ₅	RGN ₃	M	A	T	S	G	SW	CACHE ₄	ASID _{3..0}	HRWX ₃	SRWX ₃	URWX ₃
VPN _{15..0}								PPN _{15..0}					
												ASID _{7..4}	VPN _{19..16}

PAGE TABLE GROUPS – PTG

We want the search for translations to be fast. That means being able to search in parallel. So, PTEs are stored in groups that are searched in parallel for translations. This is sometimes referred to as a clustered table approach. Access to the group should be as fast as possible. There are also hardware limits to how many entries can be searched at once while retaining a high clock rate. So, the convenient size of 1024 bits was chosen as the amount of memory to fetch.

A page table group then contains five HPTE entries. All entries in the group are searched in parallel for a match. Note that the entries are searched as the PTG is loaded, so that the PTG group load may be aborted early if a matching PTE is found before the load is finished.

PTE0
PTE1
PTE2
PTE3
PTE4

Small Page Table Group

For the small page table, a fetch size of 576 bits was chosen. This allows eight SHPTEs to fit into one group.

SIZE OF PAGE TABLE

There are several conflicting elements to deal with, with regards to the size of the page table. Ideally, the hash page table is small enough to fit into the block RAM resources available in the FPGA. It may be practical to store the hash page table in block RAM as there would be only a single table for all apps in the system. This probably would not be practical for a hierarchical table.

About 1/6 of the block RAMs available are dedicated to MMU use. At the same time a multiple of the number of physical pages of memory should be supported to support page sharing and swapping pages to secondary storage. To support swapping pages, double the number of physical entries were chosen. To support page sharing, double that number again. Therefore, a minimum size of a page table would contain at least four times the number of physical pages for entries. By setting the size of the page table instead of the size of pages, it can be worked backwards how many pages of memory can be supported.

For a system using 256k block RAM to store PTEs. $256k / 8 = 32768$ entries. $32,768 / 4 = 8,192$ physical pages. Since the RAM size is 512MB, each page would be $512MB / 8,192 = 64kB$. Since half the pages may be in secondary storage, 1GB of address range is available.

Since there are 32,768 entries in the table and they are grouped into groups of eight, there are 4,096 PTGs. To get to a page table group fast a hash function is needed then that returns a 12-bit number.

Reworking things with a 64kB page size and 32,768 PTEs. The maximum memory size that can be supported is: 2.0 GB. This is only 4x the amount of RAM in the system, but may be okay for demo purposes.

HASH FUNCTION

The hash function needs to reduce the size of a virtual address down to a 10-bit number. The asid should be considered part of the virtual address. Including the asid of 10-bits and a 32-bit address is 42 bits. The first thing to do is to throw away the lowest eighteen bits as they pass through the MMU unaltered. We now have 24-bits to deal with. We can probably throw away some high order bits too, as a process is not likely to use the full 32-bit address range.

The hash function chosen uses the asid combined with virtual address bits 20 to 29. This should space out the PTEs according to the asid. Address bits 18 and 19 select one of four address ranges. the PTG supports seven PTEs. The translations where address bits 18 and 19 are involved are likely consecutive pages that would show up in the same PTG. The hash is the asid exclusively or'd with address bits 20 to 29.

COLLISION HANDLING

Quadratic probing of the page table is used when a collision occurs. The next PTG to search is calculated as the hash plus the square of the miss count. On the first miss the PTG at the hash plus one is searched. Next the PTG at the hash plus four is searched. After that the PTG at the hash plus nine is searched, and so on.

FINDING A MATCH

Once the PTG to be searched is located using the hash function, which PTE to use needs to be sorted out. The match operation must include both the virtual address bits and the asid, address space identifier, as part of the test for a match. It is possible that the same virtual address is used by two or more different address spaces, which is why it needs to be in the match.

LOCALITY OF REFERENCE

The page table group may be cached in the system read cache for performance. It is likely that the same PTG group will be used multiple times due to the locality of reference exhibited by running software.

ACCESS RIGHTS

To avoid duplication of data the access rights are stored in another table called the PMT for access rights table. The first time a translation is loaded the access rights are looked-up from the PMT. A bit is set in the TLB entry indicating that the access rights are valid. On subsequent translations the access rights are not looked up, but instead they are read from values cached in the TLB.

QUPLS HIERARCHICAL PAGE TABLE SETUP

OVERVIEW

Qupls 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 POINTER FORMAT – PTP

The PTP occupies only 32-bits. 16384 SPTEs will fit into an 64kB page. A physical address range maximum of 2^{46} bytes of memory may be mapped.

V	11	PPN _{29..0}
---	----	----------------------

PAGE TABLE ENTRY FORMAT – PTE

The PTE format may map up to 2^{33} bytes or 8GB of contiguous memory. The upper address bits for the translation are supplied by bits 17 to 29 of the PTP. The PTE is four bytes in size. 16384 SPTEs will fit into an 64kB page.

V	0	M	A	AVL ₃	CACHE4	SW ₁	URWX ₃	PPN _{16..0}
---	---	---	---	------------------	--------	-----------------	-------------------	----------------------

Field	Size	Purpose
PPN	17	Physical page number
URWX	3	User read-write-execute
SRWX	1	Supervisor write protect
CACHE	4	Cache-ability bits
AVL	3	OS software usage
A	1	1=accessed/used
M	1	1=modified
T	1	1 = PTP, 0 = PTE
V	1	1 if entry is valid, otherwise 0

TLB – TRANSLATION LOOKASIDE BUFFER

OVERVIEW

A simple page map is limited in the translations it can perform because of its size. The solution to allowing more memory to be mapped is to use main memory to store the translations tables.

However, if every memory access required two or three additional accesses to map the address to a final target access, memory access would be quite slow, slowed down by a factor of two or three, possibly more. To improve performance, the memory mapping translations are stored in another unit called the TLB standing for Translation Lookaside Buffer. This is sometimes also called an address translation cache ATC. The TLB offers a means of address virtualization and memory protection. A TLB works by caching address mappings between a real physical address and a virtual address used by software. The TLB deals with memory organized as pages. Typically, software manages a paging table whose entries are loaded into the TLB as translations are required.

The TLB is a cache specialized for address translations. Qupls's TLB contains 128 two-way associative entries. On a TLB miss the page table is searched for a translation by a hardware-based page table walker and if found the translation is stored in one of the ways of the TLB. The way selected is determined randomly.

SIZE / ORGANIZATION

The TLB has 128 entries per set.

TLB ENTRIES - TLBE

Closely related to page table entries are translation look-aside buffer, TLB, entries. TLB entries have additional fields to match against the virtual address. The count field is used to invalidate the entire TLB. Note that the least significant 7-bits of the virtual address are not stored as these bits are used as an index for the TLB entry.

Count ₆	LRU ₃
--------------------	------------------

V	LVL/BC ₅	RGN ₃	M	A	T	S	G	SW ₂	CACHE ₄	MRWX ₃	HRWX ₃	SRWX ₃	URWX ₃
PPN _{31..0}													
PPN _{63..32}													

VPN _{38..7}		
VPN _{70..39}		
ASID _{15..0}	~3	VPN _{83..71}

SMALL TLB ENTRIES - TLBE

The small TLB is used for the test system which contains only 512MB of physical RAM to conserve hardware resources. The address ranges are more limited, 40-bits for the physical address and 70-bits for the virtual address.

Count ₆	LRU ₃
--------------------	------------------

V	LVL/BC ₅	RGN ₃	M	A	T	S	G	SW ₂	CACHE ₄	MRWX ₃	HRWX ₃	SRWX ₃	URWX ₃
~8	PPN _{23..0}												

VPN _{38..7}	
ASID _{15..0}	
PS	VPN _{53..39}

WHAT IS TRANSLATED?

The TLB processes addresses including both instruction and data addresses for all modes of operation. It is known as a *unified* TLB.

PAGE SIZE

Because the TLB caches address translations it can get away with a much smaller page size than the page map can for a larger memory system. 4kB is a common size for many systems. There are some indications in contemporary documentation that a larger page size would be better. In this case the TLB uses 64kB. For a 512MB system (the size of the memory in the test system) there are 8192 64kB pages.

WAYS

The TLB is two-way associative.

MANAGEMENT

The TLB unit is updated by a hardware page table walker.

?RWX₃

If RWX₃ attributes are specified non-zero, then they will override the attributes coming from the region table. Otherwise RWX attributes are determined by the region table.

CACHE₄

The cache₄ field is combined with the cache attributes specified in the region table. The region table takes precedence; however, if the cache₄ field indicates non-cache-ability then the data will not be cached.

TLB ENTRY REPLACEMENT POLICIES

The TLB uses random replacement. Random replacement chooses a way to replace at random.

FLUSHING THE TLB

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

RESET

On a reset the TLB is preloaded with translations that allow access to the system ROM.

GLOBAL BIT

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

PTW - PAGE TABLE WALKER

The page table walker is a CPU device used to update the TLB. Whenever a TLB miss occurs the page table walker is triggered. The page table walker walks the page tables to find the translation. Once found the TLB is updated with the translation. If a translation cannot be found then a page fault occurs.

The page table walker manages several variables associated with memory management. These include the page table base register, PT_BASE, page fault address and ASID. These registers are available to software using load and store instructions.

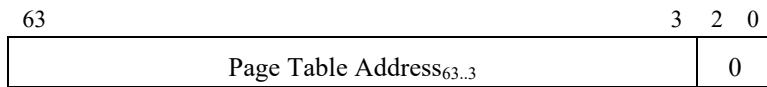
For a page fault the miss address and ASID are stored in a register in the page-table-walker. The PTW also contains the PT_BASE(page table base register) which is used to locate the page table.

The page table walker is a device located in the CPU and has a device configuration block associated with it. The default address of the device is \$FF...FF40000.

Register	Name	Description
\$FF00	PF_ADDR	Page fault address
\$FF10	PF_ASID	Page fault asid
\$FF20	PT_BASE	Page table base register
\$FF30	PT_ATTR	Page table attributes

PAGE TABLE BASE REGISTER

The page table base register locates the page table in memory. Address bits 3 to 63 are specified. The page table must be octa-byte aligned. Normally the root page table will occupy 64kB of memory and be 64kB aligned. However, for smaller apps it may be desirable to share the memory page the page table is located in with multiple applications.



Default Reset Value = 0xFFFFFFFFFFF80000

PAGE TABLE ATTRIBUTES REGISTER

The attributes register contains attributes of the page table.

63	26	25	24	12	11	8	7 6	5	4	3	2 1	0
~ ₃₈	~	Root Page Table Limit _{12..0}	Levels	AL ₂	~ ₂	S	~	Type				

Type: 0 = inverted page table, 1 = hierarchical page table

S: 1=software managed TLB miss, 0 = hardware table walking, 0 is the only currently supported option.

AL₂: TLB entry replacement algorithm, 0=fixed,1=LRU,2=random,3=reserved, 2 is the only currently supported option.

Levels are ignored for the inverted page table. For a normal page table gives the top entry level.

Root Page Table Limit specifies the number of entries in the root page table. A maximum of 8192 entries is supported.

DEFAULT RESET VALUE = 0X1FFF081

63	26	25	24	12	11	8	7 6	5	4	3	2 1	0
~ ₃₈	~	1FFFh		0	2		~ ₂	0	~	1		

CARD TABLE

OVERVIEW

Also present in the memory system is the Card table. The card table is a telescopic memory which reflects with increasing detail where in the memory system a pointer write has occurred. This is for the benefit of garbage collection systems. Card table is updated using a write barrier when a pointer value is stored to memory, or it may be updated automatically using the STPTR instruction.

ORGANIZATION

At the lowest level memory is divided into 256-byte card memory pages. Each card has a single byte recording whether a pointer store has taken place in the corresponding memory area. To cover a 512MB memory system 2MB card memory is required at the outermost layer. A byte is used rather than a bit to allow byte store operations to update the table directly without having to resort to multiple instructions to perform a bit-field update.

To improve the performance of scanning a hardware card table, HCT, is present which divides memory at an upper level into 8192-byte pages. The hardware card table indicates if a pointer store operation has taken place in one of the 8192-byte pages. It is then necessary to scan only cards representing the 8192-byte page rather than having to scan the entire 2MB card table. Note that this memory is organized as 2048 32-bit words. Allowing 32-bits at a time to be tested.

To further improve performance a master card table, MCT, is present which divides memory at the uppermost layer into 16-MB pages.

Layer	Resolving Power	
0	2 MB	256B pages
1	64k bits	8kB pages
2	32 bits	16 MB pages

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

LOCATION

The card memory location is stored in the region table. A card table may be setup for each region of memory.

OPERATION

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

bit or byte is set in each layer of the card memory system corresponding to the memory location of the pointer store.

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

SAMPLE WRITE BARRIER

```
; Milli-code routine for garbage collect write barrier.  
; This sequence is short enough to be used in-line.  
; Three level card memory.  
; a2 is a register pointing to the card table.  
; STPTR will cause an update of the master card table, and hardware card table.  
;  
GCWriteBarrier:
```

STPTR	a0,[a1]	; store the pointer value to memory at a1
LSR	t0,a1,#8	; compute card address
STB	r0,[a2+t0]	; clear byte in card memory

INSTRUCTION SET

OVERVIEW

Qupls4 is a fixed length instruction set with 48-bit instructions. There are several different classes of instructions including arithmetic, memory operate, branch, floating-point and others.

Fixed length instructions are easier to deal with and may require less hardware. There are for instance only four decoders in the front-end of the CPU. If the instruction set were variable length more decoders might be required.

There are both scalar and vector forms of instructions.

While there may seem to be an enormous number of instructions, there are significantly fewer that the CPU processes as some of the instructions are processed as the same instruction in the functional unit. For instance, the vector ADD instruction is processed internally by the CPU as a scalar ADD on multiple registers. From the CPU's perspective there is a single ADD but for the purposes of programming there are both scalar and vector ADDs. This effectively doubles the number of instructions documented.

CODE ALIGNMENT

Program code may be relocated at any wyde (16-bit boundary) address. However, within a subroutine code should be contiguous. The instruction pointer advances six bytes per instruction. It is always aligned on an even byte address.

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 the lowest seven bits of the instruction.

For some instructions the precision of the operation is encoded in the two LSBs of the opcode.

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
Func ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						R3

REGISTER SPECS

Register specifications are made with a six bit register field. For scalar instructions the register spec identifies the register directly (0 to 31). The most significant bit of the spec indicates to negate or complement the register value.

THE REGISTER TYPE FIELD - V

For vector instructions the low order five bits indicate the vector register, the most significant bit of the spec indicates to negate or complement the register value. The V₄ field indicates which registers are vector registers or scalar registers.

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
Func ₇	Ms ₃	Op ₃	V ₄		Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		Opcode ₇		R3			

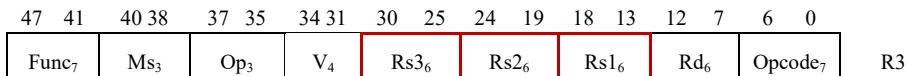
DESTINATION REGISTER SPEC

Most instructions have a destination register. The register spec for the destination register is always in the same position, bits 7 to 12 of an instruction. For some instructions, such as stores, the destination register field acts as a source register. Some instructions have two destination registers.

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
Func ₇	Ms ₃	Op ₃	V ₄		Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		Opcode ₇		R3			

SOURCE REGISTER SPEC

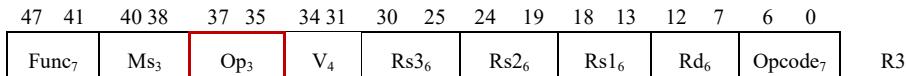
Most instructions have at least one source register. There may be as many as four source register specs. The register spec for source registers is always in the same position.



SECONDARY OPCODE

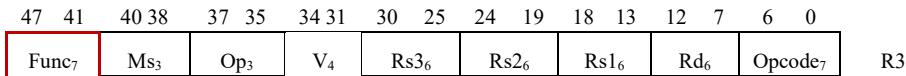
Register-register operate instructions often have slightly different forms depending on a secondary opcode. The secondary opcode generally controls the operation between the result from the first two source register and the third source register. For some instructions this field is ignored.

One of the second operations performed is typically masking the result under the guidance of the Rs3 register.



PRIMARY FUNCTION CODE

For register-to-register operate instructions the primary function code is in the most significant seven bits of the instruction, bits 41 to 47. This function code typically controls the operation between registers Rs1 and Rs2; sometimes Rs3 is also included.



PRECISION

The CPU supports multiple precisions for most operations. The precision selected is controlled by the opcode. A register may be treated as a 64-bit value, 32-bit value, 16-bit value, or 8-bit value. The same operation is applied for each value. A pair of registers may be treated as a 128-bit value for some instructions.

Opcode ₇	Values	SIMD
104	8-bit	1 x 8-bit
105	16-bit	1 x 16-bit
106	32-bit	1 x 32-bit
107	64-bit	1 x 64-bit
112	8-bit	8 x 8-bit
113	16-bit	4 x 16-bit

114	32-bit	2 x 32-bit
115	64-bit	1 x 64-bit

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 inline following the instruction using the constants. Most instructions follow the same template for constants.

The Ms field combined with the V field of an instruction indicates which register specifications represent six-bit constants or cache-line constants.

Seven-bit register spec (when Ms bit is set)		
V		
0	Six-bit embedded constant	
1	Size ₂	Zone location ₄

Cache-line constants are located in constant zones following the instruction. Please see the CZ instructions for additional information. There are 10 possible locations where a constant may begin which is specified by the zone location field. The size of the constant may be 16, 32, 48 or 64 bits.

Size ₂	
0	16-bits
1	32-bits
2	48-bits
3	64-bits

CLOCK CYCLES

Clock cycles given for instructions are approximate and represent a relative relationship between instructions. Usually, the latency is given. An instruction with a given clock cycle count of two will take approximately twice as long to execute as an instruction with a given clock cycle count of one. Generally, a clock cycle count of one means the instruction requires only a single clock to execute. However, the effective execution time of the instruction may be less if multiple execution units can execute the instruction or if execution of the instruction may be overlapped with execution of other instructions.

EXECUTION UNITS

Most instructions execute on a particular type of execution unit. For instance, the ADD instruction is executed on an ALU while the LDB instruction is executed on a memory unit. The execution unit for the instruction is listed in the instruction's description. The execution of some instructions is supported on only a single execution unit even if multiple execution units of the same type are available. For instance, infrequently used instructions like bitfield manipulations or sub-word shifts and rotates are supported on

only a single ALU. Instructions that execute on two different execution units may execute at the same time. This is important for instruction scheduling.

Some instructions can execute on different types of execution units. For instance, FABS may execute on the FPU and on the SAU type units. For many common instructions there are four execution units available.

INSTRUCTION DESCRIPTIONS

BASIC INSTRUCTION FORMATS

Although there is a large number of instructions, there are relatively few instruction formats. There are some additional formats not shown below. Please see the instruction descriptions for specific formats.

The most common instruction format is the register-register operate instruction format

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
Func ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						R3

Func ₇	Ms ₃	Rm ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇	F3 / DF3
-------------------	-----------------	-----------------	----------------	------------------	------------------	------------------	-----------------	---------------------	----------

P ₂	Immediate ₂₇				Rs1 ₆	Rd ₆	Opcode ₇	Imm
----------------	-------------------------	--	--	--	------------------	-----------------	---------------------	-----

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms	Disp _{18...0}			Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇	LS – load / store			

47	45	44	43	34	33 31	30	25	24	19	18	13	12	7	6	0
Sc ₃	Ms	Disp _{9...0}			Dt ₃	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇	VLS – vector load / store				

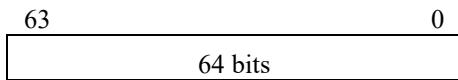
4746	45	44	31	30	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}			Rs3 ₇	Rs2 ₆	Rs1 ₆	Ms ₂	Cnd ₄	Opcode ₇	BR – conditional branch			

4746	45	44	31	30	25	24	19	18	13	1211	10	7	6	0
N ₂	1	~ ₁₃		Rs3 ₇	Rs2 ₆	Rs1 ₆	Ms ₂	Cnd ₄	Opcode ₇	BRR – branch to register				

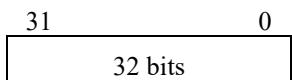
ARITHMETIC OPERATIONS

REPRESENTATIONS

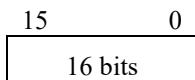
INT



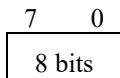
SHORT INT



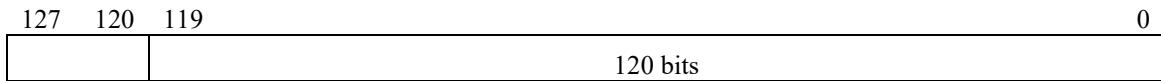
WYDE



BYTE



DECIMAL



Decimal integers use densely packed decimal format which provide 36 digits of precision.

ARITHMETIC OPERATIONS

Arithmetic operations include addition, subtraction, multiplication and division. These are available with the ADD, SUB, CMP, MUL, and DIV instructions. There are several variations of the instructions to deal with signed and unsigned values. Multiply may either multiply two values and add a third returning the low order bits, or return the entire product, referred to as a widening instruction. Divide may return both the quotient and the remainder with one instruction. The format of the typical immediate mode instruction is shown below:

ADD.sz Rd, Rs1, Imm₂₅

Instruction Format: RI

47	46	45	19	18	13	12	7	6	0
Prc ₂		Immediate ₂₇		Rs1 ₆	Rd ₆		4 ₇		

PRECISION

Four different precisions are supported encoded by the Prc₂ field of an instruction. The precision of an operation may be specified with an instruction qualifier following the mnemonic as in ADD.T to add tetras together. The assembler assumes an octa-byte operation if the size is not specified.

Prc ₂	Register treated as: Bits
0	8
1	16
2	32
3	64

ABS – ABSOLUTE VALUE

Description:

This instruction computes the absolute value of the sum of contents of the source operands and places the result in Rd. Unused source operands should be set to zero.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
13 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel

OP ₃		Mnemonic
3	Rd = ABS((Rs1 + Rs2) + Rs3)	ABS SUM

Operation:

If Source < 0

Rd = -Source

else

Rd = Source

Execution Units: SAU #0**Read Ports: Rs1, Rs2, Rs3**

Clock Cycles: 1

Exceptions: none

Notes:

ADC – ADD WITH CARRY

Description:

Add three registers, Rs1, Rs2 and Rs3 and place the result in the destination register Rd. Place the carry out from the operation in destination register Rd2.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
Rd2 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	12 ₇						

Operation:

$$\{Rd2, Rd\} = Rs1 + Rs2 + Rs3$$

Clock Cycles: 1**Execution Units: SAU #0****Read Ports: Rs1, Rs2, Rs3, Rs4****Write Ports: Rd1, Rd2****Exceptions: none****Notes:****Example: 256-bit add**

```
ADC a3, a1, a2, 0, cy0      ; a3 = sum add bits 0 to 63, carry goes into cy0
ADC b3, b1, b2, cy0, cy1    ; b3 = sum add bits 64 to 127, carry goes into cy1
ADC c3, c1, c2, cy1, cy2    ; c3 = sum add bits 128 to 191, carry goes into cy2
ADC d3, d1, d2, cy2, cy3    ; d3 = sum add bits 192 to 255, carry goes into cy3
```

ADD – ADD REGISTER-REGISTER

Description:

Add two registers Rs1 and Rs2, perform a second operation with the result and a third register Rs3 and place the result in the destination register Rd. All register values are integers.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
4 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

Operation:

Opcode ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa
112	Byte vector
113	Wyde vector
114	Tetra vector
115	Octa vector
116	vector – element size determined from VELSZ

OP ₃		Mnemonic
0	Rd = (Rs1 + Rs2) & Rs3	ADD AND
1	Rd = (Rs1 + Rs2) Rs3	ADD OR
2	Rd = (Rs1 + Rs2) ^ Rs3	ADD EOR
3	Rd = (Rs1 + Rs2) + Rs3	ADD ADD
4	Rd = (Rs1 + Rs2) << Rs3	ADD ASL
5	reserved	
6	Rd = Rs3 ? (Rs1 + Rs2) : Rd	ADDM
7	Rd = Rs3 ? (Rs1 + Rs2) : Rs2	CADD

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Read Ports: Rs1, Rs2, Rs3

Exceptions: none

Notes:

ADD.XP – ADD PARALLEL REGISTER-REGISTER

Description:

Add two vector registers, perform a second operation with the result and a third vector register and place the result in the destination vector register. All register values are integers. For parallel operations the register specifies the first register of a group of registers.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
4 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆							Opcode ₇

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

Operation:

Opc ₇	Precision
112	Byte parallel – 32 x 8 bit
113	Wyde parallel – 16x16 bit
114	Tetra parallel – 8x32 bit
115	Octa parallel – 4x64 bit

OP ₃		Mnemonic
0	Rd = (Rs1 + Rs2) & Rs3	ADD_AND
1	Rd = (Rs1 + Rs2) Rs3	ADD_OR
2	Rd = (Rs1 + Rs2) ^ Rs3	ADD_EOR
3	Rd = (Rs1 + Rs2) + Rs3	ADD_ADD
4	Rd = (Rs1 + Rs2) << Rs3	ADD_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 + Rs2) : Rd	MADD
7	Rd = Rs3 ? (Rs1 + Rs2) : Rs2	CADD

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Read Ports: Rs1, Rs2, Rs3

Exceptions: none

Notes:

ADDI - ADD IMMEDIATE

Description:

Add a register Rs1 and immediate value and place the sum in the destination register Rd. The immediate is sign extended to the machine width.

Instruction Format: RI

47	46	45	19	18	13	12	7	6	0
Prc ₂		Immediate ₂₇		Rs1 ₆	Rd ₆		4 ₇		

Clock Cycles: 1**Execution Units:** All SAUs, all FPUs**Operation:**

$$Rd = Rs1 + \text{immediate}$$

Read Ports: Rs1**Exceptions:****Notes:**

ADDJO – ADD, JUMP ON OVERFLOW

Description:

Add two registers, Rs1 and Rs2, and place the sum in the destination register, Rd. All register values are signed integers. If the result overflows, then a jump is made to the address contained in register Rs3.

Instruction Format: R3

47	41	40 39	38	35	34	33	28	27	26	21	20	19	14	13	12	7	6	0
15 ₇	2 ₂	Op ₄	N3	Rs3 ₆	N2	Rs2 ₆	N1	Rs1 ₆	Nd	Rd ₆	Opc ₇							

Operation:

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa

OP ₄		Mnemonic
3	Rd = ± (±Rs1 ± Rs2)	ADDJO
1 to 15	Reserved	

Clock Cycles: 1 (no jump)**Execution Units:** All Integer ALUs, all FPUs**Exceptions:** none**Notes:**

ADDIPI - ADD IMMEDIATE TO INSTRUCTION POINTER

Description:

Add register Rs1 and an immediate value to the instruction pointer and place the result in a destination register Rd. The immediate is sign extended to the machine width. This instruction may be used in the formation of instruction pointer relative addresses.

Instruction Format: RI

47	46	45	19s	18	13	12	7	6	0
3 ₂		Immediate ₂₇		Rs1 ₆	Rd ₆	15 ₇			

Clock Cycles: 1**Execution Units:** All SAU's**Operation:**

$$Rd = Rs1 + IP + \text{immediate}$$

Exceptions:**Notes:**

BYTENDX – CHARACTER INDEX

Description:

This instruction searches Rs1, which is treated as an array of characters, for a character value specified by Rs2 and places the index of the character into the destination register Rd. If the character is not found -1 is placed in the destination register. A common use would be to search for a null character. The index result may vary from -1 to +7. The index of the first found byte is returned (closest to zero). The result is -1 if the character could not be found.

A masking operation may be performed on the Rs1 operand to allow searches for ranges of characters according to an immediate constant. For instance, a constant could be set to 0x78 (in Rs3) and the mask ‘anded’ with Rs1 to search for any ascii control character.

Supported Operand Sizes: .b, .w, .t

Instruction Format: R3 (byte)

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
37 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	107 ₇						

Op3	Mask Operation
0	Rs1
1	Rs1 & Rs3
2	Rs1 Rs3
3	Rs1 ^ Rs3

Operation:

$$Rd = \text{Index of } (Rs2 \text{ in } Rs1)$$

Execution Units: All Integer ALU's

Read Ports: Rs1, Rs2, Rs3

Exceptions: none

Notes:

CHARNDX – CHARACTER INDEX

Description:

This instruction searches Rs1, which is treated as an array of characters, for a character value specified by Rs2 and places the index of the character into the destination register Rd. If the character is not found -1 is placed in the destination register. A common use would be to search for a null character. The index result may vary from -1 to +7. The index of the first found byte is returned (closest to zero). The result is -1 if the character could not be found.

A masking operation may be performed on the Rs1 operand to allow searches for ranges of characters according to an immediate constant. For instance, a constant could be set to 0x78 (in Rs3) and the mask ‘anded’ with Rs1 to search for any ascii control character.

Supported Operand Sizes: .b, .w, .t

Instruction Format: R3 (byte)

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
37 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	107 ₇						

Op2	Mask Operation
0	Rs1
1	Rs1 & Rs3
2	Rs1 Rs3
3	Rs1 ^ Rs3

Operation:

$$Rd = \text{Index of } (Rs2 \text{ in } Rs1)$$

Execution Units: All Integer ALU's

Read Ports: Rs1, Rs2, Rs3

Exceptions: none

Notes:

CHK – CHECK REGISTER AGAINST BOUNDS

Description:

A register, Rs1, is compared to two values Rs2 and Rs3. If the register is outside of the bounds defined by Rs2 and Rs3 then a check exception will occur.

Comparisons may be signed or unsigned.

Instruction Format: R2

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
0 ₇	Ms ₃	~ ₃	V ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆	Func ₆		47 ₇				

Func₆ exception when not

0	Rs1 >= Rs2 and Rs1 < Rs3	
1	Rs1 >= Rs2 and Rs1 <= Rs3	
2	Rs1 > Rs2 and Rs1 < Rs3	
3	Rs1 > Rs2 and Rs1 <= Rs3	
4	Not (Rs1 >= Rs2 and Rs1 < Rs3)	
5	Not (Rs1 >= Rs2 and Rs1 <= Rs3)	
6	Not (Rs1 > Rs2 and Rs1 < Rs3)	
7	Not (Rs1 > Rs2 and Rs1 <= Rs3)	
8 to 15	Unsigned comparisons as above	

16	Rs1 >= CPL	CHKCPL – code privilege level
17	Rs1 <= CPL	CHKDPL – data privilege level
18	Rs1 == SC	Stack canary check

Operation:

IF check failed

CAUSE = FLT_CHK

PUSH SR onto internal stack

PUSH IP plus 6 onto internal stack

IP = Rs4 ? Rs4 : vector at (tvec[operating mode])

Clock Cycles: 1

Execution Units: SAU #0

Read Ports: Rs1, Rs2, Rs3

Exceptions: bounds check

Notes:

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

CHKCPL – CHECK CODE PRIVILEGE LEVEL

Description:

A register, Rs1, is compared against the CPUs current privilege level. If the register is below the CPL then an exception will occur.

Instruction Format: R2

47	47	41	40 38	37	35	34	33	28	27	26	21	20	19	14	13	12	7	6	0
N4	Rs4 ₇	Ms ₃	~ ₃	N3	Rs3 ₆	N2	Rs2 ₆	N1	Rs1 ₆	0	16 ₆	47 ₇							

Clock Cycles: 1

Execution Units: Integer ALU

Exceptions: bounds check

Notes:

CNTXX – COUNT

Description:

CNTLZ: This instruction counts the number of consecutive zero bits beginning at the most significant bit towards the least significant bit.

LOPOS: This instruction returns the bit position of the leading one.

CNTLO: This instruction counts the number of consecutive one bits beginning at the most significant bit towards the least significant bit.

CNTPOP: This instruction counts the number of set bits.

CNTNPOP: This instruction counts the number of clear bits.

CNTTZ: This instruction counts the number of clear bits beginning at the least significant bit towards the most significant bit.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
26 ₇	Ms ₃	~ ₃	V ₄	0 ₆	Op ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Opc ₆	Precision
104	Byte
105	Wyde
106	Tetra
107	octa

Operation:

Op ₆		Mnemonic	Comment
0	Rd = leading zero count (Rs1)	CNTLZ	Count leading zeros
3	Rd = 63 – leading zero count (Rs1)	LOPOS	Leading one position
1	Rd = leading ones count (Rs1)	CNTLO	Count leading ones
2	Rd = population count (Rs1)	CNTPOP	Count population of set bits
34	Rd = population count (Rs1)	CNTNPOP	Count population of clear bits
6	Rd = trailing zeros count (Rs1)	CNTTZ	Count trailing zeros
others	Reserved		

Execution Units: SAU #0

Read Ports: Rs1, Rs2

Clock Cycles: 1

Exceptions: none

Notes:

CPUINFO – GET CPU INFO

Description:

This instruction returns general information about the core. The sum of Rs1 and register Rs2 is used as a table index to determine which row of information to return.

Supported Operand Sizes: N/A

Instruction Formats: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
7 ₇	Ms ₃	~ ₃	V ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	107 ₇							

Clock Cycles: 1

Execution Units: ALU #0 only

Read Ports: Rs1, Rs2

Operation:

$$Rd = \text{Info}([Rs1+Rs2])$$

Exceptions: none

Index	bits		Information Returned
0	0 to 63		The processor core identification number. This field is determined from an external input. It would be hard wired to the number of the core in a multi-core system.
2	0 to 63		Manufacturer name first eight chars “Finitron”
3	0 to 63		Manufacturer name last eight characters
4	0 to 63		CPU class “64BitSS”
5	0 to 63		CPU class
6	0 to 63		CPU Name “Qupls”
7	0 to 63		CPU Name
8	0 to 63		Model Number “M1”
9	0 to 63		Serial Number “1234”
10	0 to 63		Features bitmap
11	0 to 31		Instruction Cache Size (32kB)
11	32 to 63		Data cache size (64kB)
12	0 to 7		Maximum vector length – number of elements
13	0 to 15	VLEN_MAX	Maximum vector length in bits
14	0 to 15	VLENB_MAX	Maximum vector length in bytes (8 * VLEN_MAX)

15	0 to 15	VELSZ_MAX	Maximum vector element size in bits - power of 2 – at least 8 bits, must be <= VLEN_MAX
16	0 to 15	REGSZ	Size of a register in bits

CSR – CONTROL AND SPECIAL REGISTERS OPERATIONS

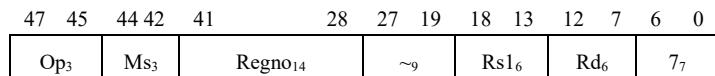
Description:

Perform an operation on a CSR. The previous value of the CSR is placed in the destination register.

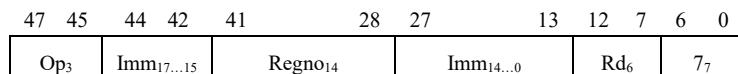
Operation	Op ₃	Mnemonic
Read CSR	0	CSRRD
Write CSR	1	CSRRW
Or to CSR (set bits)	2	CSRRS
And complement to CSR (clear bits)	3	CSRRC
reserved	4	
Write Immediate CSR	5	CSRRW
Or Immediate to CSR	6	CSRRS
And Immediate complement to CSR	7	CSRRC

Supported Operand Sizes: N/A

Instruction Formats: CSR



Instruction Formats: CSRI



Read Ports: Rs1

Notes:

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

DIV – SIGNED DIVISION

Description:

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

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
17 ₇	Ms ₃	Op ₃	V ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇							

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel

Operation:

$$Rd = Rs1 / Rs2$$

Size	Clocks
Octa-byte	34

Execution Units: DIV**Read Ports: Rs1, Rs2****Write Ports: Rd****Exceptions: DBZ****Notes:**

DIVI – SIGNED IMMEDIATE DIVISION

Description:

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

Operation:

$$Rd = Rs1 / Imm$$

Instruction Format: RI

47	46	45		19	18	13	12	7	6	0
Pr _{c2}			Immediate ₂₇		Rs1 ₆	Rd ₆		13 ₇		

Execution Units: DIV**Read Ports: Rs1****Exceptions: none****Notes:**

DIVU – UNSIGNED DIVISION

Description:

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

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
20 ₇	Ms ₃	Op ₃	V ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆	Rd ₆		Opcode ₇				

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa

Operation:

$$Rd = Rs1 / Rs2$$

Size	Clocks
Octa-byte	34

Execution Units: ALU #0 Only**Read Ports:** Rs1, Rs2**Write Ports:** Rd**Exceptions:** none**Notes:**

DIVUI – UNSIGNED IMMEDIATE DIVISION

Description:

Divide source dividend operand Rs1 by divisor operand and place the quotient in the destination register Rd. All registers are integer registers. Arithmetic is unsigned two's-complement values.

Operation:

$$Rd = Rs1 / Imm$$

Instruction Format: RI

47	46	45		19	18	13	12	7	6	0
Prc ₂			Immediate ₂₇		Rs1 ₆	Rd ₆		21 ₇		

Execution Units: DIV**Read Ports: Rs1****Write Ports: Rd****Exceptions: none****Notes:**

LOADA – LOAD ADDRESS

Description:

This instruction computes the scaled indexed virtual address and places it in the destination register. It matches the format used by the load and store instructions.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18...0}	Rs2 ₆	Rs1 ₆	Rd ₆	20 ₇					LS – load / store

If Ms = 0 then displacement is Disp_{18...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{18...0}
1	0	reserved
	1	reserved
	2	32-bit cache-line constant
	3	64-bit cache-line constant

Clock Cycles: 1**Execution Units:** All SAU's**Read Ports:** Rs1, Rs2**Write Ports:** Rd**Operation:**

$$Rd = Rs1 + Rs2 * Scale + displacement$$

Exceptions:**Notes:**

MAJ – MAJORITY LOGIC

Description:

Determines the bitwise majority of three values in registers Rs1, Rs2 and Rs3 and places the result in the destination register Rd.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
1 ₇	Ms ₃	7 ₃	V ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆		Rd ₆		Opc ₇			

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa

Execution Units: ALU #0 only

Operation:

$$Rd = (Rs1 \& Rs2) | (Rs1 \& Rs3) | (Rs2 \& Rs3)$$

PTRDIF – DIFFERENCE BETWEEN POINTERS

Asm: PTRDIF Rd, Rs1, Rs2, Rs3

Description:

Subtract two values then shift the result right. Both operands must be in a register. The right shift is provided to accommodate common object sizes. It may still be necessary to perform a divide operation after the PTRDIF to obtain an index into odd sized or large objects. Sc may vary from zero to fifteen.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
32 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Opcode ₇	Precision
104	
105	
106	Tetra
107	octa

Op3		Operation	Comment
0	PTRDIF	Rd = abs(Rs1 - Rs2) >> Rs3 _[3:0]	
1	AVG	Rd = (Rs1 + Rs2) >> Rs3 _[3:0] , trunc	Arithmetic shift right
2	AVG	Rd = (Rs1 + Rs2) >> Rs3 _[3:0] , round up	Arithmetic shift right
3		Reserved	

Operation:

$$Rd = \text{Abs}(Rs1 - Rs2) \gg Rs3_{[3:0]}$$

Clock Cycles: 1

Execution Units: SAU #0

Read Ports: Rs1, Rs2, Rs3

Write Ports: Rd

Exceptions:

None

REM – SIGNED REMAINDER

Description:

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

Operation:

$$Rd = Rs1 \% Rs2$$

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
25 ₇	Ms ₃	Op ₃	V ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇							

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa

Size	Clocks
Octa-byte	34

Execution Units: DIV**Read Ports:** Rs1, Rs2**Write Ports:** Rd**Exceptions:** DBZ**Notes:**

REMU – UNSIGNED REMAINDER

Description:

Divide source dividend operand Rs1 by divisor operand Rs2 and place the remainder in the destination register Rd. All registers are integer registers. Arithmetic is unsigned two's-complement values.

Operation:

$$Rd = Rs1 \% Rs2$$

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
28 ₇	Ms ₃	Op ₃	VN ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆		Rd ₆		Opcode ₇			

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa

Size	Clocks
Octa-byte	34

Execution Units: DIV**Read Ports:** Rs1, Rs2**Write Ports:** Rd, Rs3**Exceptions:** DBZ**Notes:**

REVBIT – REVERSE BIT ORDER

Description:

This instruction reverses the order of bits in Rs2 and stores the result in Rd.

Instruction Format: R1

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
26 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	5 ₆	Rs1 ₆	Rd ₆	Opc ₇							

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa

Operation:

Execution Units: SAU #0

Read Ports: Rs1

Write Ports: Rd

Clock Cycles: 1

Exceptions: none

Notes:

SATADD – SATURATING ADD REGISTER-REGISTER

Description:

Add two registers Rs1 and Rs2 with signed saturation, perform a second operation with the result and a third register Rs3 and place the result in the destination register Rd. All register values are integers. If the result overflows the destination register is set to the maximum signed integer value.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
12 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆							Opcode ₇

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

Operation:

Opcode ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa
112	Byte vector
113	Wyde vector
114	Tetra vector
115	Octa vector
116	vector – element size determined from VELSZ

OP ₃		Mnemonic
0	Rd = (Rs1 + Rs2) & Rs3	SATADD_AND
1	Rd = (Rs1 + Rs2) Rs3	SATADD_OR
2	Rd = (Rs1 + Rs2) ^ Rs3	SATADD_EOR
3	Rd = (Rs1 + Rs2) + Rs3	SATADD_ADD
4	Rd = (Rs1 + Rs2) << Rs3	SATADD_ASL
5	reserved	
6	Rd = Rs3 ? (Rs1 + Rs2) : Rd	SATADDM
7	Rd = Rs3 ? (Rs1 + Rs2) : Rs2	CADD

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Read Ports: Rs1, Rs2, Rs3

Exceptions: none

Notes:

SBC – SUBTRACT WITH CARRY

Description:

Subtract three registers, Rs1, Rs2 and Rs3 and place the result in the destination register Rd1. Place the carry out from the operation in destination register Rd2.

Instruction Format: R3

47	41	40 38	37	35	34	28	27	21	20	14	13	7	6	0
Rd2 ₇	Ms ₃	l ₃	Rs3 ₇	Rs2 ₇	Rs1 ₇	Rd1 ₇	12 ₇							

Operation:

$$\{Rd2, Rd1\} = Rs1 - Rs2 - Rs3$$

Clock Cycles: 1**Execution Units: SAU #0****Read Ports: Rs1, Rs2, Rs3****Write Ports: Rd1, Rd2****Exceptions: none****Notes:****Example: 256-bit subtract**

```
SBC a3, a1, a2, 0, cy0      ; a3 = sub bits 0 to 63, carry goes into cy0
SBC b3, b1, b2, cy0, cy1   ; b3 = sub bits 64 to 127, carry goes into cy1
SBC c3, c1, c2, cy1, cy2   ; c3 = sub bits 128 to 191, carry goes into cy2
SBC d3, d1, d2, cy2, cy3   ; d3 = sub bits 192 to 255, carry goes into cy3
```

SQRT – SQUARE ROOT

Description:

This instruction computes the integer square root of the contents of the source operand Rs1 and places the result in Rd.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
26 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	4 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

sOpc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa

Operation:

$$Rd = \text{SQRT}(Rs1)$$

Execution Units: DIV**Read Ports: Rs1****Write Ports: Rd****Clock Cycles: 72****Exceptions: none****Notes:**

SUB – SUBTRACT REGISTER-REGISTER

Description:

Subtract two registers perform a second operation on the intermediate result with a third register then place the result in the destination register. All registers are integer registers.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
5 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Opc ₇	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa

Operation: R3

OP ₃		Mnemonic
0	Rd = (Rs1 - Rs2) & Rs3	SUB_AND
1	Rd = (Rs1 - Rs2) Rs3	SUB_OR
2	Rd = (Rs1 - Rs2) ^ Rs3	SUB_EOR
3	Rd = (Rs1 - Rs2) + Rs3	SUB_ADD
4	Rd = (Rs1 - Rs2) << Rs3	SUB_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 - Rs2) : Rd	MSUB
7	Rd = Rs1 ? (Rs2 - Rs3) : Rs2	CSUB

$$Rd = Rs1 - Rs2 - Rs3$$

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Read Ports: Rs1, Rs2, Rs3

Write Ports: Rd

Exceptions: none

Notes:

SUBFI – SUBTRACT FROM IMMEDIATE

Description:

Subtract a register Rs1 from an immediate value and place the difference in the destination register Rd. The immediate is sign extended to the machine width.

Instruction Format: RI

47	46	45	19	18	13	12	7	6	0
Prc ₂		Immediate ₂₇		Rs1 ₆	Rd ₆		5 ₇		

Clock Cycles: 1**Execution Units:** All SAUs, all FPUs**Read Ports:** Rs1**Write Ports:** Rd**Operation:**

$$Rd = \text{immediate} - Rs1$$

Exceptions:**Notes:**

TETRANDX – CHARACTER INDEX

Description:

This instruction searches Rs1, which is treated as an array of characters, for a character value specified by Rs2 and places the index of the character into the destination register Rd. If the character is not found -1 is placed in the destination register. A common use would be to search for a null character. The index result may vary from -1 to +1. The index of the first found tetra is returned (closest to zero). The result is -1 if the character could not be found.

A masking operation may be performed on the Rs1 operand to allow searches for ranges of characters according to a constant in Rs3. For instance, the constant could be set to 0x1F8 and the mask ‘anded’ with Rs1 to search for any ascii control character.

Supported Operand Sizes: .b, .w, .t

Instruction Format: R3 (tetra)

47	41	40	38	37	35	34	33	28	27	26	21	20	19	14	13	12	7	6	0
39 ₇	Op ₃	~ ₃	N3	Rs3 ₆	N2	Rs2 ₆	N1	Rs1 ₆	Nd	Rd ₆	107 ₇								

Op3	Mask Operation
0	Rs1
1	Rs1 & Rs3
2	Rs1 Rs3
3	Rs1 ^ Rs3

Operation:

$$Rd = \text{Index of } (Rs1 \text{ in } Rs1)$$

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

WYDENDX – CHARACTER INDEX

Description:

This instruction searches Rs1, which is treated as an array of characters, for a character value specified by Rs2 and places the index of the character into the destination register Rd. If the character is not found -1 is placed in the destination register. A common use would be to search for a null character. The index result may vary from -1 to +3. The index of the first found wyde is returned (closest to zero). The result is -1 if the character could not be found.

A masking operation may be performed on the Rs1 operand to allow searches for ranges of characters according to a constant in Rs3. For instance, the constant could be set to 0xF8 and the mask ‘anded’ with Rs1 to search for any ascii control character.

Supported Operand Sizes: .b, .w, .t

Instruction Format: R3 (wyde)

47	41	40	38	37	35	34	33	28	27	26	21	20	19	14	13	12	7	6	0
38 ₇	Op ₃	~ ₃	N3	Rs3 ₆	N2	Rs2 ₆	N1	Rs1 ₆	Nd	Rd ₆	107 ₇								

Op3	Mask Operation
0	Rs1
1	Rs1 & Rs3
2	Rs1 Rs3
3	Rs1 ^ Rs3

Operation:

$$Rd = \text{Index of } (Rs2 \text{ in } Rs1)$$

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Vector Arithmetic Operations

VECTOR ARITHMETIC OPERATIONS

VABS – ABSOLUTE VALUE

Description:

This instruction computes the absolute value of the source operand Rs2 and places the result in Rd. Only destination elements where the corresponding R3 bit is set will be updated in Rd.

Instruction Format: R1

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
13 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆							Opcode ₇

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

Opc ₇	Precision
116	vector
100	Scalar to vector

OP ₃		Mnemonic
3	Rd = ABS(Rs2)	ABS_SUM

Operation:

If Source < 0

Rd = -Source

else

Rd = Source

Execution Units: SAU #0

Read Ports: Rs1, Rs2, Rs3

Clock Cycles: 1

Exceptions: none

Notes:

VADD – ADD REGISTER-REGISTER

Description:

Add two registers Rs1 and Rs2, perform a second operation with the result and a third register Rs3 and place the result in the destination register Rd. All register values are integers.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
4 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

Operation:

OP ₃		Mnemonic	Comment
0	Rd = (Rs1 + Rs2) & Rs3	VADD_AND	No masking
1	Rd = (Rs1 + Rs2) Rs3	VADD_OR	
2	Rd = (Rs1 + Rs2) ^ Rs3	VADD_EOR	
3	Rd = (Rs1 + Rs2) ± Rs3	VADD_ADD	
4	Rd = (Rs1 + Rs2) << Rs3	VADD_ASL	
5	reserved		
6	Rd = Rs3 ? (Rs1 + Rs2) : Rd	VADDMM	For each element where Rs3 bit is set
7	Rd = Rs3 ? (Rs1 + Rs2) : Rs2	CADD	

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Read Ports: Rs1, Rs2, Rs3

Exceptions: none

Notes:

VCNTXX – COUNT

Description:

The counting operation takes place for each vector element. Only destination elements where the corresponding R3 bit is set will be updated in Rd.

VCNTLZ: This instruction counts the number of consecutive zero bits beginning at the most significant bit towards the least significant bit.

VLOPOS: This instruction returns the bit position of the leading one.

VCNTLO: This instruction counts the number of consecutive one bits beginning at the most significant bit towards the least significant bit.

VCNTPOP: This instruction counts the number of set bits.

VCNTNPOP: This instruction counts the number of clear bits.

VCNTTZ: This instruction counts the number of clear bits beginning at the least significant bit towards the most significant bit.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
26 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Operation:

Op ₇		Mnemonic	Comment
0	Rd = leading zero count (Rs1)	VCNTLZ	Count leading zeros
3	Rd = 63 – leading zero count (Rs1)	VLOPOS	Leading one position
1	Rd = leading ones count (Rs1)	VCNTLO	Count leading ones
2	Rd = population count (Rs1)	VCNTPOP	Count population of set bits
34	Rd = population count (Rs1)	VCNTNPOP	Count population of clear bits
6	Rd = trailing zeros count (Rs1)	VCNTTZ	Count trailing zeros

Execution Units: SAU #0**Read Ports:** Rs1, Rs2**Clock Cycles:** 1**Exceptions:** none**Notes:**

VDIV – SIGNED DIVISION

Description:

Divide source dividend operand Rs1 by divisor operand Rs2 and place the quotient in the destination register Rd. All registers are integer registers. Arithmetic is signed two's-complement values. Only destination elements where the corresponding R3 bit is set will be updated in Rd.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
17 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇							

Opc ₇	Precision
116	Vector / Vector

Operation:

$$Rd = Rs1 / Rs2$$

Size	Clocks
Octa-byte	34 per 64-bits

Execution Units: DIV**Read Ports:** Rs1, Rs2, Rs3**Write Ports:** Rd**Exceptions:** DBZ**Notes:**

VDIVU – UNSIGNED DIVISION

Description:

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

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
20 ₇	Ms ₃	Op ₃	VN ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆	Rd ₆		116 ₇				

Opc ₇	Precision
116	Vector / Vector

Operation:

$$Rd = Rs1 / Rs2$$

Size	Clocks
Octa-byte	34

Execution Units: ALU #0 Only**Read Ports:** Rs1, Rs2**Write Ports:** Rd**Exceptions:** none**Notes:**

VMASK – MASK RESULT

Description:

This is an alternate mnemonic for the VADDM instruction. Copy only the elements from vector Rs1 to vector Rd that are specified by the mask. All register values are integers. For vector operations the register specifies the first register of a group of registers. Only destination elements where the corresponding R3 bit is set will be updated in Rd.

This instruction may be used to mask the result vector for operations that do not support a vector mask.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
4 ₇	Ms ₃	6 ₃	VN ₄	Rs3 ₆	0 ₆	Rs1 ₆	Rd ₆	116 ₇						

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

Operation:

OP ₃		Mnemonic	Comment
6	Rd = Rs3 ? Rs1 : Rd	VMASK	For each element where Rs3 bit is set

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Read Ports: Rs1, Rs2, Rs3

Exceptions: none

Notes:

VREM – SIGNED REMAINDER

Description:

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

Operation:

$$Rd = Rs1 \% Rs2$$

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
25 ₇	Ms ₃	6 ₃	V ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆	Rd ₆		Opcode ₇				

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa

Size	Clocks
Octa-byte	34

Execution Units: DIV**Read Ports:** Rs1, Rs2, Rs3**Write Ports:** Rd**Exceptions:** DBZ**Notes:**

VSATADD – SATURATING ADD REGISTER-REGISTER

Description:

Add two registers Rs1 and Rs2 with signed saturation, perform a second operation with the result and a third register Rs3 and place the result in the destination register Rd. All register values are integers. If the result overflows the destination register is set to the maximum signed integer value.

The precision of the operation is determined from the U_VELSZ register.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
12 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

Operation:

OP ₃		Mnemonic
0	Rd = (Rs1 + Rs2) & Rs3	SATADD_AND
1	Rd = (Rs1 + Rs2) Rs3	SATADD_OR
2	Rd = (Rs1 + Rs2) ^ Rs3	SATADD_EOR
3	Rd = (Rs1 + Rs2) + Rs3	SATADD_ADD
4	Rd = (Rs1 + Rs2) << Rs3	SATADD_ASL
5	reserved	
6	Rd = Rs3 ? (Rs1 + Rs2) : Rd	SATADDMM
7	reserved	

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Read Ports: Rs1, Rs2, Rs3

Exceptions: none

Notes:

VSUB – SUBTRACT REGISTER-REGISTER

Description:

Subtract two registers Rs2 from Rs1 and place the result in the destination register Rd. All register values are integers. For vector operations the register specifies the first register of a group of registers. Only destination elements where the corresponding R3 bit is set will be updated in Rd.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
5 ₇	Ms ₃	Op ₃	VN ₄	Rs ₃₆	Rs ₂₆	Rs ₁₆	Rd ₆	116 ₇						

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

Operation:

OP ₃		Mnemonic	Comment
0	Rd = (Rs1 - Rs2) & Rs3	VSUB_AND	No masking
1	Rd = (Rs1 - Rs2) Rs3	VSUB_OR	
2	Rd = (Rs1 - Rs2) ^ Rs3	VSUB_EOR	
3	Rd = (Rs1 - Rs2) ± Rs3	VSUB_ADD	
4	Rd = (Rs1 - Rs2) << Rs3	VSUB_ASL	
5	Reserved		
6	Rd = Rs3 ? (Rs1 - Rs2) : Rd	VSUBM	For each element where Rs3 bit is set
7	Rd = Rs3 ? (Rs1 - Rs2) : Rs2	CSUB	

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Read Ports: Rs1, Rs2, Rs3

Exceptions: none

Notes:

EXAMPLES:

VECTOR ADD

```
; for (i = 0; i < n; i++) { sum[i] = a[i] + b[i]; }
; a1 = pointer to a, a2 = pointer to b, a3 = pointer to sum, a4 = n
; t0 = index
; r76 = mask
    BLE a4,$0,.done                                ; anything to do?
    CSRRW r0, VELSZ, $0x04                          ; select 32-bit element sizes
    CSRRW r0, VLEN, $0x20                            ; select 32 byte vectors (8x4 byte elements)
    SYNC                                              ; ensure the CSR updates are seen
    OR t0, r0, r0, r0                                ; t0 = 0

.again1
    VGMASK r76, a4, $8                             ; generate mask for operation
    VLOAD r64, (a1, t0 * 4), r76                   ; get a
    VLOAD r68, (a2, t0 * 4), r76                   ; get b
    VADDM r72, r64, r68, r76                      ; sum = a + b
    VSTORE r72, (a3, t0 * 4), r76                 ; store sum
    ADD t0, t0, $8                                 ; increment index (8 elements)
    SUB a4, a4, $8                               ; reduce number left by number of elements
    BGT a4, $0, .again1                           ; if there are elements left to process, go back

.done
```

MIXED WIDTHS CODE

```
; a = char, b = int, c = int
; for (i = 0; i < n; i++) { b[i] = (a[i] < 5) ? c[i] : 1; }
; a1 = pointer to a, a2 = pointer to b, a3 = pointer to c, a4 = n
; t0 = index
; r76 = mask

        BLE a4,$0,.done
        CSRRW r0, VELSZ, $0x0801040404
        CSRRW r0, VLEN, $0x4008202020
        SYNC
        OR t0, r0, r0, r0
        VGMASK r77, a4, $8
; anything to do?
; select vector element sizes (in bytes)
; select vector lengths (in bytes)
; ensure the CSR updates are seen
; t0 = 0
; generate mask for first operation

.again1
        VLOAD.C r64, (a1, t0 * 4), r77
        VSLT.B r76, r64, $5, r77
        VMOV r72, $1
        VLOAD r72, (a2, t0 * 4), r76
        VSTORE r68, (a3, t0 * 4), r77
        ADD t0, t0, $8
        SUB a4, a4, $8
        VGMASK r77, a4, $8
        BGT a4, $0, .again1
; a[i] = char from memory
; t0 = a[i] < 5
; set all elements to 1
; copy c[i] from memory
; set b[i] in memory
; increment index (8 elements)
; reduce number left by number of elements
; generate mask for next operation
; if there are elements left to process, go back

.done
```

MEMCPY

```
; a0 = src
; a1 = dest
; a2 = n

        BLE a2, $0, .done
        CSRRW r0, VELSZ, $0x0801040404
        CSRRW r0, VLEN, $0x4040202020
        SYNC
        OR t0, r0, r0, r0
        VGMASK r77, a2, $64
; anything to do?
; select vector element sizes (in bytes)
; select vector lengths (in bytes)
; ensure the CSR updates are seen
; t0 = 0
; generate mask for first operation

.again1
        VLOAD.C r64, (a0, t0 * 1), r77
        VSTORE.C r64, (a1, t0 * 1), r77
        ADD t0, t0, $64
        SUB a4, a4, $64
        VGMASK r77, a4, $64
        BGT a4, $0, .again1
; increment index (64 elements)
; reduce number left by number of elements
; generate mask for next operation
; if there are elements left to process, go back

.done
```

SAXPY

```
;  
;  
;  
;  
;  
; a0 = n, a1 = x, a2 = y, a3 = a  
;  
  
VBROADCAST r80, a3  
VGENMASK r97, a0, $16  
OR t0, r0, r0, r0  
; t0 = 0  
.again1  
VLOAD.F r64, (a1, t0 * 4), r97 ; r64 = x[i]  
VLOAD.F r72, (a2, t0 * 4), r97 ; r72 = y[i]  
VFMA r98, r80, r64, r72 ; tmp = a * x[i] + y[i]  
VMASK r72, r98, r97 ; y[i] = tmp (masked)  
VSTORE r72, (a2, t0 * 4), r97 ; update y[i] in memory  
ADD t0, t0, $16 ; increment index (16 elements)  
SUB a0, a0, $16 ; reduce number left by number of elements  
VGMASK r97, a0, $16 ; generate mask for next operation  
BGT a0, $0, .again1 ; if there are elements left to process, go back  
.done
```

MULTIPLY

BMM – BIT MATRIX MULTIPLY

BMM Rd, Rs1, Rs2

Description:

The BMM instruction treats the bits of register Rs1 and register Rs2 as an 8x8 matrix and performs a bit matrix multiply of the two registers and stores the result in the destination register. An alternate mnemonic for this instruction is MOR.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
34 ₇	Ms ₃	Op ₃	VN ₄	~ ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

for I = 0 to 7

 for j = 0 to 7

 Rd.bit[i][j] = (Rs1[i][0]&Rs2[0][j]) | (Rs1[i][1]&Rs2[1][j]) | ... | (Rs1[i][7]&Rs2[7][j])

Clock Cycles: 1

Execution Units: First Integer ALU

Exceptions: none

Notes:

The bits are numbered with bit 63 of a register representing I,j = 0,0 and bit 0 of the register representing I,j = 7,7.

CLMUL – CARRY-LESS MULTIPLY

Description:

Compute the low order product bits of a carry-less multiply.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
70 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Opc ₇	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa

Exceptions: none

Execution Units: First Integer ALU

Operations

$$Rd = Rs1 * Rs2$$

Exceptions: none

MUL_XX – MULTIPLY REGISTER-REGISTER

Description:

Multiply two registers Rs1 and Rs2 and perform and second operation between the product and a third register Rs3 and place the result in the destination register Rd. All registers are integer registers. Values are treated as signed integers.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
16 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa

OP ₃		Mnemonic
0	Rd = (Rs1 * Rs2) & Rs3	MUL_AND
1	Rd = (Rs1 * Rs2) Rs3	MUL_OR
2	Rd = (Rs1 * Rs2) ^ Rs3	MUL_EOR
3	Rd = (Rs1 * Rs2) + Rs3	MUL_ADD
6	Rd = Rs3 ? (Rs1 * Rs2) : Rd	MULM
others	Reserved	

Size	Clocks
Octa-byte	4

Operation: R2

$$Rd = Rs1 * Rs2 + Rs3$$

Execution Units: All SAUs

Read Ports: Rs1, Rs2, Rs3

Write Ports: Rd

Exceptions: none

Notes:

MULW – MULTIPLY WIDENING

Description:

Compute the product of two values in registers Rs1 and Rs2. Both the operands are treated as signed values. The result is a signed result stored in Rd and Rs3.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
24_7	Ms_3	Op_3		VN_4	$Rs3_6$	$Rs2_6$		$Rs1_6$		Rd_6		$Opcode_7$			

Exceptions: none**Execution Units:** SAUs**Operation**

$$Rd = \text{low bits } (Rs1 * Rs2)$$

$$Rs3 = \text{high bits } (Rs1 * Rs2)$$

Read Ports: Rs1, Rs2**Write Ports:** Rd, Rs3**Exceptions:** none

MULI - MULTIPLY IMMEDIATE

Description:

Multiply a register Rs1 and immediate value and place the product in the destination register Rd. The immediate is sign extended to the machine width. Values are treated as signed integers.

Instruction Format: RI

47	46	45	19	18	13	12	7	6	0
Prc ₂		Immediate ₂₇		Rs1 ₆	Rd ₆	6 ₇			

Clock Cycles: 4

Execution Units: All ALUs

Operation:

$$Rd = Rs1 * \text{immediate}$$

Read Ports: Rs1

Write Ports: Rd

Exceptions:**Notes:**

MULSU_XX – MULTIPLY SIGNED UNSIGNED

Description:

Multiply two registers Rs1 and Rs2 and perform and second operation between the product and a third register Rs3 and place the result in the destination register Rd. All registers are integer registers. The first operand is signed, the second unsigned.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
21 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa

Operation: R3

OP ₃		Mnemonic
0	Rd = (\pm Rs1 * \pm Rs2) & \pm Rs3	MULSU_AND
1	Rd = (\pm Rs1 * \pm Rs2) \pm Rs3	MULSU_OR
2	Rd = (\pm Rs1 * \pm Rs2) ^ \pm Rs3	MULSU_EOR
3	Rd = (\pm Rs1 * \pm Rs2) \pm Rs3	MULSU_ADD
Others	Reserved	

Clock Cycles: 4**Execution Units: SAUs****Read Ports: Rs1, Rs2, Rs3****Write Ports: Rd****Exceptions: none****Notes:**

MULSUW – MULTIPLY SIGNED UNSIGNED WIDENING

Description:

Multiply two registers and place product in the destination register. All registers are integer registers. The first operand is signed, the second unsigned.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
29 ₇	Ms ₃	Op ₃	V ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆		Rd ₆		Opcode ₇			

Operation:

$$Rd = \text{Lower bits} (Rs1 * Rs2)$$

$$Rs3 = \text{Upper bits}(Rs1 * Rs2)$$

Clock Cycles: 1**Execution Units:** DIV**Read Ports:** Rs1, Rs2**Write Ports:** Rd, Rs3**Exceptions:** none**Notes:**

MULU_XX – UNSIGNED MULTIPLY REGISTER-REGISTER

Description:

Multiply two registers and place the product in the destination register. All registers are integer registers.
Values are treated as unsigned integers.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
19 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opc ₇						

Opc ₇	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa

Operation:

OP ₃		Mnemonic
0	Rd = (\pm Rs1 * \pm Rs2) & \pm Rs3	MULU_AND
1	Rd = (\pm Rs1 * \pm Rs2) \pm Rs3	MULU_OR
2	Rd = (\pm Rs1 * \pm Rs2) ^ \pm Rs3	MULU_EOR
3	Rd = (\pm Rs1 * \pm Rs2) \pm Rs3	MULU_ADD
others	Reserved	
6	Rd = Rs3 ? (Rs1 * Rs2) : Rd	MMULU

Clock Cycles: 4**Execution Units: All SAUs****Read Ports: Rs1, Rs2, Rs3****Write Ports: Rd****Exceptions: none****Notes:**

MULUW – UNSIGNED MULTIPLY WIDENING

Description:

Multiply two registers and place the product bits in the destination register. All registers are integer registers. Values are treated as unsigned integers.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
27_7	Ms_3	Op_3		V_4	$Rs3_6$	$Rs2_6$		$Rs1_6$		Rd_6		$Opcode_7$			

Operation:

$$Rd = \text{Lower bits } (Rs1 * Rs2)$$

$$Rs3 = \text{Upper bits } (Rs1 * Rs2)$$

Clock Cycles: 4**Execution Units:** IMUL**Read Ports:** Rs1, Rs2**Write Ports:** Rd, Rs3**Exceptions:** none**Notes:**

MULUI - MULTIPLY UNSIGNED IMMEDIATE

Description:

Multiply a register and immediate value and place the product in the destination register. The immediate is zero extended to the machine width. Values are treated as unsigned integers. Unsigned multiplies are often used in array index calculations.

Instruction Format: RI

47	46	45	19	18	13	12	7	6	0
Prc ₂		Immediate ₂₇		Rs1 ₆	Rd ₆		14 ₇		

Clock Cycles: 4**Execution Units:** All ALUs**Operation:**

$$Rd = Rs1 * \text{immediate}$$

Read Ports: Rs1**Write Ports:** Rd**Exceptions:****Notes:**

VECTOR MULTIPLY

VCLMUL – CARRY-LESS MULTIPLY

Description:

Compute the low order product bits of a carry-less multiply.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
70 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Opc ₇	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa

Exceptions: none

Execution Units: First Integer ALU

Operations

$$Rd = Rs1 * Rs2$$

Exceptions: none

VMUL_XX – MULTIPLY REGISTER-REGISTER

Description:

Multiply two registers Rs1 and Rs2 and perform and second operation between the product and a third register Rs3 and place the result in the destination register Rd. All registers are integer registers. Values are treated as signed integers.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
16 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

OP ₃		Mnemonic
0	Rd = (Rs1 * Rs2) & Rs3	MUL_AND
1	Rd = (Rs1 * Rs2) Rs3	MUL_OR
2	Rd = (Rs1 * Rs2) ^ Rs3	MUL_EOR
3	Rd = (Rs1 * Rs2) + Rs3	MUL_ADD
6	Rd = Rs3 ? (Rs1 * Rs2) : Rd	MULM
others	Reserved	

Size	Clocks
Octa-byte	4

Operation: R2

$$Rd = Rs1 * Rs2 + Rs3$$

Execution Units: All SAUs**Read Ports:** Rs1, Rs2, Rs3**Write Ports:** Rd**Exceptions:** none**Notes:**

VMULU_XX – UNSIGNED MULTIPLY REGISTER-REGISTER

Description:

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

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
19 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Operation:

OP ₃		Mnemonic
0	Rd = (\pm Rs1 * \pm Rs2) & \pm Rs3	MULU_AND
1	Rd = (\pm Rs1 * \pm Rs2) \pm Rs3	MULU_OR
2	Rd = (\pm Rs1 * \pm Rs2) ^ \pm Rs3	MULU_EOR
3	Rd = (\pm Rs1 * \pm Rs2) \pm Rs3	MULU_ADD
others	Reserved	
6	Rd = Rs3 ? (Rs1 * Rs2) : Rd	MMULU

Clock Cycles: 4**Execution Units:** All SAUs**Read Ports:** Rs1, Rs2, Rs3**Write Ports:** Rd**Exceptions:** none**Notes:**

DATA MOVEMENT

SCALAR DATA MOVEMENT

BNDXB – BROADCAST BYTE INDEX

Description:

The destination register is set equal to a constant corresponding to the byte element number. This is equivalent to the following:

OR r8,r0,\$0x0706050403020100,0 OR r9,r0,\$0x0F0E0D0C0B0A0908,0 OR r10,r0,\$0x1716151413121110,0 OR r9,r0,\$0x1F1E1D1C1B1A1918,0	Load v2 with a constant containing the byte index
---	---

Instruction Format: R3

BNDXB Rd

47	41	40 38	37	35	34	28	27	21	20	14	13	7	6	0
26 ₇	Ms ₃	Op ₃	32 ₇	~ ₇	Rs1 ₇	Rd ₇	104 ₇							

Operation:

$$Rd = Rd[1:0] * 8 + 0x0706050403020100$$

Execution Units: SAU #0

Read Ports: Rs1, Rs2

Write Ports: Rd

Exceptions: none

Notes:

BMAP – BYTE MAP

Description:

First the destination register is cleared, then bytes are mapped from the 8-byte source Rs1 into bytes in the destination register. 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 destination register. Each nibble of the 32-bit value indicates the target byte in the destination register.

Note that a 32-bit immediate may be substituted for Rs2.

Instruction Format: R3

BMAP Rd, Rs1, Rs2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
35_7	Ms_3	Op_3	VN_4	$Rs3_6$	$Rs2_6$	$Rs1_6$	Rd_6		107_7					

Operation:

OP_3		Mnemonic
0	$Rd = bmap(Rs, Rs2) \& Rs3$	BMAP_AND
1	$Rd = bmap(Rs1, Rs2) Rs3$	BMAP_OR
2	$Rd = bmap(Rs1, Rs2) ^ Rs3$	BMAP_EOR
3	$Rd = bmap(Rs1, Rs2) \pm Rs3$	BMAP_ADD
4	$Rd = bmap(Rs1, Rs2) \ll Rs3$	BMAP_ASL
5 to 7	Reserved	

Execution Units: SAU #0

Read Ports: Rs1, Rs2

Write Ports: Rd

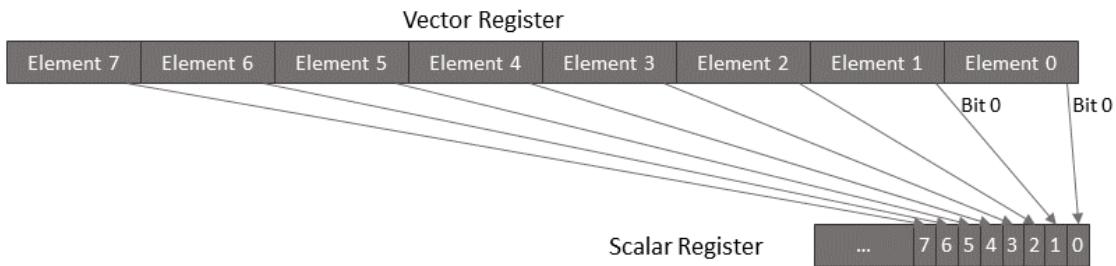
Exceptions: none

Notes:

V2BITS – VECTOR TO BITS

Description:

Convert Boolean vector to bits. The first bit of each vector element is copied to the bit corresponding to the vector element in the destination register. The destination register is a scalar register. The vector elements are stored in Rs1, Rs2, Rs3 and Rs4. Rs1, Rs2, Rs3 and Rs4 are typically four consecutive registers where Rs1 specifies the first register of a quad group.



First the destination register is cleared then bits are copied from the four source operands to the destination operand according to the selected precision.

Instruction Format: R4

V2BITS Rd, Rs1, Rs2, Rs3, Rs4

47	41	40	38	37	35	34	28	27	21	20	14	13	7	6	0
Rs4 ₇	Prc ₃	5 ₃		Rs3 ₇		Rs2 ₇		Rs1 ₇		Rd ₇		12 ₇			

Operation:

Prc ₃	
0	Byte The first bit of each byte in Rs1 is copied to the corresponding bit position in byte #0 of Rd. The first bit of each byte in Rs2 is copied to the corresponding bit position in byte #1 of Rd. The first bit of each byte in Rs3 is copied to the corresponding bit position in byte #2 of Rd. The first bit of each byte in Rs4 is copied to the corresponding bit position in byte #3 of Rd. The result is a 32-bit vector of bits in Rd.
1	Wyde The first bit of each wyde in Rs1 is copied to the corresponding bit position in nybble #0 of Rd. The first bit of each wyde in Rs2 is copied to the corresponding bit position in nybble #1 of Rd. The first bit of each wyde in Rs3 is copied to the corresponding bit position in nybble #2 of Rd. The first bit of each wyde in Rs4 is copied to the corresponding bit position in nybble #3 of Rd. The result is a 16-bit vector of bits in Rd.
2	Tetra The first bit of each tetra in Rs1 is copied to the bit position 0,1 of Rd. The first bit of each tetra in Rs2 is copied to the bit position 2,3 of Rd. The first bit of each tetra in Rs3 is copied to the bit position 4,5 of Rd.

	The first bit of each tetra in Rs4 is copied to the bit position 6,7 of Rd. The result is an 8-bit vector of bits in Rd.
3	Octa The first bit Rs1 is copied to the bit position 0 of Rd. The first bit Rs2 is copied to the bit position 1 of Rd. The first bit Rs3 is copied to the bit position 2 of Rd. The first bit Rs4 is copied to the bit position 3 of Rd. The result is a 4-bit vector of bits in Rd.

Operation

For $x = 0$ to $VL-1$

$$Rd.bit[x] = Rs1[x].bit[Rs2|imm6]$$

Exceptions: none

Example:

```
slt.bp v1, v2, v3, 1    ; compare vectors v2 and v3 for signed less than
v2bits v4, v1            ; move status to bits in v4
add.bp v4,v5,v6          ; perform some masked vector operations
pred pr1,"TTTTIIII"
vmuls v7,v8,v9
pred pr1,"TTTTIIII"
vadd v7,v7,v4
```

Execution Units: SAU #0

Read Ports: Rs1, Rs2

Write Ports: Rd

Exceptions: none

Notes:

CMOVEVN – CONDITIONAL MOVE IF EVEN

CMOVEVN Rd, Rs1, Rs2, Rs3

Description:

If the value in Rs1 is even (LSB=0) then the destination register is set to Rs2, otherwise the destination register is to Rs3.

For CMOVODD swap operands Rs2 and Rs3.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
11 ₇	Ms ₃	4 ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel
116	vector

Clock Cycles: 1

Execution Units: SAU #0

Read Ports: Rs1, Rs2, Rs3

Write Ports: Rd

Operation:

If Rs1[0] = 0 then

Rd = Rs2

else

Rd = Rs3

Exceptions: none

CMOVLEZ – CONDITIONAL MOVE IF LESS THAN OR EQUAL TO ZERO

CMOVLEZ Rd, Rs1, Rs2, Rs3

Description:

If Rs1 is less than or equal to zero then the destination register is set to Rs2, otherwise the destination register is to Rs3.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
11 ₇	Ms ₃	3 ₃	V ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆		Rd ₆		Opcode ₇			

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel

Clock Cycles: 1

Execution Units: SAU #0

Read Ports: Rs1, Rs2, Rs3

Write Ports: Rd

Operation:

If Rs1 <= 0 then

Rd = Rs2

else

Rd = Rs3

Exceptions: none

CMOVLTZ – CONDITIONAL MOVE IF LESS THAN ZERO

CMOVLTZ Rd, Rs1, Rs2, Rs3

Description:

If the value in Rs1 is less than zero then the destination register is set to Rs2, otherwise the destination register is to Rs3.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
11 ₇	Ms ₃	2 ₃	V ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆		Rd ₆		Opcode ₇			

Opcode ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel

Clock Cycles: 1

Execution Units: SAU #0

Read Ports: Rs1, Rs2, Rs3

Write Ports: Rd

Operation:

If Rs1 < 0 then

Rd = Rs2

else

Rd = Rs3

Exceptions: none

CMOVNZ – CONDITIONAL MOVE IF NON-ZERO

CMOVNZ Rd, Rs1, Rs2, Rs3

Description:

If the value in Rs1 is non-zero then the destination register is set to Rs2, otherwise the destination register is to Rs3.

For CMOVZ swap operands Rs2 and Rs3.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
11 ₇	Ms ₃	1 ₃	V ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆	Rd ₆		Opcode ₇				

Opcode ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel

Clock Cycles: 1

Execution Units: SAU #0

Read Ports: Rs1, Rs2, Rs3

Write Ports: Rd

Operation:

If Rs1 then

$$Rd = Rs2$$

else

$$Rd = Rs3$$

Exceptions: none

MAX3 – MAXIMUM SIGNED VALUE

Description:

Determines the maximum of three values in registers Rs1, Rs2 and Rs3 and places the result in the destination register Rd.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
18 ₇	Ms ₃	1 ₃	V ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆	Rd ₆		Opcode ₇				

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel

Execution Units: SAU #0**Read Ports:** Rs1, Rs2, Rs3**Write Ports:** Rd**Operation:**

IF (Rs1 > Rs2 and Rs1 > Rs3)

Rd = Rs1

Else if (Rs2 > Rs3)

Rd = Rs2

Else

Rd = Rs3

MAXU3 – MAXIMUM UNSIGNED VALUE

Description:

Determines the maximum of three values in registers Rs1, Rs2 and Rs3 and places the result in the destination register Rd. Values are unsigned integers.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
23_7	Ms_3	l_3		V_4	$Rs3_6$	$Rs2_6$		$Rs1_6$		Rd_6		Opc_7			

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel

Execution Units: ALU #0 only

Operation:

IF ($Rs1 > Rs2$ and $Rs1 > Rs3$)

Rd = Rs1

Else if ($Rs2 > Rs3$)

Rd = Rs2

Else

Rd = Rs3

MID3 – MIDDLE VALUE

Description:

Determines the middle value of three values in registers Rs1, Rs2 and Rs3 and places the result in the destination register Rd.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
18 ₇	Ms ₃	2 ₃	V ₄		Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		Opcode ₇					

Opc ₇	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa

Execution Units: ALU #0 only

Operation:

IF (Rs1 > Rs2 and Rs1 < Rs3)

Rd = Rs1

Else if (Rs2 > Rs1 and Rs2 < Rs3)

Rd = Rs2

Else

Rd = Rs3

MIDU3 – MIDDLE UNSIGNED VALUE

Description:

Determines the middle value of three values in registers Rs1, Rs2 and Rs3 and places the result in the destination register Rd.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
23 ₇	Ms ₃	2 ₃	V ₄		Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		Opcode ₇					

Opc ₇	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa

Execution Units: ALU #0 only

Operation:

IF (Rs1 > Rs2 and Rs1 < Rs3)

$$Rd = Rs1$$

Else if (Rs2 > Rs1 and Rs2 < Rs3)

$$Rd = Rs2$$

Else

$$Rd = Rs3$$

MIN3 – MINIMUM VALUE

Description:

Determines the minimum of three values in registers Rs1, Rs2 and Rs3 and places the result in the destination register Rd.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
18 ₇	Ms ₃	0 ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Opc ₇	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa

Execution Units: ALU #0 only

Operation:

IF (Rs1 < Rs2 and Rs1 < Rs3)

Rd = Rs1

Else if (Rs2 < Rs3)

Rd = Rs2

Else

Rd = Rs3

MINU3 – MINIMUM UNSIGNED VALUE

Description:

Determines the minimum of three values in registers Rs1, Rs2 and Rs3 and places the result in the destination register Rd.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
23 ₇	Ms ₃	0 ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Opc ₇	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa

Execution Units: ALU #0 only

Operation:

IF (Rs1 < Rs2 and Rs1 < Rs3)

Rd = Rs1

Else if (Rs2 < Rs3)

Rd = Rs2

Else

Rd = Rs3

MOVE – 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. This instruction has access to an eight bit register code and may move between chunks of the vector register file and the scalar register file.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24 23	22 21	20 19	18	13	12	7	6	0
15 ₇	Ms ₃	4 ₃	V ₄	Rs3 ₆	~ ₂	Rs1 _{7:6}	Rd _{7:6}	Rs1 _{5:0}	Rd _{5:0}						Opcode ₇

Op ₃	Movement Operation
0	Move between same type
1	Move scalar to vector (broadcast value)

Operation:

$$Rd = Rs1$$

Clock Cycles: 1**Execution Units:** All Integer ALU's**Exceptions:** none**Notes:**

MOVMR – MOVE MULTIPLE REGISTERS TO REGISTERS

Description:

Move registers-to-registers. This instruction moves up to seven scalar registers to the argument registers in sequence. The first register will be moved to r1, the second register to r2, up to r7 as a destination.

Instruction Format: R3

47	45	44 42	41	37	36	32	31	27	26	22	21	17	16	12	11	7	6	0
N ₃	~ ₃	Rs7 ₅		Rs6 ₅	Rs5 ₅	Rs4 ₅	Rs3 ₅	Rs2 ₅		Rs1 ₅	23 ₇							

Operation:

If (N > 0) R1 = Rs1

If (N > 1) R2 = Rs2

If (N > 2) R3 = Rs3

...

If (N > 6) R7 = Rs7

Clock Cycles: 2**Execution Units:** All Integer ALU's**Exceptions:** none**Notes:**

MOVSXB – MOVE, SIGN EXTEND BYTE

MOVSXB Rd, Rs1

Description:

A byte is extracted from the source operand Rs1, sign extended, and the result placed in the destination register Rd. This is an alternate mnemonic for the [EXT](#) instruction where the bitfield is fixed as the first byte.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
91 ₇	Ms ₃	1 ₃	V ₄	7 ₆	0 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

Clock Cycles:

Execution Units: SAU #0

Exceptions: none

Notes:

MOVSXT – MOVE, SIGN EXTEND TETRA

MOVSXT Rd, Rs1

Description:

A tetra is extracted from the source operand, sign extended, and the result placed in the destination register.
This is an alternate mnemonic for the [EXT](#) instruction where the bitfield is fixed as the first tetra.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
91 ₇	Ms ₃	1 ₃		V ₄	31 ₆	0 ₆		Rs1 ₆	Rd ₆		Opcode ₇				

Operation:

Clock Cycles:

Execution Units: SAU #0

Exceptions: none

Notes:

MOVSW – MOVE, SIGN EXTEND WYDE

MOVSW Rd, Rs1

Description:

A wyde is extracted from the source operand, sign extended, and the result placed in the destination register. This is an alternate mnemonic for the [EXT](#) instruction where the bitfield is fixed as the first wyde.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
91 ₇	Ms ₃	1 ₃		V ₄	15 ₆	0 ₆		Rs1 ₆	Rd ₆		Opcode ₇				

Operation:

Clock Cycles:

Execution Units:

 SAU #0

Exceptions:

 none

Notes:

MUX – MULTIPLEX

MUX Rd, Rs1, Rs2, Rs3

Description:

If a bit in Rs1 is set then the bit of the destination register is set to the corresponding bit in Rs2, otherwise the bit in the destination register is set to the corresponding bit in Rs3.

The multiplex instruction can be used to cause an update of only the selected bits in the destination register. This could be used after a vector instruction which does not support a mask.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
35_7	Ms_3	0_3	V_4	$Rs3_6$	$Rs2_6$	$Rs1_6$	Rd_6							$Opcode_7$

$Opcode_7$	Precision
104	Byte parallel
105	Wyde parallel
106	Tetra parallel
107	octa

Clock Cycles: 1

Execution Units: SAU #0 only

Operation:

For $n = 0$ to 63

If $Rs1_{[n]}$ is set then

$$Rd_{[n]} = Rs2_{[n]}$$

else

$$Rd_{[n]} = Rs3_{[n]}$$

Exceptions: none

VECTOR DATA MOVEMENT

VCMOVEVN – CONDITIONAL MOVE IF EVEN

VCMOVEVN Rd, Rs1, Rs2, Rs3

Description:

If the value in Rs1 is even (LSB=0) then the destination register is set to Rs2, otherwise the destination register is to Rs3. There is no mask for this instruction.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
11 ₇	Ms ₃	4 ₃		V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		116 ₇					

Clock Cycles: 1

Execution Units: SAU #0

Read Ports: Rs1, Rs2, Rs3

Write Ports: Rd

Operation:

If Rs1[0] = 0 then

Rd = Rs2

else

Rd = Rs3

Exceptions: none

VCMOVLEZ – CONDITIONAL MOVE IF LESS THAN OR EQUAL TO ZERO

VCMOVLEZ Rd, Rs1, Rs2, Rs3

Description:

If Rs1 is less than or equal to zero then the destination register is set to Rs2, otherwise the destination register is to Rs3.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
11 ₇	Ms ₃	3 ₃		V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆						116 ₇	

Clock Cycles: 1

Execution Units: SAU #0

Read Ports: Rs1, Rs2, Rs3

Write Ports: Rd

Operation:

If Rs1 <= 0 then

Rd = Rs2

else

Rd = Rs3

Exceptions: none

VCMOVLTZ – CONDITIONAL MOVE IF LESS THAN ZERO

VCMOVLTZ Rd, Rs1, Rs2, Rs3

Description:

If the value in Rs1 is less than zero then the destination register is set to Rs2, otherwise the destination register is to Rs3.

There is no vector mask for this instruction.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
11 ₇	Ms ₃	2 ₃	V ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆	Rd ₆		116 ₇				

Clock Cycles: 1

Execution Units: SAU #0

Read Ports: Rs1, Rs2, Rs3

Write Ports: Rd

Operation:

If Rs1 < 0 then

Rd = Rs2

else

Rd = Rs3

Exceptions: none

VCMOVNZ – CONDITIONAL MOVE IF NON-ZERO

VCMOVNZ Rd, Rs1, Rs2, Rs3

Description:

If the value in Rs1 is non-zero then the destination register is set to Rs2, otherwise the destination register is to Rs3. There is no vector mask for this instruction.

For VCMOVZ swap operands Rs2 and Rs3.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
11 ₇	Ms ₃	1 ₃	V ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆	Rd ₆		116 ₇				

Clock Cycles: 1

Execution Units: SAU #0

Read Ports: Rs1, Rs2, Rs3

Write Ports: Rd

Operation:

If Rs1 then

$$Rd = Rs2$$

else

$$Rd = Rs3$$

Exceptions: none

VMAX – MAXIMUM SIGNED VALUE

Description:

Determines the maximum of two values in registers Rs1 and Rs2 and places the result in the destination register Rd. Only destination elements where the corresponding R3 bit is set will be updated in Rd.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
112 ₇	Ms ₃	1 ₃		V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		116 ₇					

Execution Units: SAU #0

Read Ports: Rs1, Rs2, Rs3

Write Ports: Rd

Operation:

IF (Rs1 > Rs2)

Rd = Rs1

Else

Rd = Rs2

VMAXU – MAXIMUM UNSIGNED VALUE

Description:

Determines the maximum of two values in registers Rs1 and Rs2 and places the result in the destination register Rd. Only destination elements where the corresponding R3 bit is set will be updated in Rd.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
113 ₇	Ms ₃	1 ₃		V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		116 ₇					

Execution Units: SAU #0

Read Ports: Rs1, Rs2, Rs3

Write Ports: Rd

Operation:

IF (Rs1 > Rs2)

Rd = Rs1

Else

Rd = Rs2

VMIN – MINIMUM VALUE

Description:

Determines the minimum of two values in registers Rs1 and Rs2 and places the result in the destination register Rd. Only destination elements where the corresponding R3 bit is set will be updated in Rd.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
112 ₇	Ms ₃	0 ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Execution Units: ALU #0 only

Operation:

IF (Rs1 < Rs2)

Rd = Rs1

Else

Rd = Rs2

VMINU – MINIMUM UNSIGNED VALUE

Description:

Determines the minimum of two unsigned values in registers Rs1 and Rs2 and places the result in the destination register Rd. Only destination elements where the corresponding R3 bit is set will be updated in Rd.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
113 ₇	Ms ₃	0 ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇							

Execution Units: ALU #0 only**Operation:**

IF (Rs1 < Rs2)

$$Rd = Rs1$$

Else

$$Rd = Rs2$$

VMOVE – MOVE REGISTER TO REGISTER

Description:

Move vector register-to-register. This instruction may move between different types of registers. Raw binary data is moved. No data conversions are applied. Only destination elements where the corresponding R3 bit is set will be updated in Rd.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24 23	22 21	20 19	18	13	12	7	6	0
15 ₇	Ms ₃	4 ₃	V ₄	Rs _{3:6}	~ ₂	Rs1 _{7:6}	Rd _{7:6}	Rs1 _{5:0}	Rd _{5:0}	116 ₇					

Op ₃	Movement Operation
0	Move between same type
1	Move scalar to vector (broadcast value)

Operation:

$$Rd = Rs2$$

Clock Cycles: 1**Execution Units:** All Integer ALU's**Exceptions:** none**Notes:**

VMUX – MULTIPLEX

VMUX Rd, Rs1, Rs2, Rs3

Description:

If a bit in Rs1 is set then the bit of the destination register is set to the corresponding bit in Rs2, otherwise the bit in the destination register is set to the corresponding bit in Rs3.

The multiplex instruction can be used to cause an update of only the selected bits in the destination register. This could be used after a vector instruction which does not support a mask.

This instruction does not support a vector mask.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
35_7	Ms_3	0_3	V_4	$Rs3_6$	$Rs2_6$	$Rs1_6$	Rd_6		116_7					

Clock Cycles: 1

Execution Units: SAU #0 only

Operation:

For $n = 0$ to 63

If $Rs1_{[n]}$ is set then

$$Rd_{[n]} = Rs2_{[n]}$$

else

$$Rd_{[n]} = Rs3_{[n]}$$

Exceptions: none

LOGICAL OPERATIONS

SCALAR OPERATIONS

AND – BITWISE AND

Description:

Bitwise ‘and’ two registers with the complement of a third and place the result in the destination register. All registers are treated as integer registers. A sign extended seven-bit immediate constant may be selected for Rs2 and/or Rs3.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
0 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	Octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel

OP ₃		Mnemonic
0	Rd = (Rs1 & Rs2) & Rs3	AND_AND
1	Rd = (Rs1 & Rs2) Rs3	AND_OR
2	Rd = (Rs1 & Rs2) ^ Rs3	AND_EOR
3	Rd = (Rs1 & Rs2) + Rs3	AND_ADD
4	Rd = (Rs1 & Rs2) << Rs3	AND_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 & Rs2) : Rd	MAND

7	Reserved	
---	----------	--

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Exceptions: none

Notes:

ANDI – BITWISE ‘AND’ IMMEDIATE

Description:

Bitwise ‘And’ a register and immediate value and place the result in the destination register. The immediate is one extended to the machine width.

Instruction Format: RI

47	46	45	21	20	14	13	7	6	0
Prc ₂		Immediate ₂₅		Rs1 ₇		Rd ₇		8 ₇	

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Operation:

$$Rd = Rs1 \text{ & immediate}$$

Exceptions:

Notes:

ENOR – BITWISE EXCLUSIVE NOR

Description:

Bitwise exclusively nor three registers and place the result in the destination register. All registers are integer registers.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
10 ₇	Ms ₃	Op ₃	VN ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆		Rd ₆		Opcode ₇			

Operation:

OP ₃		Mnemonic
0	Rd = ~(Rs1 ^ Rs2) & Rs3	ENOR_AND
1	Rd = ~(Rs1 ^ Rs2) Rs3	ENOR_OR
2	Rd = ~(Rs1 ^ Rs2) ^ Rs3	ENOR_EOR
3	Rd = ~(Rs1 ^ Rs2) + Rs3	ENOR_ADD
4	Rd = ~(Rs1 ^ Rs2) << Rs3	ENOR_ASL
5 to 7	Reserved	

Clock Cycles: 1**Execution Units:** All SAUs, all FPUs**Exceptions:** none**Notes:**

EOR – BITWISE EXCLUSIVE OR

Description:

Bitwise exclusively or three registers and place the result in the destination register. All registers are integer registers. A zero extended seven-bit immediate constant may be selected for Rs2 and/or Rs3.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
2 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa

Operation: R3

OP ₃		Mnemonic
0	Rd = (Rs1 ^ Rs2) & Rs3	EOR_AND
1	Rd = (Rs1 ^ Rs2) Rs3	EOR_OR
2	Rd = (Rs1 ^ Rs2) ^ Rs3	EOR_EOR
3	Rd = (Rs1 ^ Rs2) + Rs3	EOR_ADD
4	Rd = (Rs1 ^ Rs2) << Rs3	EOR_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 ^ Rs2) : Rd	MEOR
7	Rd = (^Rs1) ^ (^Rs2) ^ (^Rs3)	PAR (triple parity)

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Exceptions: none

Notes:

EORI – EXCLUSIVE OR IMMEDIATE

Description:

Exclusive Or a register and immediate value and place the sum in the destination register. The immediate is zero extended to the machine width.

Instruction Format: RI

47	46	45	21	20	14	13	7	6	0
Prc ₂		Immediate ₂₅		Rs1 ₇	Rd ₇		10 ₇		

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Operation:

$$Rd = Rs1 \wedge \text{immediate}$$

Exceptions:**Notes:**

MKBOOL – MAKE BOOLEAN

Description:

This instruction reduces the value to a Boolean true or false. If the value is non-zero it is considered true, otherwise if zero it is false.

Instruction Format: R3

47	41	40 38	37	35	34	28	27	21	20	14	13	7	6	0
26_7	Ms_3	Op_3		0_7		12_7		$Rs1_7$		Rd_7		Opc_7		

Operation:

$$Rd = !!Rs1$$

Execution Units: SAU #0**Clock Cycles:** 1**Exceptions:** none**Notes:**

NAND – BITWISE NAND

Description:

Bitwise nand two registers and place the result in the destination register. All registers are integer registers.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
8 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

OP ₃		Mnemonic
0	Rd = ~(Rs1 & Rs2) & Rs3	NAND AND
1	Rd = ~(Rs1 & Rs2) Rs3	NAND OR
2	Rd = ~(Rs1 & Rs2) ^ Rs3	NAND EOR
3	Rd = ~(Rs1 & Rs2) + Rs3	NAND ADD
4	Rd = ~(Rs1 & Rs2) << Rs3	NAND ASL
5 to 7	Reserved	

Clock Cycles: 1**Execution Units:** All SAUs, all FPUs**Exceptions:** none**Notes:**

NOR – BITWISE OR

Description:

Bitwise nor two registers and place the result in the destination register. All registers are integer registers. A zero extended seven-bit immediate constant may be selected for Rs2 and/or Rs3.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
9 ₇	Ms ₃	Op ₃	VN ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆		Rd ₆		Opcode ₇			

Operation:

OP ₃		Mnemonic
0	Rd = ~(Rs1 Rs2) & Rs3	NOR_AND
1	Rd = ~(Rs1 Rs2) Rs3	NOR_OR
2	Rd = ~(Rs1 Rs2) ^ Rs3	NOR_EOR
3	Rd = ~(Rs1 Rs2) + Rs3	NOR_ADD
5	Reserved	
6	Rd = Rs3 ? ~(Rs1 Rs2) : Rd	MNOR
7	Reserved	

Clock Cycles: 1**Execution Units:** All SAUs, all FPUs**Exceptions:** none**Notes:**

NOT – LOGICAL NOT

Description:

This instruction reduces the value to a Boolean true or false. If the value is non-zero it is considered false, otherwise if zero it is true.

Instruction Format: R3

47	41	40 38	37	35	34	28	27	21	20	14	13	7	6	0
26_7	Ms_3	Op_3		0_7	7_7	$Rs1_7$	Rd_7	Opc_7						

Operation:

$$Rd = !Rs1$$

Execution Units: SAU #0**Clock Cycles:** 1**Exceptions:** none**Notes:**

OR – BITWISE OR

Description:

Bitwise or three registers and place the result in the destination register. All registers are integer registers. A zero extended seven-bit immediate constant may be selected for Rs2 and/or Rs3.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
1 ₇	Ms ₃	Op ₃	VN ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆		Rd ₆		Opcode ₇			

Operation:

OP ₃		Mnemonic
0	Rd = (Rs1 Rs2) & Rs3	OR AND
1	Rd = (Rs1 Rs2) Rs3	OR OR
2	Rd = (Rs1 Rs2) ^ Rs3	OR EOR
3	Rd = (Rs1 Rs2) + Rs3	OR ADD
4	Rd = (Rs1 Rs2) << Rs3	OR ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 Rs2) : Rd	MOR
7	Rd = (Rs1 & Rs2) (Rs1 & Rs3) (Rs2 & Rs3)	MAJ

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Exceptions: none

Notes:

ORI - OR IMMEDIATE

Description:

Or a register and immediate value and place the sum in the destination register. The immediate is zero extended to the machine width.

Instruction Format: RI

47	46	45	19	18	13	12	7	6	0
Prc ₂		Immediate ₂₇		Rs1 ₆	Rd ₆		9 ₇		

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Operation:

$$Rd = Rs1 + \text{immediate}$$

Exceptions:**Notes:**

VECTOR LOGICAL OPERATIONS

VAND – BITWISE AND

Description:

Bitwise ‘and’ two registers Rs1 and Rs2 then perform a second operation with a third register Rs3, and place the result in the destination register Rd. All registers are treated as integer registers. A sign extended six-bit immediate constant may be selected for Rs2 and/or Rs3. For a masked AND only destination elements where the corresponding R3 bit is set will be updated in Rd.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
0 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

Ms[0]: 0 = Rs1 register, 1= constant
Ms[1]: 0 = Rs2 register, 1= constant
Ms[2]: 0 = Rs3 register, 1= constant

Opc ₇	Precision
116	Vector, vector
100	Vector, scalar

OP ₃		Mnemonic
0	Rd = (Rs1 & Rs2) & Rs3	VAND_AND
1	Rd = (Rs1 & Rs2) Rs3	VAND_OR
2	Rd = (Rs1 & Rs2) ^ Rs3	VAND_EOR
3	Rd = (Rs1 & Rs2) + Rs3	VAND_ADD
4	Rd = (Rs1 & Rs2) << Rs3	VAND_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 & Rs2) : Rd	VANDM
7	Reserved	

Clock Cycles: 1**Execution Units:** All SAUs, all FPUs**Exceptions:** none**Notes:**

VEOR – BITWISE EXCLUSIVE OR

Description:

Bitwise exclusively or two registers Rs1 and Rs2 and perform a second operation with a third register Rs3, and place the result in the destination register Rd. All registers are integer registers. A zero extended six-bit immediate constant may be selected for Rs2 and/or Rs3. For a masked EOR only destination elements where the corresponding R3 bit is set will be updated in Rd.

Instruction Format: R3

47	41	40	38	37	35	34	28	27	21	20	14	13	7	6	0
2 ₇	Ms ₃	Op ₃		Rs3 ₇		Rs2 ₇		Rs1 ₇		Rd ₇		Opc ₇			

Opc ₇	Operands
116	Vector, vector
100	Vector, scalar

Operation: R3

OP ₃		Mnemonic
0	Rd = (Rs1 ^ Rs2) & Rs3	VEOR_AND
1	Rd = (Rs1 ^ Rs2) Rs3	VEOR_OR
2	Rd = (Rs1 ^ Rs2) ^ Rs3	VEOR_EOR
3	Rd = (Rs1 ^ Rs2) + Rs3	VEOR_ADD
4	Rd = (Rs1 ^ Rs2) << Rs3	VEOR_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 ^ Rs2) : Rd	VEORM
7	Rd = (^Rs1) ^ (^Rs2) ^ (^Rs3)	PAR (triple parity)

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Exceptions: none

Notes:

VENOR – BITWISE EXCLUSIVE NOR

Description:

Bitwise exclusively ‘nor’ two registers Rs1 and Rs2 and perform a second operation with a third register Rs3, and place the result in the destination register Rd. All registers are integer registers. A zero extended six-bit immediate constant may be selected for Rs2 and/or Rs3. For a masked EOR only destination elements where the corresponding R3 bit is set will be updated in Rd.

Instruction Format: R3

47	41	40	38	37	35	34	28	27	21	20	14	13	7	6	0
10 ₇	Ms ₃	Op ₃		Rs3 ₇	Rs2 ₇	Rs1 ₇		Rd ₇	Opc ₇						

Opc ₇	Operands
116	Vector, vector
100	Vector, scalar

Operation: R3

OP ₃		Mnemonic
0	Rd = ~(Rs1 ^ Rs2) & Rs3	VENOR AND
1	Rd = ~(Rs1 ^ Rs2) Rs3	VENOR OR
2	Rd = ~(Rs1 ^ Rs2) ^ Rs3	VENOR EOR
3	Rd = ~(Rs1 ^ Rs2) + Rs3	VENOR ADD
4	Rd = ~(Rs1 ^ Rs2) << Rs3	VENOR ASL
5	Reserved	
6	Rd = Rs3 ? ~(Rs1 ^ Rs2) : Rd	VENORM
7	reserved	

Clock Cycles: 1**Execution Units:** All SAUs, all FPUs**Exceptions:** none**Notes:**

VMKBOOL – MAKE BOOLEAN

Description:

This instruction reduces the value to a Boolean true or false. If the value is non-zero it is considered true, otherwise if zero it is false. For a masked MKBOOL only destination elements where the corresponding R3 bit is set will be updated in Rd.

Instruction Format: R3

47	41	40 38	37 35	34	28	27	21	20	14	13	7	6	0
26_7	Ms_3	Op_3	$Rs3_7$		12_7	$Rs1_7$	Rd_7		Opc_7				

Operation:

$$Rd = \text{!!}Rs1$$

Execution Units: SAU #0**Clock Cycles:** 1**Exceptions:** none**Notes:**

VNAND – BITWISE NAND

Description:

Bitwise ‘nand’ two registers Rs1 and Rs2 then perform a second operation with a third register Rs3, and place the result in the destination register Rd. All registers are treated as integer registers. A sign extended six-bit immediate constant may be selected for Rs2 and/or Rs3. For a masked NAND only destination elements where the corresponding R3 bit is set will be updated in Rd.

Instruction Format: R3

47	41	40	38	37	35	34	28	27	21	20	14	13	7	6	0
8 ₇	Ms ₃	Op ₃		Rs3 ₇		Rs2 ₇		Rs1 ₇		Rd ₇		Opc ₇			

Operation:

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

Opc ₇	Operands
116	Vector, vector
100	Vector, scalar

OP ₃		Mnemonic
0	Rd = ~(Rs1 & Rs2) & Rs3	VNAND_AND
1	Rd = ~(Rs1 & Rs2) Rs3	VNAND_OR
2	Rd = ~(Rs1 & Rs2) ^ Rs3	VNAND_EOR
3	Rd = ~(Rs1 & Rs2) + Rs3	VNAND_ADD
4	Rd = ~(Rs1 & Rs2) << Rs3	VNAND_ASL
5	Reserved	
6	Rd = Rs3 ? ~(Rs1 & Rs2) : Rd	VNANDM
7	Reserved	

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Exceptions: none

VNOR – BITWISE INCLUSIVE NOR

Description:

Bitwise inclusively ‘nor’ two registers Rs1 and Rs2 and perform a second operation with a third register Rs3, and place the result in the destination register Rd. All registers are integer registers. A zero extended six-bit immediate constant may be selected for Rs2 and/or Rs3. For a masked NOR only destination elements where the corresponding R3 bit is set will be updated in Rd.

Instruction Format: R3

47	41	40	38	37	35	34	28	27	21	20	14	13	7	6	0
9 ₇	Ms ₃	Op ₃		Rs3 ₇		Rs2 ₇		Rs1 ₇		Rd ₇		Opc ₇			

Opc ₇	Operands
116	Vector, vector
100	Vector, scalar

Operation: R3

OP ₃		Mnemonic
0	Rd = ~(Rs1 Rs2) & Rs3	VNOR_AND
1	Rd = ~(Rs1 Rs2) Rs3	VNOR_OR
2	Rd = ~(Rs1 Rs2) ^ Rs3	VNOR_EOR
3	Rd = ~(Rs1 Rs2) + Rs3	VNOR_ADD
4	Rd = ~(Rs1 Rs2) << Rs3	VNOR_ASL
5	Reserved	
6	Rd = Rs3 ? ~(Rs1 Rs2) : Rd	VNORM
7	reserved	

Clock Cycles: 1**Execution Units:** All SAUs, all FPUs**Exceptions:** none**Notes:**

VNOT – LOGICAL NOT

Description:

This instruction reduces the value to a Boolean true or false. If the value is non-zero it is considered false, otherwise if zero it is true. All registers are integer registers. For a masked NOT only destination elements where the corresponding R3 bit is set will be updated in Rd.

Instruction Format: R3

47	41	40 38	37	35	34	28	27	21	20	14	13	7	6	0
26_7	Ms_3	Op_3	$Rs3_7$	7_7	$Rs1_7$	Rd_7	Opc_7							

Operation:

$$Rd = !Rs1$$

Execution Units: SAU #0**Clock Cycles:** 1**Exceptions:** none**Notes:**

VOR – BITWISE INCLUSIVE OR

Description:

Bitwise inclusively or two registers Rs1 and Rs2 and perform a second operation with a third register Rs3, and place the result in the destination register Rd. All registers are integer registers. A zero extended six-bit immediate constant may be selected for Rs2 and/or Rs3. For a masked OR only destination elements where the corresponding R3 bit is set will be updated in Rd.

Instruction Format: R3

47	41	40	38	37	35	34	28	27	21	20	14	13	7	6	0
1 ₇	Ms ₃	Op ₃		Rs3 ₇		Rs2 ₇		Rs1 ₇		Rd ₇		Opc ₇			

Opc ₇	Operands
116	Vector, vector
100	Vector, scalar

Operation: R3

OP ₃		Mnemonic
0	Rd = (Rs1 Rs2) & Rs3	VOR AND
1	Rd = (Rs1 Rs2) Rs3	VOR OR
2	Rd = (Rs1 Rs2) ^ Rs3	VOR EOR
3	Rd = (Rs1 Rs2) + Rs3	VOR ADD
4	Rd = (Rs1 Rs2) << Rs3	VOR ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 Rs2) : Rd	VORM
7	Rd = (Rs1 & Rs2) (Rs1 & Rs3) (Rs2 & Rs3)	VMAJ

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Exceptions: none

Notes:

VECTOR REDUCTION OPERATIONS

OVERVIEW

Vector reduction operations reduce the vector value to a scalar value in the destination scalar register. All elements of the vector being reduced are combined under the guidance of a mask register, and the combined with a scalar cascading input. The cascading input allows the vector reduction to span multiple registers.

External exceptions are not allowed in the middle of the vector reduction operation. Vector reduction operations will not cause an exception.

ARITHMETIC OPERATIONS

VREDSUM – VECTOR REDUCING SUMMATION

Description:

Add elements of vector register Rs1 and scalar register Rs2 and place the result in the destination scalar register Rd. All registers are treated as integer registers. Only elements where the corresponding R3 bit is set will be summed. Overflow results in the value wrapping around.

If the cascading input Rs2 is not needed it should be set to 0.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
100 ₇	Ms ₃	0 ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Operation:

Ms[0]: 0 = Rs1 register, 1= constant
Ms[1]: 0 = Rs2 register, 1= constant
Ms[2]: 0 = Rs3 register, 1= constant

For each element in Rs1 where the Rs3 bit is set

Rd[element 0] = element1 + element2 + + element N + Rs2[element 0];

Clock Cycles: 1 per 64-bits

Execution Units: All SAUs, all FPUs

Exceptions: none

Notes:

VREDSSUM – VECTOR REDUCING SATURATING SUMMATION

Description:

Add elements of vector register Rs1 and scalar register Rs2 and place the result in the destination scalar register Rd. All registers are treated as integer registers. Only elements where the corresponding R3 bit is set will be summed. Overflow results in the value saturating at the maximum signed value.

If the cascading input Rs2 is not needed it should be set to 0.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
100 ₇	Ms ₃	4 ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Operation:

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

For each element in Rs1 where the Rs3 bit is set

Rd[element 0] = element1 + element2 + + element N + Rs2[element 0];

Clock Cycles: 1 per 64-bits

Execution Units: All SAUs, all FPUs

Exceptions: none

Notes:

VWREDSUM – VECTOR WIDENING REDUCING SUMMATION

Description:

Sign-extend to double width, then add the elements of vector register Rs1 and scalar register Rs2 and place the result in the destination scalar register Rd. All registers are treated as integer registers. Only elements where the corresponding R3 bit is set will be summed. Overflow results in the value wrapping around.

If the cascading input Rs2 is not needed it should be set to 0.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
100 ₇	Ms ₃	1 ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Operation:

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

For each element in Rs1 where the Rs3 bit is set

Rd[element 0] = element1 + element2 + + element N + Rs2[element 0];

Clock Cycles: 1 per 64-bits

Execution Units: All SAUs, all FPUs

Exceptions: none

Notes:

VWREDSUMU – VECTOR WIDENING REDUCING SUMMATION, UNSIGNED

Description:

Zero-extend to double width, then add the elements of vector register Rs1 and scalar register Rs2 and place the result in the destination scalar register Rd. All registers are treated as integer registers. Only elements where the corresponding R3 bit is set will be summed. Overflow results in the value wrapping around.

If the cascading input Rs2 is not needed it should be set to 0.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
100 ₇	Ms ₃	2 ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Operation:

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

For each element in Rs1 where the Rs3 bit is set

Rd[element 0] = element1 + element2 + + element N + Rs2[element 0];

Clock Cycles: 1 per 64-bits

Execution Units: All SAUs, all FPUs

Exceptions: none

Notes:

LOGICAL OPERATIONS

VREDAND – VECTOR REDUCING BITWISE AND

Description:

Bitwise ‘and’ elements of vector register Rs1 and the first element of Rs2 and place the result in the destination register Rd. All registers are treated as integer registers. Only destination elements where the corresponding R3 bit is set will be ‘ANDED’.

If the cascading input Rs2 is not needed it should be set to -1.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
96 ₇	Ms ₃	0 ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Operation:

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

For each element in Rs1 where the Rs3 bit is set

Rd[element 0] = element1 & element2 & & element N & Rs2[element 0];

Clock Cycles: 1 per 64-bits

Execution Units: All SAUs, all FPUs

Exceptions: none

Notes:

VREDEOR – VECTOR REDUCING BITWISE EXCLUSIVE OR

Description:

Bitwise exclusively ‘or’ elements of vector register Rs1 and the first element of Rs2 and place the result in the destination register Rd. All registers are treated as integer registers. Only destination elements where the corresponding R3 bit is set will be ‘EORed’.

If the cascading input Rs2 is not needed it should be set to 0.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
98 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Operation:

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

For each element in Rs1 where the Rs3 bit is set

Rd[element 0] = element1 ^ element2 ^ ^ element N ^ Rs2[element 0];

Clock Cycles: 1 per 64-bits

Execution Units: All SAUs, all FPUs

Exceptions: none

Notes:

VREDOR – VECTOR REDUCING BITWISE OR

Description:

Bitwise inclusively ‘or’ elements of vector register Rs1 and the first element of Rs2 and place the result in the destination register Rd. All registers are treated as integer registers. Only destination elements where the corresponding R3 bit is set will be ‘ORed’.

If the cascading input Rs2 is not needed it should be set to 0.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
97 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Operation:

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

For each element in Rs1 where the Rs3 bit is set

Rd[element 0] = element1 | element2 | | element N | Rs2[element 0];

Clock Cycles: 1 per 64-bits

Execution Units: All SAUs, all FPUs

Exceptions: none

Notes:

DATA MOVEMENT OPERATIONS

VREDMAX – VECTOR REDUCING MAXIMUM VALUE

Description:

Compare elements of vector register Rs1 and the first element of Rs2 and place the signed maximum value in the destination register Rd. All registers are treated as signed integer registers. Only destination elements where the corresponding R3 bit is set will be tested.

If the cascading input Rs2 is not needed it should be set to 0.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
103 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		116 ₇					

Operation:

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

For each element in Rs1 where the Rs3 bit is set

Rd[element 0] = max(element1, element2, , element N, Rs2[element 0]);

Clock Cycles: 1 per 64-bits

Execution Units: All SAUs, all FPUs

Exceptions: none

Notes:

VREDMAXU – VECTOR REDUCING MAXIMUM UNSIGNED VALUE

Description:

Compare elements of vector register Rs1 and the first element of Rs2 and place the maximum unsigned value in the destination register Rd. All registers are treated as signed integer registers. Only destination elements where the corresponding R3 bit is set will be tested.

If the cascading input Rs2 is not needed it should be set to -1.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
101 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Operation:

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

For each element in Rs1 where the Rs3 bit is set

Rd[element 0] = maxu(element1, element2, , element N, Rs2[element 0]);

Clock Cycles: 1 per 64-bits

Execution Units: All SAUs, all FPUs

Exceptions: none

Notes:

VREDMIN – VECTOR REDUCING MINIMUM VALUE

Description:

Compare elements of vector register Rs1 and the first element of Rs2 and place the signed minimum value in the destination register Rd. All registers are treated as signed integer registers. Only destination elements where the corresponding R3 bit is set will be tested.

If the cascading input Rs2 is not needed it should be set to Rs1.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
102 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		116 ₇					

Operation:

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

For each element in Rs1 where the Rs3 bit is set

Rd[element 0] = min(element1, element2, ..., element N, Rs2[element 0]);

Clock Cycles: 1 per 64-bits

Execution Units: All SAUs, all FPUs

Exceptions: none

Notes:

VREDMINU – VECTOR REDUCING MINIMUM VALUE

Description:

Compare elements of vector register Rs1 and the first element of Rs2 and place the unsigned minimum value in the destination register Rd. All registers are treated as unsigned integer registers. Only destination elements where the corresponding R3 bit is set will be tested.

If the cascading input Rs2 is not needed it should be set to -1.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
99 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Operation:

Ms[0]: 0 = Rs1 register, 1= constant

Ms[1]: 0 = Rs2 register, 1= constant

Ms[2]: 0 = Rs3 register, 1= constant

For each element in Rs1 where the Rs3 bit is set

Rd[element 0] = minu(element1, element2,, element N, Rs2[element 0]);

Clock Cycles: 1 per 64-bits

Execution Units: All SAUs, all FPUs

Exceptions: none

Notes:

COMPARISON OPERATIONS

OVERVIEW

There are two basic types of comparison operators. The first type, compare, returns a bit vector indicating the relationship between the operands, the second type, set, returns a false or a constant depending on the result of the comparison.

CMP - COMPARISON

Description:

Compare two source operands Rs1 and Rs2 and place the result in the destination register Rd. The result is a bit vector identifying the relationship between the two source operands as signed integers. The compare may be cumulative by or'ing the result of previous comparisons with the current one. This may be used to test for the presence or absence of data in an array.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
3 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	Octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel

OP ₃		Mnemonic
0	Rd = (Rs1 ? Rs2) & Rs3	CMP_AND
1	Rd = (Rs1 ? Rs2) Rs3	CMP_OR
2	Rd = (Rs1 ? Rs2) ^ Rs3	CMP_EOR
3 to 5	Reserved	
6	Rd = Rs3 ? (Rs1 ? Rs2) : Rd	CMPM
7	Range Check	CMP_RNG

Operation:
 $Rd = Rs1 ? Rs2$
 $Rd = (Rs1 ? Rs2) | Rs3 ; \text{cumulative}$
Clock Cycles: 1**Execution Units:** All Integer SAUs, all FPUs**Exceptions:** none**Notes:**

Rd Bit	Mnem.	Meaning	Test
Integer Compare Results			
0	EQ	= equal	$a == b$
1	NE	$< >$ not equal	$a <> b$
2	LT	$<$ less than	$a < b$
3	LE	\leq less than or equal	$a \leq b$
4	GE	\geq greater than or equal	$a \geq b$
5	GT	$>$ greater than	$a > b$
6	BC	Bit clear	$!a[b]$
7	BS	Bit set	$a[b]$

Range Check:

Rd Bit	Mnem.	Meaning	Test
Integer Compare Results			
0	GEL		$a \geq b$ and $a < c$
1	GELE		$a \geq b$ and $a \leq c$
2	GL		$a > b$ and $a < c$
3	GLE		$a > b$ and $a \leq c$
4	NGEL		Not ($a \geq b$ and $a < c$)
5	NGELE		Not ($a \geq b$ and $a \leq c$)
6	NGL		Not ($a > b$ and $a < c$)
7	NGLE		Not ($a > b$ and $a \leq c$)

CMPI – COMPARE IMMEDIATE

Description:

Compare two source operands Rs1 and an immediate value and place the result in the destination register Rd. The result is a vector identifying the relationship between the two source operands as signed integers.

Operation:

$$Rd = Rs1 ? Imm$$

Clock Cycles: 1**Execution Units:** All SAUs, all FPUs**Exceptions:** none**Notes:****Instruction Format:** RI

47	46	45	19	28	13	12	7	6	0
Pr _{c2}		Immediate ₂₇		Rs1 ₆	Rd ₆		11 ₇		

Rd Bit	Mnem.	Meaning	Test
Integer Compare Results			
0	EQ	= equal	a == b
1	NE	< > not equal	a <> b
2	LT	< less than	a < b
3	LE	<= less than or equal	a <= b
4	GE	>= greater than or equal	a >= b
5	GT	> greater than	a > b
6	BC	Bit clear	!a[b]
7	BS	Bit set	a[b]

CMPU – UNSIGNED COMPARISON

Description:

Compare two source operands Rs1 and Rs2 and place the result in the destination register Rd. The result is a bit vector identifying the relationship between the two source operands as unsigned integers.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
6 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	Octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel

Op ₃		Mnemonic
0	Rd = (Rs1 ? Rs2) & Rs3	CMPU AND
1	Rd = (Rs1 ? Rs2) Rs3	CMPU OR
2	Rd = (Rs1 ? Rs2) ^ Rs3	CMPU EOR
3 to 6	Reserved	
7	Range Check	CMPU RNG

Operation:

$$Rd = Rs1 ? Rs2$$

Clock Cycles: 1**Execution Units:** All SAUs, all FPUs**Exceptions:** none**Notes:**

Rd Bit	Mnem.	Meaning	Test
Integer Compare Results			
0	EQ	= equal	$a = b$
1	NE	Not equal	$a \neq b$
2	LTU	< less than	$a < b$
3	LEU	\leq less than or equal	$a \leq b$
4	GEU	\geq greater than or equal	$a \geq b$
5	GTU	> greater than	$a > b$
6			
7			

CMPUI – COMPARE UNSIGNED IMMEDIATE

Description:

Compare two source operands Rs1 and an immediate value and place the result in the destination register Rd. The result is a vector identifying the relationship between the two source operands as unsigned integers.

Operation:

$$Rd = Rs1 ? Imm$$

Clock Cycles: 1**Execution Units:** All SAUs, all FPUs**Exceptions:** none**Notes:****Instruction Format:** RI

47	46	45	19	18	13	12	7	6	0
Prc ₂		Immediate ₂₇		Rs1 ₆	Rd ₆	19 ₇			

Rd Bit	Mnem.	Meaning	Test
Integer Compare Results			
0	EQ	Equal	a = b
1	NE	Not equal	a >< b
2	LTU	< less than	a < b
3	LEU	<= less than or equal	a <= b
4	GEU	>= greater than or equal	a >= b
5	GTU	> greater than	a > b
6			
7			

SEQ – SET IF EQUAL

Description:

Compare two source operands Rs1 and Rs2 for equality and place the result in the destination register Rd. The result is the value in Rs3 or zero. Note that constants may be substituted for Rs1, Rs2 and Rs3.

Instruction Format: R3

SEQ Rd, Rs1, Rs2, Rs3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
120 ₇	Ms ₃	Op ₃	VN ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆		Rd ₆		Opcode ₇			

Operation: R3

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	Octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel
116	Size determined by VELSZ

OP ₃		Mnemonic
0	Rd = (Rs1 == Rs2) & Rs3	SEQ_AND
1	Rd = (Rs1 == Rs2) Rs3	SEQ_OR
2	Rd = (Rs1 == Rs2) ^ Rs3	SEQ_EOR
3	Rd = (Rs1 == Rs2) + Rs3	SEQ_ADD
4	Rd = (Rs1 == Rs2) << Rs3	SEQ_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 == Rs2) : Rd	SEQM
7	Reserved	

Clock Cycles: 1

Execution Units: All SAUs

Exceptions: none

Notes:

OR.WP v3, \$-1, \$-1, \$-1	Set all elements of v3 to -1
SEQ.WP v4, v1, v2, v3	Create a mask in v4 for all equal elements of v1 and v2
MADD.WP v5, v6, v7, v4	Perform masked ADD of v6, v7 guided by mask in v4

SEQM – MASKED SET IF EQUAL

Description:

Compare two source operands Rs1 and Rs2 for equality and place the result in the destination register Rd. The result is masked by Rs3.

Instruction Format: R3

SEQ Rd, Rs1, Rs2, Rs3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
120 ₇	Ms ₃	6 ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opc ₇							

Operation: R3

$$Rd = ((Rs1 == Rs2) \& Rs3) ? 1 : 0$$

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	Octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel
116	Size determined by VELSZ

Clock Cycles: 1

Execution Units: All SAUs

Exceptions: none

Notes:

MSEQ.WP v4, v1, v2, vgm MADD.WP v5, v6, v7, v4	Create a mask in v4 for all equal elements of v1 and v2 Perform masked ADD of v6, v7 guided by mask in v4
---	--

SLE – SET IF LESS THAN OR EQUAL

Description:

Compare two source operands Rs1 and Rs2 for signed less than or equal and place the result in the destination register Rd. The result is the value in Rs3 if the comparison is true, otherwise the destination register is set to zero. This instruction may also test for greater than or equal by swapping operands. Note that constants may be substituted for Rs1, Rs2 and Rs3.

Instruction Format: R3

SLE Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
123 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆							Opcode ₇

Operation: R3

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	Octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel
116	Size determined by VELSZ

OP ₃		Mnemonic
0	Rd = (Rs1 <= Rs2) & Rs3	SLE AND
1	Rd = (Rs1 <= Rs2) Rs3	SLE OR
2	Rd = (Rs1 <= Rs2) ^ Rs3	SLE EOR
3	Rd = (Rs1 <= Rs2) + Rs3	SLE ADD
4	Rd = (Rs1 <= Rs2) << Rs3	SLE ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 <= Rs2) : Rd	MSLE
7	Reserved	

Clock Cycles: 1

Execution Units: All SAUs

Exceptions: none

Notes:

SLEU – SET IF UNSIGNED LESS THAN OR EQUAL

Description:

Compare two source operands Rs1 and Rs2 for unsigned less than or equal and place the result in the destination register Rd. The result is the contents of register Rs3 if the condition is true, otherwise the destination register is set to zero. This instruction may also test for greater than or equal by swapping operands. Note that constants may be substituted for Rs1, Rs2 and Rs3.

Instruction Format: R3

SLEU Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
125 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆							Opcode ₇

Operation: R3

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	Octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel
116	Size determined by VELSZ

OP ₃		Mnemonic
0	Rd = (Rs1 <= Rs2) & Rs3	SLEU_AND
1	Rd = (Rs1 <= Rs2) Rs3	SLEU_OR
2	Rd = (Rs1 <= Rs2) ^ Rs3	SLEU_EOR
3	Rd = (Rs1 <= Rs2) + Rs3	SLEU_ADD
4	Rd = (Rs1 <= Rs2) << Rs3	SLEU_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 <= Rs2) : Rd	MSLEU
7	Reserved	

Clock Cycles: 1

Execution Units: All SAUs

Exceptions: none

Notes:

SLT – SET IF LESS THAN

Description:

Compare two source operands Rs1 and Rs2 for signed less than and place the result in the destination register Rd. If Rs1 is less than Rs2 then the result is set to one, otherwise the result is set to zero. This instruction may also test for greater than by swapping operands. Note that constants may be substituted for Rs1, Rs2 and Rs3.

Instruction Format: R3

SLT Rd, Rs1, Rs2, Rs3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
122 ₇	Ms ₃	Op ₃	VN ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆		Rd ₆		Opcode ₇			

Operation: R3

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	Octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel
116	Size determined by VELSZ

OP ₃		Mnemonic
0	Rd = (Rs1 < Rs2) & Rs3	SLT AND
1	Rd = (Rs1 < Rs2) Rs3	SLT OR
2	Rd = (Rs1 < Rs2) ^ Rs3	SLT EOR
3	Rd = (Rs1 < Rs2) + Rs3	SLT ADD
4	Rd = (Rs1 < Rs2) << Rs3	SLT ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 < Rs2) : Rd	MSLT
7	Reserved	

Assembler Default Format:

The default assembler format places a one or a zero in the destination register.

SLT Rd, Rs1, Rs2

Operation:

$Rd = Rs1 < Rs2 ? Imm_{13} : 0$

Clock Cycles: 1

Execution Units: All SAUs

Exceptions: none

Notes:

SLTU –SET IF UNSIGNED LESS THAN

Description:

Compare two source operands Rs1 and Rs2 for unsigned less than and place the result in the destination register Rd. The result is one if the condition is true, otherwise the destination register is set to zero. This instruction may also test for greater than by swapping operands.

Instruction Format: R3

SLTU Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
124 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation: R3

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	Octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel
116	Size determined by VELSZ

OP ₃		Mnemonic
0	Rd = (Rs1 < Rs2) & Rs3	SLTU_AND
1	Rd = (Rs1 < Rs2) Rs3	SLTU_OR
2	Rd = (Rs1 < Rs2) ^ Rs3	SLTU_EOR
3	Rd = (Rs1 < Rs2) + Rs3	SLTU_ADD
4	Rd = (Rs1 < Rs2) << Rs3	SLTU_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 < Rs2) : Rd	MSLTU
7	Reserved	

Clock Cycles: 1

Execution Units: All SAUs

Exceptions: none

Notes:

SNE – SET IF NOT EQUAL

Description:

Compare two source operands in Rs1 and Rs2 for inequality and place the result in the destination register Rd if the comparison is true. The result is set to one. If the comparison is false, the destination register is set to zero.

Instruction Format: R3

SNE Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
121 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation: R3

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	Octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel
116	Size determined by VELSZ

OP ₃		Mnemonic
0	Rd = (Rs1 != Rs2) & Rs3	SNE_AND
1	Rd = (Rs1 != Rs2) Rs3	SNE_OR
2	Rd = (Rs1 != Rs2) ^ Rs3	SNE_EOR
3	Rd = (Rs1 != Rs2) + Rs3	SNE_ADD
4	Rd = (Rs1 != Rs2) << Rs3	SNE_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 != Rs2) : Rd	MSNE
7	Reserved	

Clock Cycles: 1

Execution Units: All SAUs

Exceptions: none

Notes:

VECTOR COMPARE OPERATIONS

VCMP - COMPARISON

Description:

Compare two source operands Rs1 and Rs2 and place the result in the destination register Rd. The result is a bit vector identifying the relationship between the two source operands as signed integers. The compare may be cumulative by or'ing the result of previous comparisons with the current one. This may be used to test for the presence or absence of data in an array.

For a masked compare (VCMPM) only destination elements where the corresponding R3 bit is set will be updated in Rd.

The precision of the operation is determined by the value of the integer field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the integer field of the [U_VLEN](#) register.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
3 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

OP ₃		Mnemonic
0	Rd = (Rs1 ? Rs2) & Rs3	VCMP AND
1	Rd = (Rs1 ? Rs2) Rs3	VCMP OR
2	Rd = (Rs1 ? Rs2) ^ Rs3	VCMP EOR
3 to 5	Reserved	
6	Rd = Rs3 ? (Rs1 ? Rs2) : Rd	VCMPM
7	Range Check	VCMP RNG

VCMPU – UNSIGNED COMPARISON

Description:

Compare two source operands Rs1 and Rs2 and place the result in the destination register Rd. The result is a bit vector identifying the relationship between the two source operands as unsigned integers.

The precision of the operation is determined by the value of the integer field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the integer field of the [U_VLEN](#) register.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
6 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Operation:

Op ₃		Mnemonic
0	Rd = (Rs1 ? Rs2) & Rs3	VCMPU_AND
1	Rd = (Rs1 ? Rs2) Rs3	VCMPU_OR
2	Rd = (Rs1 ? Rs2) ^ Rs3	VCMPU_EOR
3 to 5	Reserved	
6	Rd = Rs3 ? (Rs1 ? Rs2) : Rd	VCMPUM
7	Range Check	VCMPU_RNG

Clock Cycles: 1**Execution Units:** All SAUs, all FPUs**Exceptions:** none**Notes:**

Operation:
 $Rd = Rs1 ? Rs2$
 $Rd = (Rs1 ? Rs2) | Rs3 ; \text{cumulative}$
Clock Cycles: 1**Execution Units:** All Integer SAUs, all FPUs**Exceptions:** none**Notes:**

Rd Bit	Mnem.	Meaning	Test
Integer Compare Results			
0	EQ	= equal	$a == b$
1	NE	$< >$ not equal	$a <> b$
2	LT	$<$ less than	$a < b$
3	LE	\leq less than or equal	$a \leq b$
4	GE	\geq greater than or equal	$a \geq b$
5	GT	$>$ greater than	$a > b$
6	BC	Bit clear	$!a[b]$
7	BS	Bit set	$a[b]$

Range Check:

Rd Bit	Mnem.	Meaning	Test
Integer Compare Results			
0	GEL		$a \geq b$ and $a < c$
1	GELE		$a \geq b$ and $a \leq c$
2	GL		$a > b$ and $a < c$
3	GLE		$a > b$ and $a \leq c$
4	NGEL		Not ($a \geq b$ and $a < c$)
5	NGELE		Not ($a \geq b$ and $a \leq c$)
6	NGL		Not ($a > b$ and $a < c$)
7	NGLE		Not ($a > b$ and $a \leq c$)

VSEQ – SET IF EQUAL

Description:

Compare two source operands Rs1 and Rs2 for equality and place the result in the destination register Rd. The result is the value in Rs3 or zero. Note that constants may be substituted for Rs1, Rs2 and Rs3. For a masked SET only destination elements where the corresponding R3 bit is set will be updated in Rd.

The precision of the operation is determined by the value of the integer field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the integer field of the [U_VLEN](#) register.

Instruction Format: R3

SEQ Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
120 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation: R3

Opc ₇	Operands
116	Vector = vector == vector Size determined by VELSZ

OP ₃		Mnemonic
0	Rd = (Rs1 == Rs2) & Rs3	VSEQ_AND
1	Rd = (Rs1 == Rs2) Rs3	VSEQ_OR
2	Rd = (Rs1 == Rs2) ^ Rs3	VSEQ_EOR
3	Rd = (Rs1 == Rs2) + Rs3	VSEQ_ADD
4	Rd = (Rs1 == Rs2) << Rs3	VSEQ_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 == Rs2) : Rd	VSEQM
7	Reserved	

Clock Cycles: 1

Execution Units: All SAUs

Exceptions: none

Notes:

VSLE – SET IF LESS THAN OR EQUAL

Description:

Compare two source operands Rs1 and Rs2 for signed less than or equal and place the result in the destination register Rd. This instruction may also test for greater than or equal by swapping operands. Note that constants may be substituted for Rs1, Rs2 and Rs3.

The precision of the operation is determined by the value of the integer field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the integer field of the [U_VLEN](#) register.

Instruction Format: R3

VSLE Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
123 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Operation: R3

OP ₃		Mnemonic
0	Rd = (Rs1 <= Rs2) & Rs3	VSLE AND
1	Rd = (Rs1 <= Rs2) Rs3	VSLE OR
2	Rd = (Rs1 <= Rs2) ^ Rs3	VSLE_EOR
3	Rd = (Rs1 <= Rs2) + Rs3	VSLE_ADD
4	Rd = (Rs1 <= Rs2) << Rs3	VSLE_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 <= Rs2) : Rd	VSLEM
7	Reserved	

Clock Cycles: 1

Execution Units: All SAUs

Exceptions: none

Notes:

VSLEU – SET IF LESS THAN OR EQUAL

Description:

Compare two source operands Rs1 and Rs2 for signed less than or equal and place the result in the destination register Rd. This instruction may also test for greater than or equal by swapping operands. Note that constants may be substituted for Rs1, Rs2 and Rs3.

The precision of the operation is determined by the value of the integer field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the integer field of the [U_VLEN](#) register.

Instruction Format: R3

VSLEU Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
125 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Operation: R3

OP ₃		Mnemonic
0	Rd = (Rs1 <= Rs2) & Rs3	VSLE AND
1	Rd = (Rs1 <= Rs2) Rs3	VSLE OR
2	Rd = (Rs1 <= Rs2) ^ Rs3	VSLE_EOR
3	Rd = (Rs1 <= Rs2) + Rs3	VSLE_ADD
4	Rd = (Rs1 <= Rs2) << Rs3	VSLE_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 <= Rs2) : Rd	VSLEM
7	Reserved	

Clock Cycles: 1

Execution Units: All SAUs

Exceptions: none

Notes:

VSLT – SET IF LESS THAN

Description:

Compare two source operands Rs1 and Rs2 for signed less than and place the result in the destination register Rd. If Rs1 is less than Rs2 then the result is set to one, otherwise the result is set to zero. This instruction may also test for greater than by swapping operands. Note that constants may be substituted for Rs1, Rs2 and Rs3. For a masked SET only destination elements where the corresponding R3 bit is set will be updated in Rd.

The precision of the operation is determined by the value of the integer field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the integer field of the [U_VLEN](#) register.

Instruction Format: R3

VSLT Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
122 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation: R3

Opc ₇	Operands
116	Vector = vector < vector Size determined by VELSZ

OP ₃		Mnemonic
0	Rd = (Rs1 < Rs2) & Rs3	VSLT AND
1	Rd = (Rs1 < Rs2) Rs3	VSLT OR
2	Rd = (Rs1 < Rs2) ^ Rs3	VSLT_EOR
3	Rd = (Rs1 < Rs2) + Rs3	VSLT_ADD
4	Rd = (Rs1 < Rs2) << Rs3	VSLT_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 < Rs2) : Rd	VSLTM
7	Reserved	

Assembler Default Format:

The default assembler format places a one or a zero in the destination register.

SLT Rd, Rs1, Rs2

Operation:

$Rd = Rs1 < Rs2 ? Imm_{13} : 0$

Clock Cycles: 1

Execution Units: All SAUs

Exceptions: none

Notes:

VSLTU – SET IF LESS THAN UNSIGNED

Description:

Compare two source operands Rs1 and Rs2 for unsigned less than and place the result in the destination register Rd. If Rs1 is less than Rs2 then the result is set to one, otherwise the result is set to zero. This instruction may also test for greater than by swapping operands. Note that constants may be substituted for Rs1, Rs2 and Rs3. For a masked SET only destination elements where the corresponding R3 bit is set will be updated in Rd.

The precision of the operation is determined by the value of the integer field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the integer field of the [U_VLEN](#) register.

Instruction Format: R3

VSLTU Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
124 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Operation: R3

OP ₃		Mnemonic
0	Rd = (Rs1 < Rs2) & Rs3	VSLT_AND
1	Rd = (Rs1 < Rs2) Rs3	VSLT_OR
2	Rd = (Rs1 < Rs2) ^ Rs3	VSLT_EOR
3	Rd = (Rs1 < Rs2) + Rs3	VSLT_ADD
4	Rd = (Rs1 < Rs2) << Rs3	VSLT_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 < Rs2) : Rd	VSLTM
7	Reserved	

Assembler Default Format:

The default assembler format places a one or a zero in the destination register.

VSLTU Rd, Rs1, Rs2, Rs3

Clock Cycles: 1

Execution Units: All SAUs

Exceptions: none

Notes:

SHIFT AND ROTATE OPERATIONS

OVERVIEW

Shift instructions can take the place of some multiplication and division instructions. Some architectures provide shifts that shift only by a single bit. Others use counted shifts, the original 80x86 used multiple clock cycles to shift by an amount stored in the CX register. Table888 and Thor use a barrel shifter to allow shifting by an arbitrary amount in a single clock cycle. Shifts are infrequently used, and a barrel (or funnel) shifter is relatively expensive in terms of hardware resources.

Qupls4 has a full complement of shift instructions including rotates.

SCALAR SHIFT INSTRUCTIONS

ASLC –ARITHMETIC SHIFT LEFT WITH CARRY

Description:

This instruction is an extended precision shift operation with carry in and carry out. Left shift an operand Rs1 by an operand Rs2 and place result in the destination register Rd1. Use operand Rs3 as a carry input and output carry bits to destination register Rd2.

Instruction Format: EXTD

47	41	40 38	37 35	34	28	27	21	20	14	13	7	6	0
Rd2 ₇	Ms ₃	2 ₃	Rs3 ₇	Rs2 ₇	Rs1 ₇	Rd1 ₇	12 ₇						

Operation:

$$\{Rd2, Rd1\} = (\{Rs1, Rs3\} \ll Rs2) \gg 64$$

Operation Size: .o**Execution Units:** SAU #0**Exceptions:** none**Example:**

ASL_XX –ARITHMETIC SHIFT LEFT THEN OP

Description:

Left shift an operand value Rs1 by an operand value Rs2. Perform a second operation between the intermediate result and Rs3and place the result in the destination register Rd.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
83 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcde ₇						

Operation:

OP ₃		Mnemonic
0	Rd = (Rs1 << Rs2) & Rs3	ASL_AND
1	Rd = (Rs1<< Rs2) Rs3	ASL_OR
2	Rd = (Rs1 << Rs2) ^ Rs3	ASL_EOR
3	Rd = (Rs1 << Rs2) + Rs3	ASL_ADD
4	Rd = (Rs1 << Rs2) << Rs3	ASL_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 << Rs2) : Rd	ASLM
7	reserved	

Operation Size: .o

Execution Units: SAU #0

Exceptions: none

Example:

ASR –ARITHMETIC SHIFT RIGHT

Description:

Right shift an operand value in Rs1 by an operand value in Rs2 and place the result in the destination register Rd. The sign bit is shifted into the most significant bits.

Instruction Format: SHIFT

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
82 ₇	Ms ₃	Rmd ₃	VN ₄		Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		Opcode ₇					

37:35	Round Mode	
2	Truncate	Discards bits
3	Round towards zero	If result is negative, then it is rounded up
4	Round up	If there was a carry out of the LSB, add one

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel

Operation:

$$Rd = Rs3 ? \text{round}(Rs1 \gg R2) : Rd$$

Operation Size: .o**Execution Units: SAU #0****Exceptions:** none**Example:**

ASRC –ARITHMETIC SHIFT RIGHT WITH CARRY

Description:

This instruction is an extended precision shift operation with carry in and carry out. Left shift an operand Rs1 by an operand Rs2 and place result in the destination register Rd1. Use operand Rs3 as a carry input and output carry bits to destination register Rd2.

Instruction Format: EXTD

47	41	40 38	37	35	34	28	27	21	20	14	13	7	6	0
Rd2 ₇	Ms ₃	3 ₃		Rs3 ₇		Rs2 ₇		Rs1 ₇		Rd1 ₇		12 ₇		

Operation:

$$\{Rd1, Rd2\} = (\{Rs3, Rs1, 0\} \gg Rs2)$$

Operation Size: .o**Execution Units:** SAU #0**Exceptions:** none**Example:**

LSRC –LOGICAL SHIFT RIGHT WITH CARRY

Description:

This instruction is an extended precision shift operation with carry in and carry out. Right shift an operand Rs1 by an operand Rs2 and place result in the destination register Rd1. Use operand Rs3 as a carry input and output carry bits to destination register Rd2.

Instruction Format: EXTD

47	41	40 38	37 35	34	28	27	21	20	14	13	7	6	0
Rd2 ₇	Ms ₃	4 ₃	Rs3 ₇	Rs2 ₇	Rs1 ₇	Rd1 ₇	12 ₇						

Operation:

$$\{Rd1, Rd2\} = (\{Rs3, Rs1, 0\} \gg Rs2)$$

Operation Size: .o**Execution Units:** SAU #0**Exceptions:** none**Example:**

LSR_XX –LOGIC SHIFT RIGHT THEN OP

Description:

Right shift an operand value Rs1 by an operand value Rs2. Perform a second operation between the intermediate result and Rs3and place the result in the destination register Rd. This instruction may be used to perform unsigned bitfield extracts.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
84 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇							

Operation:

OP ₃		Mnemonic
0	Rd = (Rs1 >> Rs2) & Rs3	LSR_AND
1	Rd = (Rs1>> Rs2) Rs3	LSR_OR
2	Rd = (Rs1 >> Rs2) ^ Rs3	LSR_EOR
3	Rd = (Rs1 >> Rs2) ± Rs3	LSR_ADD
4	Rd = (Rs1 >> Rs2) << Rs3	LSR_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 >> Rs2) : Rd	MLSR
7	Reserved	

Operation Size: .o**Execution Units: SAU #0****Exceptions:** none**Example:**

ROL_ XX –ROTATE LEFT THEN OP

Description:

Rotate left an operand value by an operand value and place the result in the destination register. The most significant bits are shifted into the least significant bits. The first operand must be in a register specified by Rs1. The second operand may be either a register specified by the Rs2 field of the instruction, or an immediate value.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
80 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		Opcode ₇					

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel

Operation:

OP ₃		Mnemonic
0	Rd = (Rs1 << Rs2) & Rs3	ROL AND
1	Rd = (Rs1<< Rs2) Rs3	ROL OR
2	Rd = (Rs1 << Rs2) ^ Rs3	ROL EOR
3	Rd = (Rs1 << Rs2) + Rs3	ROL ADD
4	Rd = (Rs1 << Rs2) << Rs3	ROL ASL
5 to 7	Reserved	

Operation Size: .o

Execution Units: SAU #0

Exceptions: none

Example:

ROR_XX –ROTATE RIGHT THEN OP

Description:

Rotate right an operand value by an operand value and place the result in the destination register. The least significant bits are shifted into the most significant bits. The first operand must be in a register specified by Rs1 and Rs2. The second operand may be either a register specified by the Rs3 field of the instruction, or an immediate value.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
81 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Opc ₇	Precision
104	Byte
105	Wyde
106	Tetra
107	octa
112	Byte parallel
113	Wyde parallel
114	Tetra parallel
115	Octa parallel

Operation:

OP ₃		Mnemonic
0	Rd = (Rs1 >> Rs2) & Rs3	ROR_AND
1	Rd = (Rs1>> Rs2) Rs3	ROR_OR
2	Rd = (Rs1 >> Rs2) ^ Rs3	ROR_EOR
3	Rd = (Rs1 >> Rs2) + Rs3	ROR_ADD
4	Rd = (Rs1 >> Rs2) << Rs3	ROR_ASL
5 to 7	Reserved	

Operation Size: .o

Execution Units: SAU #0

Exceptions: none

Example:

VECTOR SHIFT INSTRUCTIONS

OVERVIEW

There are two kinds of vector shifting instructions, one that works on individual elements and a second which operates on the whole vector. The vector slide instructions operate on the whole vector but are currently limited to shifting by 64-bits as that is the size of the carry register.

VASL_XX – ARITHMETIC SHIFT LEFT THEN OP

Description:

Left shift an operand value Rs1 by an operand value Rs2. Perform a second operation between the intermediate result and Rs3 and place the result in the destination register Rd.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
83 ₇	Ms ₃	Op ₃	VN	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Operation:

OP ₃		Mnemonic
0	Rd = (Rs1 << Rs2) & Rs3	ASL_AND
1	Rd = (Rs1 << Rs2) Rs3	ASL_OR
2	Rd = (Rs1 << Rs2) ^ Rs3	ASL_EOR
3	Rd = (Rs1 << Rs2) + Rs3	ASL_ADD
4	Rd = (Rs1 << Rs2) << Rs3	ASL_ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 << Rs2) : Rd	ASLM
7	reserved	

Operation Size: .o

Execution Units: SAU #0

Exceptions: none

Example:

VASR –ARITHMETIC SHIFT RIGHT

Description:

Right shift an operand value in Rs1 by an operand value in Rs2 and place the result in the destination register Rd. The sign bit is shifted into the most significant bits.

Instruction Format: SHIFT

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
82 ₇	Ms ₃	Rmd ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

37:35	Round Mode	
2	Truncate	Discards bits
3	Round towards zero	If result is negative, then it is rounded up
4	Round up	If there was a carry out of the LSB, add one

Operation:

$$Rd = Rs3 ? \text{round}(Rs1 \gg R2) : Rd$$

Operation Size: .o**Execution Units:** SAU #0**Exceptions:** none**Example:**

VLSR_XX –LOGIC SHIFT RIGHT THEN OP

Description:

Right shift an operand value Rs1 by an operand value Rs2. Perform a second operation between the intermediate result and Rs3and place the result in the destination register Rd. This instruction may be used to perform unsigned bitfield extracts.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
84 ₇	Ms ₃	Op ₃		VN ₄	Rs3 ₆	Rs2 ₆		Rs1 ₆		Rd ₆		Opcode ₇			

Operation:

OP ₃		Mnemonic
0	Rd = (Rs1 >> Rs2) & Rs3	LSR AND
1	Rd = (Rs1>> Rs2) Rs3	LSR OR
2	Rd = (Rs1 >> Rs2) ^ Rs3	LSR EOR
3	Rd = (Rs1 >> Rs2) ± Rs3	LSR ADD
4	Rd = (Rs1 >> Rs2) << Rs3	LSR ASL
5	Reserved	
6	Rd = Rs3 ? (Rs1 >> Rs2) : Rd	MLSR
7	Reserved	

Operation Size: .o

Execution Units: SAU #0

Exceptions: none

Example:

VROL_XX –ROTATE LEFT THEN OP

Description:

Rotate left an operand value by an operand value and place the result in the destination register. The most significant bits are shifted into the least significant bits. The first operand must be in a register specified by Rs1. The second operand may be either a register specified by the Rs2 field of the instruction, or an immediate value.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
80 ₇	Ms ₃	Op ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	116 ₇						

Operation:

OP ₃		Mnemonic
0	Rd = (Rs1 << Rs2) & Rs3	ROL AND
1	Rd = (Rs1<< Rs2) Rs3	ROL OR
2	Rd = (Rs1 << Rs2) ^ Rs3	ROL EOR
3	Rd = (Rs1 << Rs2) + Rs3	ROL ADD
4	Rd = (Rs1 << Rs2) << Rs3	ROL ASL
5 to 7	Reserved	
6	Rd = Rs3 ? (Rs1 << Rs2) : Rd	ROLM

Operation Size: .o

Execution Units: SAU #0

Exceptions: none

Example:

VROR_XX –ROTATE RIGHT THEN OP

Description:

Rotate right an operand value by an operand value and place the result in the destination register. The least significant bits are shifted into the most significant bits. The first operand must be in a register specified by Rs1 and Rs2. The second operand may be either a register specified by the Rs3 field of the instruction, or an immediate value.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
81 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		116 ₇					

Operation:

OP ₃		Mnemonic
0	Rd = (Rs1 >> Rs2) & Rs3	ROR_AND
1	Rd = (Rs1>> Rs2) Rs3	ROR_OR
2	Rd = (Rs1 >> Rs2) ^ Rs3	ROR_EOR
3	Rd = (Rs1 >> Rs2) + Rs3	ROR_ADD
4	Rd = (Rs1 >> Rs2) << Rs3	ROR_ASL
5 to 7	Reserved	
6	Rd = Rs3 ? (Rs1 >> Rs2) : Rd	RORM

Operation Size: .o

Execution Units: SAU #0

Exceptions: none

Example:

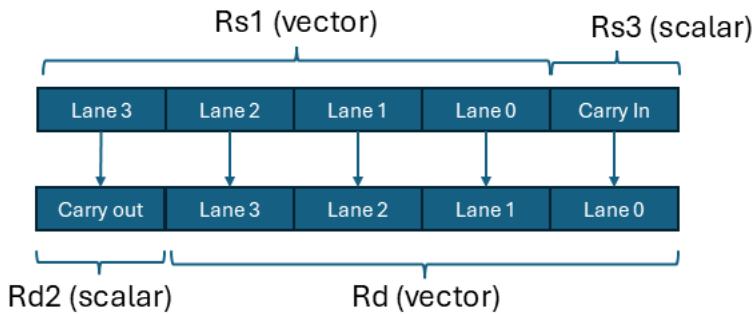
VSHLV – SHIFT VECTOR LEFT (VECTOR SLIDE UP)

Description

Elements of the vector are transferred upwards to the next element position. The first is loaded with the value in scalar register Rs3. The highest element is stored in scalar register Rd2. This is also called a slide operation. The vector register may be shifted a maximum of 64-bits.

Rs1 contains the vector to be shifted. The resulting vector is stored in Rd.

Rs2 should be loaded with the size of a vector element, otherwise the operation will act as a very wide left shift operation.



Instruction Formats:

VSHLV Rd, Rs1, Rs2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
Rd2 ₇	Ms ₃	6 ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	12 ₇						R3

Operation

$$\{Rd2, Rd\} = \{Rs1, Rs3\} \ll Rs2 \quad ; \text{ Rs1 and Rd are vector registers}$$

Exceptions:

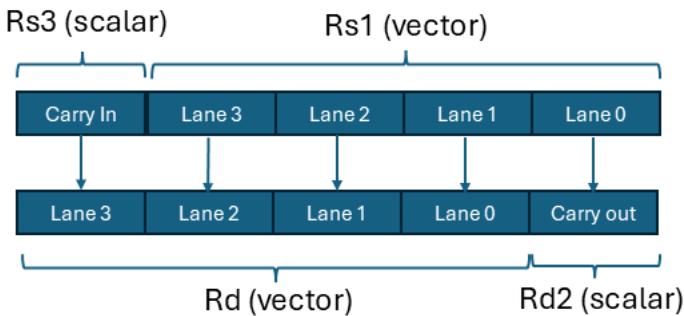
Notes:

This instruction is converted by the CPU into multiple ASLC instructions.

VSHRV – SHIFT VECTOR RIGHT (VECTOR SLIDE DOWN)

Description

Elements of the vector are transferred downwards to the next element position. The last is loaded with the value from scalar register Rs3. Carry out is loaded into the scalar register Rd2. This is also called a slide operation. The vector register may be shifted a maximum of 64-bits to the right. The image depicts just a single element shift. Rs2 should be loaded with a multiple of the lane size in bits otherwise a wide shift will result.



VSHRV Rd, Rs1, Rs2

47	41	40	38	37	35	34	31	30	25	24	19	18	13	12	7	6	0	R3
Rd2 ₇	Ms ₃	7 ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	12 ₇										

Operation

$\{Rd2, Rd\} = \{Rs1, Rs3\} \ll Rs2$; Rd and Rs1 are vector registers

Exceptions: none

Notes:

This instruction is converted by the CPU into multiple LSRC instructions.

BIT-FIELD MANIPULATION OPERATIONS

OVERVIEW

Many CPUs do not have direct support for bit-field manipulation. Instead, they rely on ordinary logical and shift operations. The benefit of having bit-field operations is that they are more code dense than performing the operations using other ALU ops.

The beginning and end of a bitfield may be specified as either a pair of immediate constants or in a pair of registers.

Bitfield operations are only supported on the first SAU (simple arithmetic unit) as they occur infrequently.

CLR – CLEAR BIT FIELD

Description:

A bit field in the source operand Rs1 is cleared and the result placed in the destination register Rd. Rs2 specifies the first bit of the bitfield, Rs3 specifies the last bit of the bitfield. Immediate constants may be substituted for Rs2 and Rs3.

Instruction Format: R3

CLR Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
88 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = Rs1$$

$$Rd[ME:MB] = 0$$

Clock Cycles:

Execution Units: SAU #0

Exceptions: none

Notes:

COM – COMPLEMENT BIT FIELD

Description:

A bit field in the source operand Rs1 is one's complemented and the result placed in the destination register Rd. Rs2 specifies the first bit of the bitfield, Rs3 specifies the last bit of the bitfield. Immediate constants may be substituted for Rs2 and Rs3.

Operation:**Instruction Format:** R3**COM Rd, Rs1, Rs2, Rs3**

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
89 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆							Opcode ₇

Clock Cycles:**Execution Units:** SAU #0**Exceptions:** none**Notes:**

DEP – DEPOSIT BIT FIELD

Description:

A source operand Rs1 is transferred to a bitfield in the destination register Rd. Rs2 specifies the first bit of the bitfield, Rs3 specifies the last bit of the bitfield. Immediate constants may be substituted for Rs2 and Rs3.

Instruction Formats:**DEP Rd, Rs1, Rs2, Rs3**

47	41	40	38	37	35	34	33	28	27	26	21	20	19	14	13	12	7	6	0
92 ₇	Ms ₃	Op ₃	N3	Rs3 ₆	N2	Rs2 ₆	N1	Rs1 ₆	Nd	Rd ₆	Opc ₇								

Operation:

MB = Rs2 or Imm

ME = Rs3 or Imm

Rd[ME:MB] = Rs1

Clock Cycles: 1**Execution Units:** SAU #0**Read Ports:** Rd, Rs1, Rs2, Rs3**Exceptions:** none**Notes:**

EXT – EXTRACT BIT FIELD

Description:

A bit field is extracted from the source operand Rs1, sign extended, and the result placed in the destination register Rd. Rs2 specifies the first bit of the bitfield, Rs3 specifies the last bit of the bitfield. Immediate constants may be substituted for Rs2 and Rs3.

Instruction Format: R3

EXT Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
91 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = \text{sign extend}(Rs1[ME:MB])$$

Clock Cycles:

Execution Units: SAU #0

Exceptions: none

Notes:

EXTU – EXTRACT UNSIGNED BIT FIELD

EXTU Rd, Rs1, Rs2, Rs3

Description:

A bit field is extracted from the source operand Rs1, zero extended, and the result placed in the destination register Rd. Rs2 specifies the first bit of the bitfield, Rs3 specifies the last bit of the bitfield. Immediate constants may be substituted for Rs2 and Rs3.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
90 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

Rd = zero extend(Rs1[ME:MB])

Clock Cycles:

Execution Units: SAU #0

Exceptions: none

Notes:

SET – SET BIT FIELD

SET Rd, Rs1, Rs2, Rs3

Description:

A bit field in the source operand Rs1 is set to all ones and the result placed in the destination register Rd. Rs2 specifies the first bit of the bitfield, Rs3 specifies the last bit of the bitfield. Immediate constants may be substituted for Rs2 and Rs3.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
93 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = Rs1$$

$$Rd[ME:MB] = 111\dots$$

Clock Cycles:

Execution Units: SAU #0

Exceptions: none

Notes:

CRYPTOGRAPHIC ACCELERATOR INSTRUCTIONS

AES64DS – FINAL ROUND DECRYPTION

Description:

Perform the final round of decryption for the AES standard. Register Rs1 represents the entire AES state.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
26 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	18 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

Exceptions:

none

AES64DSM – MIDDLE ROUND DECRYPTION

Description:

Perform a middle round of decryption for the AES standard. Register Rs1 represents the entire AES state.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
26 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	19 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

Execution Units:

crypto

Exceptions:

none

AES64ES – FINAL ROUND ENCRYPTION

Description:

Perform the final round of encryption for the AES standard. Register Rs1 represents the entire AES state.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
26 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	20 ₆	Rs1 ₆	Rd ₆	Opcod _{e7}							

Operation:

Execution Units: crypto

Exceptions: none

AES64ESM – MIDDLE ROUND ENCRYPTION

Description:

Perform a middle round of encryption for the AES standard. Register Rs1 represents the entire AES state.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
26 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	21 ₆	Rs1 ₆	Rd ₆	Opcod _{e7}							

Operation:

Execution Units: crypto

Exceptions: none

SHA256SIG0

Description:

Implements the Sigma0 transformation function used in the SHA2-256 and SHA2-224 hash function. Only the low order 32 bits of Rs1 are operated on. The 32-bit result is sign extended to the machine width.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
26_7	Ms_3	\sim_3		VN_4	0_6		24_6		$Rs1_6$		Rd_6		$Opcode_7$		

Operation:

$$Rd = \text{sign extend}(\text{ror32}(Rs1,7) \wedge \text{ror32}(Rs1,18) \wedge (Rs1_{32} \gg 3))$$

Execution Units: crypto**Exceptions:** none

SHA256SIG1

Description:

Implements the Sigma1 transformation function used in the SHA2-256 and SHA2-224 hash function. Only the low order 32 bits of Rs1 are operated on. The 32-bit result is sign extended to the machine width.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
26_7	Ms_3	\sim_3		VN_4	0_6	25_6	$Rs1_6$	Rd_6		$Opcode_7$					

Clock Cycles: 1**Operation:**

$$Rd = \text{sign extend}(\text{ror32}(Rs1,17) \wedge \text{ror32}(Rs1,19) \wedge (Rs1_{32} \gg 10))$$

Execution Units: crypto**Exceptions:** none

SHA256SUM0

Description:

Implements the Sum0 transformation function used in the SHA2-256 and SHA2-224 hash function. Only the low order 32 bits of Rs1 are operated on. The 32-bit result is sign extended to the machine width.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
26_7	Ms_3	\sim_3		VN_4	0_6	26_6		$Rs1_6$		Rd_6		$Opcode_7$			

Operation:

$$Rd = \text{sign extend}(\text{ror32}(Rs1,2) \wedge \text{ror32}(Rs1,13) \wedge \text{ror32}(Rs1,22))$$

Execution Units: crypto**Exceptions:** none

SHA256SUM1

Description:

Implements the Sum1 transformation function used in the SHA2-256 and SHA2-224 hash function. Only the low order 32 bits of Rs1 are operated on. The 32-bit result is sign extended to the machine width.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
26_7	Ms_3	\sim_3		VN_4	0_6	27_6		$Rs1_6$		Rd_6		$Opcode_7$			

Operation:

$$Rd = \text{sign extend}(\text{ror32}(Rs1,6) \wedge \text{ror32}(Rs1,11) \wedge \text{ror32}(Rs1,25))$$

Execution Units: crypto**Exceptions:** none

SHA512SIG0

Description:

Implements the Sigma0 transformation function used in the SHA2-512 hash function.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
26 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	28 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = \text{ror64}(Rs1, 1) \wedge \text{ror64}(Rs1, 8) \wedge (Rs1 \gg 7)$$

Execution Units: crypto**Exceptions:** none

SHA512SIG1

Description:

Implements the Sigma1 transformation function used in the SHA2-512 hash function.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
26 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	29 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = \text{ror64}(Rs1, 19) \wedge \text{ror64}(Rs1, 61) \wedge (Rs1 \gg 6)$$

Execution Units: crypto**Exceptions:** none

SHA512SUM0

Description:

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
26_7	Ms_3	\sim_3	VN_4	0_6	30_6	$Rs1_6$	Rd_6	$Opcode_7$						

Execution Units: crypto

SHA512SUM1

Description:

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
26_7	Ms_3	\sim_3	VN_4	0_6	31_6	$Rs1_6$	Rd_6	$Opcode_7$						

Execution Units: crypto

SM3P0

Description:

P0 transform of SM3 hash function.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
26 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	14 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation

$$Rd = Rs1 \wedge \text{rol}(Rs1,9) \wedge \text{rol}(Rs1,17)$$

Execution Units: crypto

SM3P1

Description:

P1 transform of SM3 hash function.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
26 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	15 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation

$$Rd = Rs1 \wedge \text{rol}(Rs1,15) \wedge \text{rol}(Rs1,23)$$

Execution Units: crypto

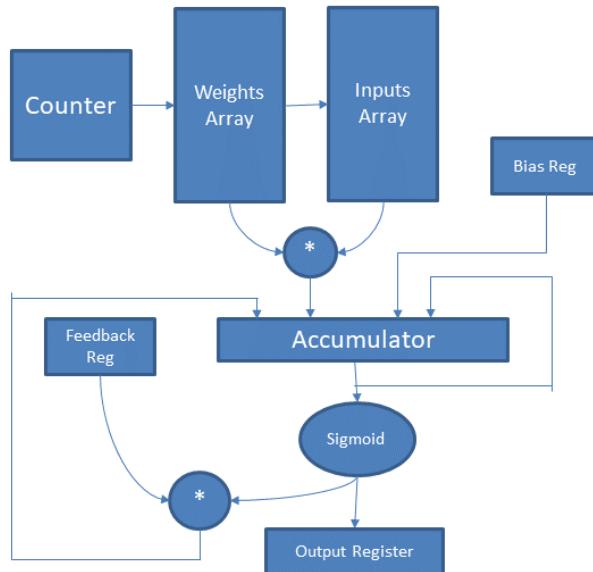
NEURAL NETWORK ACCELERATOR INSTRUCTIONS

OVERVIEW

Included in the ISA are instructions for neural network acceleration. Each neuron is composed of an accumulator that sums the product of weights and inputs and an output activation function. Neurons may be biased with a bias value and may also have feedback from output to input via a feedback constant. The neurons are implemented using 16.16 fixed-point arithmetic. There are 8 neurons in a single layer which may calculate simultaneously. Following is a sketch of the NNA organization. Note that multi-layer networks and additional neurons may be implemented by appropriate software modification of the NNA. The weights and input arrays have a depth of 1024 entries. Not all entries need be used. The number of entries in use is configurable programmatically with the base count and maximum count register using the [NNA_MTBC](#) and [NNA_MTMC](#) instruction.

Several of the NNA instructions allow multiple neurons to be updated at the same time by representing the neuron update list as a bitmask.

Neural Network Accelerator – One Neuron



NNA_MFACT – MOVE FROM OUTPUT ACTIVATION

Description:

Move from activation output register. Move a value from the neuron's activation register output to the destination register Rd. Bits 0 to 3 of Rs1 specify the neuron.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
26 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	10 ₆	Rs1 ₆	Rd ₆	107 ₇						

Clock Cycles: 1**Execution Units:** NNA**Notes:**

NNA_MTBC – MOVE TO BASE COUNT

Description:

Move to base count register. Move the value in Rs1 to the base count register for the neurons identified with a bitmask in Rs2. Each bit of Rs2 represents a neuron. Multiple neurons may be initialized at the same time. Rs1 contains the base count value.

The neuron calculates the activation output using weight and input array entries between the base count and maximum count inclusive.

Manipulating the base count and maximum count registers ease the implementation of multi-layer networks that do not require the use of all array entries.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
45 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	107 ₇						

Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_MTBIAS – MOVE TO BIAS

Description:

Move to bias value. Move the value in Rs1 to the bias register for the neurons identified with a bitmask in Rs2. Each bit of Rs2 represents a neuron. Multiple neurons may be initialized at the same time. Rs1 contains the bias value.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
42 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	107 ₇						

Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_MTFB – MOVE TO FEEDBACK

Description:

Move to feedback constant. Move the value in Rs1 to the feedback constant for the neurons identified with a bitmask in Rs2. Each bit of Rs2 represents a neuron. Multiple neurons may be initialized at the same time. Rs1 contains the feedback constant.

The feedback constant acts to create feedback in the neuron by multiplying the output activation level by the feedback constant and using the result as an input. If no feedback is desired then this constant should be set to zero.

Instruction Format: R2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
43 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	107 ₇						

Clock Cycles: 1**Execution Units:** NNA**Notes:**

NNA_MTIN – MOVE TO INPUT

Description:

Move to input array. Move the value in Rs1 to the input memory cell identified with Rs2. Bits 0 to 15 of Rs2 specify the memory cell address, bits 0 to 31 of Rs3 are a bit mask specifying the neurons to update. Bits 0 to 15 of Rs2 are incremented and stored in Rd.

Instruction Format: R2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
41 ₇	Ms ₃	~ ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	107 ₇						

Clock Cycles: 1**Execution Units: NNA****Notes:**

Multiple neurons may have their inputs updated at the same time with the same value. All the neurons may have the same inputs but the weights for the individual neurons would be different so that a pattern may be recognized.

NNA_MTMC – MOVE TO MAX COUNT

Description:

Move to maximum count register. Move the value in Rs1 to the maximum count register for the neurons identified with a bitmask in Rs2. Each bit of Rs2 represents a neuron. Multiple neurons may be initialized at the same time. Rs1 contains the maximum count value.

The maximum count is the upper limit of inputs and weights to use in the calculation of the activation function. The maximum count should not exceed the hardware table size. The table size is 1024 entries.

The neuron calculates the activation output using weight and input array entries between the base count and maximum count inclusive.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
44 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	107 ₇						

Clock Cycles: 1**Execution Units:** NNA**Notes:**

NNA_MTWT – MOVE TO WEIGHTS

Description:

Move to weights array. Move the value in Rs1 to the weight memory cell identified with Rs2. Bits 0 to 15 or Rs2 specify the memory cell address, bits 0 to 63 of Rs3 are a bit mask specifying the neurons to update. Bits 0 to 15 of Rs2 are incremented and stored in Rd.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
40 ₇	Ms ₃	~ ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	107 ₇						

Clock Cycles: 1**Execution Units:** NNA**Notes:**

NNA_STAT – GET STATUS

Description:

This instruction gets the status of the neurons. There is a bit in Rd for each neuron. A bit will be set if the neuron is finished performing the calculation of the activation function, otherwise the bit will be clear.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
26_7	Ms_3	\sim_3	VN_4	0_6	9_6	$Rs1_6$	Rd_6		107_7					

Clock Cycles: 1**Execution Units:** NNA**Notes:**

NNA_TRIG – TRIGGER CALC

Description:

This instruction triggers an NNA cycle for the neurons identified in the bit mask. The bit mask is contained in register Rs1.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
26_7	Ms_3	\sim_3		VN_4	0_6	8_6		$Rs1_6$		Rd_6		107_7			

Clock Cycles: 1**Execution Units:** NNA**Notes:**

FLOATING-POINT OPERATIONS

PRECISION

Three storage formats are supported for binary floats: 64-bit double precision, 32-bit single precision and 16-bit half precision.

Opcodē ₇	Qualifier	Precision	Store Op
56	H	Half precision	STW
48	S	Single precision	STT
16	D	Double precision	STO
90	Q	Quad precision	2xSTO

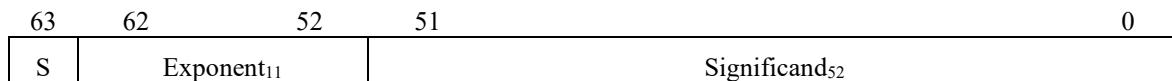
REPRESENTATIONS

BINARY FLOATS

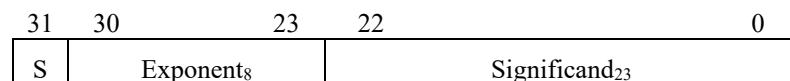
Double Precision, Float:64

The core uses a 64-bit double precision binary floating-point representation.

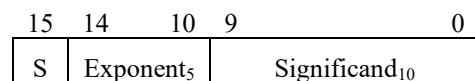
Double Precision



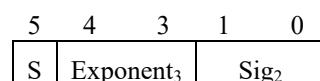
Single Precision, float



Half Precision, float



Six-bit embedded constant float



NANS - NOT A NUMBER

NAN FORMAT

The NaN format is shown for double precision. Other precisions are similar but store fewer IP bits.

63	62	52	51	0
S	Exponent ₁₁		Significand ₅₂	
S	7FFh	Q	Cause ₃	Bit Reversed IP ₄₈

Q: this bit indicates a signalling (0) versus quiet (1) NaN.

Note that the significand must be non-zero for a NaN.

The Cause₃ field is a three-bit code indicating the cause of the NaN.

Cause ₃	Cause
1	Infinity - infinity
2	Infinity / Infinity
3	Zero / zero
4	Infinity * zero
5	Square root of infinity
6	Square root of negative number

The Bit Reversed IP is the value of instruction pointer bits 0 to 47 in bit reversed order, at which the NaN occurred.

NAN BOXING

Lower precision values are ‘NaN boxed’ meaning all the bits needed to extend the value to the width of the register are filled with ones. The sign bit of the number is preserved. Thus, lower precision values encountered in calculations are treated as NaNs.

Example: NaN boxed single precision value.

63	62	52	51	0
S	Exponent ₁₁	Significand ₅₂		
S	7FFh	FFFFFh	Single Precision Float ₃₂	
S	7FFh	FFFFFFFh		Half Precision Float ₁₆

ROUNDING MODES

BINARY FLOAT ROUNDING MODES

Rm3	Rounding Mode
000	Round to nearest ties to even
001	Round to zero (truncate)
010	Round towards plus infinity
011	Round towards minus infinity
100	Round to nearest ties away from zero
101	Reserved
110	Reserved
111	Use rounding mode in float control register

DECIMAL FLOAT ROUNDING MODES

Rm3	Rounding Mode
000	Round ceiling
001	Round floor
010	Round half up
011	Round half even
100	Round down
101	Reserved
110	Reserved
111	Use rounding mode in float control register

SCALAR FLOATING-POINT OPERATIONS

FABS – ABSOLUTE VALUE

Description:

This instruction computes the absolute value of the contents of the source operand Rs1 and places the result in Rd. The sign bit of the value is cleared. No rounding occurs.

Integer Instruction Format: FLT

FABS Rd, Rs1

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
32 ₇	0 ₃	~ ₃	VN ₄	~ ₆	~ ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = \text{Fabs}(Rs1)$$

Execution Units: All FPUs, All SAUs

Clock Cycles: 1

Exceptions: none

Notes:

Opcode ₇		Precision	Clocks
56	H	Half precision parallel	1
57	S	Single precision parallel	1
58	D	Double precision parallel	1
59	Q	Quad precision	1

FADD –FLOAT ADDITION

Description:

Add two source operands Rs1 and Rs2 and place the sum in the destination register Rd. Values are treated as floating-point values.

Supported Operand Sizes:**Instruction Format:** FLT

FADD Rd, Rs1, Rs2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
4 ₇	Ms ₃	Rm ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = (Rs1 + Rs2)$$

Execution Units: All FMA's

Exceptions: none

Notes:

Opcode ₇		Precision	Clocks
56	H	Half precision	8
57	S	Single precision	8
58	D	Double precision	8
59	Q	Quad precision	

FCLASS – CLASSIFY VALUE

Description:

FCLASS classifies the value in register Rs1 and returns the information as a bit vector in the integer register Rd.

Integer Instruction Format: F3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
62_7	Ms_3	\sim_3		VN_4	0_6	\sim_6		$Rs1_6$	Rd_6		$Opcode_7$				

Bit in Rd	Meaning
0	1 = negative infinity
1	1 = negative number
2	1 = negative subnormal number
3	1 = negative zero
4	1 = positive zero
5	1 = positive subnormal number
6	1 = positive number
7	1 = positive infinity
8	1 = signalling nan
9	1 = quiet nan
10 to 62	not used
63	1 = negative, 0 = positive number

FCMP - COMPARISON

Description:

Compare two source operands Rs1, Rs2 and place the result in the destination register Rd. The result is a vector identifying the relationship between the two source operands as floating-point values. This instruction may compare against lower precision immediate values to conserve code space. The source operands are floating-point values, the destination operand is an integer. No rounding occurs.

Instruction Format: FLT

FCMP Rd, Rs1, Rs2

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
13 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇							

Opcode ₇		Precision	Clocks
56	H	Half precision	1
57	S	Single precision	1
58	D	Double precision	1
59	Q	Quad precision	

Operation:

Rd = Rs1 ? Rs2

Clock Cycles: 1

Execution Units: All Integer SAUs, all FPUs

Exceptions: none

Rd Bit ₄	Mnem.	Meaning	Test
0	EQ	equal	!nan & eq
1	NE	not equal	!eq
2	LT	Less than	Lt & (!nan & !inf & !eq)
3	LE	Less than or equal	Eq (Lt & !nan)
4	GE	greater than or equal	Eq (!nan & !lt & !inf)
5	GT	greater than	!nan & !eq & !lt & !inf
6	GL	Greater than or less than	!nan & (!eq & !inf)
7	OR	Greater than less than or equal / ordered	!nan
8		reserved	
9	UGL	Unordered or greater than or less than	Nan !eq
10	ULT	Unordered or less than	Nan (!eq & lt)
11	ULE	unordered less than or equal	Nan (eq lt)

12	UGE	Unordered or greater than or equal	Nan (!lt eq)
13	UGT	Unordered or greater than	Nan (!eq & !lt & !inf)
14		reserved	
15	UN	Unordered	Nan

FCONST – LOAD FLOAT CONSTANT

Description:

This instruction loads a constant from the constant ROM and places the value in Rd.

Integer Instruction Format: R1

FCONST Rd, N

63	57	56	54	53	52 50	49 47	46	43	42	41	34	33	32	25	24	23	16	15	14	7	6	0
? ₇	Pr ₃	~	~ ₃	0 ₃	~ ₄	~	l ₈	~	4 ₈	~	N ₈	Nt	Rt ₈	Opc ₇								

Clock Cycles: 1

Operation:

$$Ft = FConst[N]$$

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

N ₆	Binary64	Decimal	
0	3fe0000000000000	0.5	
1	3ff0000000000000	1.0	
2	4000000000000000	2.0	
3	3ff8000000000000	1.5	
4	0x5FE6EB50C7B537A9		Lomont reciprocal square root magic
21			
22			
23			
57	7FF0000000000000		infinity
58	7FF0000000000001		Nan – infinity - infinity
59	7FF0000000000002		Nan – infinity / infinity
60	7FF0000000000003		Nan – zero / zero
61	7FF0000000000004		Nan – infinity * zero
62	7FF0000000000005		Nan – square root of infinity
63	7FF0000000000006		Nan – square root of negative

FCVTD2Q – CONVERT DOUBLE TO QUAD PRECISION

Description:

This instruction converts the contents of the source operand to the equivalent of a quad precision value and places the result in Rd.

Integer Instruction Format: R1

FCVTD2Q Rd, Rs1

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
43 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	90 ₇						

Clock Cycles: 1

Operation:

$$Rd = \text{Cvt}(Rs1, \text{double})$$

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FCVTH2D – CONVERT HALF TO DOUBLE PRECISION

Description:

This instruction converts the contents of the source operand to the equivalent of a double precision value and places the result in Rd. No rounding occurs.

Integer Instruction Format: R1

FCVTH2D Rd, Rs1

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
45 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	16 ₇						

Clock Cycles: 1

Operation:

$$Rd = \text{Cvt}(Rs1, \text{half})$$

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FCVTQ2D – ROUND QUAD TO DOUBLE PRECISION

Description:

This instruction rounds the contents of the source operand to the equivalent of a double precision value and places the result in Rd. This instruction may be used in preparation for a store.

Integer Instruction Format: R1

FCVTQ2D Rd, Rs1

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
50 ₇	Ms ₃	Rm ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	16 ₇						

Clock Cycles: 2

Operation:

$$Rd = \text{Round}(Rs1, \text{double})$$

Execution Units: FPU #0

Clock Cycles: 2

Exceptions: none

Notes:

FCVTQ2H – ROUND QUAD TO HALF PRECISION

Description:

This instruction rounds the contents of the source operand to the equivalent of a half precision value and places the result in Rd. Note the register continues to contain a quad precision value. This instruction may be used in preparation for a store.

Integer Instruction Format: R1

FCVTQ2H Rd, Rs1

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
48 ₇	Ms ₃	Rm ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	90 ₇						

Clock Cycles: 1

Operation:

$$Rd = \text{Round}(Rs1, \text{half})$$

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FCVTQ2S – ROUND QUAD TO SINGLE PRECISION

Description:

This instruction rounds the contents of the source operand to the equivalent of a single precision value and places the result in Rd. Note the register continues to contain a quad precision value. This instruction may be used in preparation for a store.

Integer Instruction Format: R1

FCVTQ2S Rd, Rs1

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
49 ₇	Ms ₃	Rm ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	90 ₇						

Clock Cycles: 1

Operation:

$$Rd = \text{Round}(Rs1, \text{single})$$

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FCVTS2D – CONVERT SINGLE TO DOUBLE PRECISION

Description:

This instruction converts the contents of the source operand to the equivalent of a double precision value and places the result in Rd. No rounding occurs.

Integer Instruction Format: R1

FCVTS2D Rd, Rs1

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
41 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	58 ₇						

Clock Cycles: 1

Operation:

$$Rd = \text{Cvt}(Rs1, \text{single})$$

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FCVTS2Q – CONVERT SINGLE TO QUAD PRECISION

Description:

This instruction converts the contents of the source operand to the equivalent of a quad precision value and places the result in Rd. No rounding occurs.

Integer Instruction Format: R1

FCVTS2Q Rd, Rs1

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
42 ₇	Ms ₃	Rm ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	90 ₇						

Clock Cycles: 1

Operation:

$$Rd = \text{Cvt}(Rs1, \text{single})$$

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FCX – CLEAR FLOATING-POINT EXCEPTIONS

Description:

This instruction clears floating point exceptions. The Exceptions to clear are identified as the bits set in register Rs1 of the instruction. Rs1 may also be a six-bit constant, controlled by the Ms₃ field of the instruction.

Instruction Format: EX

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
3 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	125 ₇						

Execution Units: All Floating Point

Operation:**Exceptions:**

Bit	Exception Enabled
0	global invalid operation clears the following: <ul style="list-style-type: none">- division of infinities- zero divided by zero- subtraction of infinities- infinity times zero- NaN comparison- division by zero
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	summary exception

FDIV –FLOAT DIVISION

Description:

Divide two source operands Rs1 by Rs2 and place the quotient in the destination register Rd. All registers values are treated as floating-point values.

Supported Operand Sizes:**Instruction Format:** FLT

FDIV Rd, Rs1, Rs2

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
7 ₇	Ms ₃	Rm ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇							

Operation:

$$Rd = Rs1 / Rs2$$

Clock Cycles:

Execution Units: All FPUs

Exceptions: none

Notes:

This instruction is currently implemented as a macro instruction.

Opcode ₇		Precision	Clocks
56	H	Half precision	10
57	S	Single precision	30
58	D	Double precision	46
59	Q	reserved	

FDP –FLOAT DOT PRODUCT

Description:

Multiply two pairs of source operands, add the products and place the result in the destination register.

This instruction may not be supported in all versions of the core.

Instruction Format: FLT4**FDP Rd, Rs1, Rs2, Rs3, Rs4**

47	46	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
~	Rs4 ₆	Ms ₃	Rm ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇							

Opcode ₇		Precision	Clocks
96	H	Half precision	8
97	S	Single precision	8
98	D	Double precision	8
99	Q	Quad precision	

Operation:

$$Rd = (Rs1 * Rs2) + (Rs3 * Rs4)$$

Clock Cycles: 8**Execution Units:****Exceptions:** none**Notes:**

FDX – DISABLE FLOATING POINT EXCEPTIONS

Description:

This instruction disables floating point exceptions. The Exceptions disabled are identified as the bits set in register Rs1 of the instruction. Rs1 may also be a six-bit constant, controlled by the Ms₃ field of the instruction.

Instruction Format: EX

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
4 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	125 ₇						

Execution Units: All Floating Point

Operation:**Exceptions:**

Bit	Exception Disabled
0	invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

FEX – ENABLE FLOATING POINT EXCEPTIONS

Description:

This instruction enables floating point exceptions. The Exceptions enabled are identified as the bits set in register Rs1 of the instruction. Rs1 may also be a six-bit constant, controlled by the Ms₃ field of the instruction.

Instruction Format: EX

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
5 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	125 ₇						

Execution Units: All Floating Point

Operation:**Exceptions:**

Bit	Exception Enabled
0	invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

FINITE – NUMBER IS FINITE

Description:

Test the value in Rs1 to see if it's a finite number and return 1 or 0 in register Rd.

Integer Instruction Format: F3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
47 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Clock Cycles: 1**Execution Units:** Floating Point**Example:**

ffinite Rd, Rs1

FMA –FLOAT MULTIPLY AND ADD

Description:

Multiply two source operands Rs1 and Rs2, add a third operand Rs3 and place the result in the destination register Rd. All register values are treated as floating-point values.

The Ms₃ field indicates where to use a seven-bit float-point constant in place of the corresponding source register. The constant contains a sign bit, three-bit exponent, and three-bit significand. The constant will be expanded to the required precision for the operation.

Alternate Mnemonics: FNMA, FMS, FNMS

Instruction Format: F3

FMA Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
1 ₇	Ms ₃	Rm ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Opcode ₇		Precision	Clocks
96	H	Half precision	8
97	S	Single precision	8
98	D	Double precision	8
99	Q	Quad precision	

Operation:

$$Rd = \pm (\pm Rs1 * \pm Rs2 \pm Rs3)$$

Execution Units: All FPUs

Exceptions: none

Notes:

FMAX –FLOAT MAXIMUM VALUE

Description:

Compare two source operands Rs1, Rs2 and place the maximum in the destination register Rd. Values are treated as floating-point values.

Supported Operand Sizes:**Instruction Format: FLT**

FMAX Rd, Rs1, Rs2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
3 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

If Rs1 is a signalling NaN or Rs2 is a signalling Nan
Rd = quiet NaN
Else If Rs1 is a quiet NaN and Rs2 is a quiet NaN
Rd = quiet NaN
Else if Rs1 is a quiet NaN and Rs2 is not a NaN
Rd = Rs2
Else if Rs1 is not a NaN and Rs2 is a quiet NaN
Rd = Rs1
Else if Rs1 < Rs2
Rd = Rs2
Else
Rd = Rs1

Execution Units: All FPU's

Exceptions: none

Notes:

Opcode ₇		Precision	Clocks
56	H	Half precision	1
57	S	Single precision	1
58	D	Double precision	1
59	Q	Quad precision	

FMIN –FLOAT MINIMUM VALUE

Description:

Compare two source operands Rs1, Rs2 and place the minimum in the destination register Rd. Values are treated as floating-point values.

Supported Operand Sizes:**Instruction Format: FLT**

FMIN Rd, Rs1, Rs2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
2 ₇	Ms ₃	Rm ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

If Rs1 is a signalling NaN or Rs2 is a signalling Nan
Rd = quiet NaN

Else If Rs1 is a quiet NaN and Rs2 is a quiet NaN
Rd = quiet NaN

Else if Rs1 is a quiet NaN and Rs2 is not a NaN
Rd = Rs2

Else if Rs1 is not a NaN and Rs2 is a quiet NaN
Rd = Rs1

Else if Rs1 < Rs2
Rd = Rs1

Else
Rd = Rs2

Execution Units: All FPU's

Exceptions: none

Notes:

Opcode ₇		Precision	Clocks
56	H	Half precision	1
57	S	Single precision	1
58	D	Double precision	1
59	Q	Quad precision	

FMS –FLOAT MULTIPLY AND SUBTRACT

Description:

Multiply two source operands, subtract a third operand and place the result in the destination register. All register values are treated as quad precision floating-point values. This instruction is an alternate mnemonic for the [FMA](#) instruction where the Rs3 register is negated (VN[3]).

Instruction Format: F3**FMS Rd, Rs1, Rs2, Rs3**

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
1 ₇	Ms ₃	Rm ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = Rs1 * Rs2 - Rs3$$

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 floating-point values.

Alternate Mnemonics: FNMUL

Instruction Format: FLT

FMUL Rd, Rs1, Rs2

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
6 ₇	Ms ₃	Rm ₃	VN ₄		Rs3 ₆	Rs2 ₆		Rs1 ₆		Rd ₆		Opcode ₇			

Operation:

$$Rd = Rs1 * Rs2$$

Clock Cycles: 8

Execution Units: All FPUs

Exceptions: none

Notes:

FNEG – NEGATIVE VALUE

Description:

This instruction computes the negative value of the contents of the source operand and places the result in Rd. The sign bit of the value is inverted. No rounding occurs.

Integer Instruction Format: FLT1**FNEG Rd, Rs1**

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
33 ₇	Ms ₃	Rm ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = \text{Fneg}(Rs1)$$

Execution Units: All FPUs, All ALUs**Clock Cycles:** 1**Exceptions:** none**Notes:**

FNMA –FLOAT NEGATE MULTIPLY AND ADD

Description:

Multiply two source operands, add a third operand and place the negative of the result in the destination register. All register values are treated as floating-point values. This instruction is an alternate mnemonic for the [FMA](#) instruction where the Rd register is negated (VN[0]).

Instruction Format: FLT3**FNMA Rd, Rs1, Rs2, Rs3**

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
1 ₇	Ms ₃	Rm ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = -(Rs1 * Rs2 + Rs3)$$

Clock Cycles: 8**Execution Units:** All FPUs**Exceptions:** none**Notes:**

FNMS –FLOAT NEGATE MULTIPLY AND SUBTRACT

Description:

Multiply two source operands, subtract a third operand and place the negative of the result in the destination register. All register values are treated as quad precision floating-point values. This instruction is an alternate mnemonic for the [FMA](#) instruction where the Rs3 and Rd registers are negated.

Instruction Format: FLT3**FNMS Rd, Rs1, Rs2, Rs3**

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
1 ₇	Ms ₃	Rm ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = -(Rs1 * Rs2 - Rs3)$$

Clock Cycles: 8**Execution Units:** All FPUs**Exceptions:** none**Notes:**

FRES – FLOATING POINT RECIPROCAL ESTIMATE

Description:

Estimates the reciprocal of the floating-point number in register Rs1 and place the result into destination register Rd.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
55_7	Ms_3	Est_3		VN_4	0_6	\sim_6		$Rs1_6$	Rd_6		$Opcode_7$				

Est₃	Bits	Clocks
0	8	2
1	16	22
2	32	38
3	53	54

Operation:

$$Rd = fres(Rs1)$$

Execution Units: Floating Point**Notes:**

This function is currently micro-coded.

FRM – SET FLOATING POINT ROUNDING MODE

Description:

This instruction sets the rounding mode bits in the floating-point control register (FPCSR). The rounding mode bits are set to contents of register Rs1.

Instruction Format: F3

47	41	40	38	37	35	34	28	27	21	20	14	13	7	6	0
63_7	Ms ₃		\sim_3	0 ₇		0 ₇	$Rs1_7$		Rd ₇	Opcode ₇					

Execution Units: All Floating Point

Operation:

FPSCR.RM = Rs1

FRSQRTE – FLOAT RECIPROCAL SQUARE ROOT ESTIMATE

Description:

Estimate the reciprocal of the square root of the number in register Rs1 and place the result into destination register Rd.

Instruction Format: FLT

47	41	40	38	37	35	34	28	27	21	20	14	13	7	6	0
54_7	Ms_3	Est_3		0_7		0_7		$Rs1_7$		Rd_7		$Opcode_7$			

Est_3	Bits	Clocks
0	9	46
1	17	70
2	34	94
3	68	119

Execution Units: Floating Point**Notes:**

Taking the reciprocal square root of a negative number or of infinity results in a Nan output.

FSCALEB –SCALE EXPONENT

Description:

Add the source operand Rs2 to the exponent of operand Rs1 and place the result in the destination register Rd. The second source operand is an integer value. No rounding occurs.

Instruction Formats:**FSCALEB Rd, Rs1, Rs2**

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
20 ₇	Ms ₃	~ ₃	V ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:**Clock Cycles:**

Execution Units: All FPUs

Exceptions: none

Notes:

FSEQ – FLOAT SET IF EQUAL

Description:

Compares two source operands Rs1 and Rs2 for equality and places the result in the destination register Rd. The result is an integer Boolean true or false. Positive and negative zero are considered equal. For FSEQ if either operand is a NaN zero the result is false. No rounding occurs.

Instruction Formats:

FSEQ Rd, Rs1, Rs2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
8 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = Rs1 == Rs2 ? 1 : 0$$

Clock Cycles: 1

Execution Units: All FPU's, ALL SAUs

Exceptions: none

Notes:

FSGNJ – FLOAT SIGN INJECT

Description:

Copy the sign of Rs1 and the exponent and significand of Rs2 into the destination register Rd. No rounding occurs.

Instruction Format:

FSGNJ Rd, Rs1, Rs2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
16 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = \{Rs1.\text{sign}, Rs2.\text{exp}, Rs2.\text{sig}\}$$

Clock Cycles: 1

Execution Units: All FPUs, All ALUs

Exceptions: none

Notes:

FSGNIN – FLOAT NEGATIVE SIGN INJECT

Description:

Copy the negative of the sign of Rs1 and the exponent and significand of Rs2 into the destination register Rd. No rounding occurs.

Instruction Format:

FSGNIN Rd, Rs1, Rs2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
17 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = \{\sim R1.sign, R2.exp, R2.sig\}$$

Clock Cycles: 1

Execution Units: All FPUs, All ALUs

Exceptions: none

Notes:

FSGNJPX – FLOAT SIGN INJECT XOR

Description:

Copy the xor of the sign of Rs1 and Rs2 and the exponent and significand of Rs2 into the destination register Rd. No rounding occurs.

Instruction Format:

FSGNJPX Rd, Rs1, Rs2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
18 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = \{Rs1.\text{sign} \wedge Rs2.\text{sign}, Rs2.\text{exp}, Rs2.\text{sig}\}$$

Clock Cycles: 1

Execution Units: All FPU's

Exceptions: none

Notes:

FSIG – GET SIGNIFICAND OF NUMBER

Description:

This instruction provides the significand of a floating-point number contained in a general-purpose register Rs1 as a zero extended result. The hidden bit of the floating-point number remains hidden.

Integer Instruction Format: F3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
39 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	Opcode ₇							

Clock Cycles: 1**Execution Units:** All Floating Point**Operation:**

$$Rd = \text{significand of } (Rs1)$$

FSIGMOID – SIGMOID APPROXIMATE

Description:

This function uses a 1024 entry 32-bit precision lookup table with linear interpolation to approximate the logistic sigmoid function in the range -8.0 to +8.0. Outside of this range 0.0 or +1.0 is returned. The sigmoid output is between 0.0 and +1.0. The value of the sigmoid for register Rs1 is returned in register Rd as a 64-bit double precision floating-point value.

The value is an approximation which is not rounded.

Instruction Format: FLT

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
80 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Clock Cycles: 5

Execution Units: FPU #0 only

FSIGN – SIGN OF NUMBER

Description:

This instruction provides the sign of a floating-point number contained in a general-purpose 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: F3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
38 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Clock Cycles: 1**Execution Units:** All Floating Point**Operation:**

$$Rd = \text{sign of } (Rs1)$$

FSINCOS – FLOAT SINE AND COSINE

Description:

This instruction computes the sine and cosine values of the contents of the source operand and places the result in Rd (sine) and Rs3 (cosine).

Integer Instruction Format: R1

FSINCOS Rd, Rs1

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
64 ₇	Ms ₃	Rm ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = \sin(Rs1)$$

$$Rs3 = \cos(Rs1)$$

Execution Units: TRIG

Exceptions: none

Notes:

Precision (Bits)		Precision	Clocks
16	H	Half precision	24
32	S	Single precision	36
64	D	Double precision	72
128	Q	Quad precision	136

FSLE – FLOAT SET IF LESS THAN OR EQUAL

Description:

Compares two source operands Rs1 and Rs2 for less than or equal and places the result in the destination register Rd. The result is a Boolean true or false. Positive and negative zero are considered equal. For FSLE if either operand is a NaN zero the result is false. No rounding occurs. This instruction may also test for greater than or equal by swapping operands.

Instruction Format:

FSLE Rd, Rs1, Rs2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
11 ₇	Ms ₃	Rm ₃	VN ₄	~ ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = \text{Rs1} < \text{Rs2} ? 1 : 0$$

Clock Cycles: 1**Execution Units:** All FPU's**Exceptions:** none**Notes:**

FSLT – FLOAT SET IF LESS THAN

Description:

Compares two source operands Rs1 and Rs2 for less than and places the result in the destination register Rd. The result is a Boolean true or false. Positive and negative zero are considered equal. For FSLT if either operand is a NaN zero the result is false. No rounding occurs. This instruction may also test for greater than by swapping operands.

Instruction Formats:

FSLT Rd, Rs1, Rs2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
10 ₇	Ms ₃	Rm ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Operation:

$$Rd = \text{Rs1} < \text{Rs2} ? 1 : 0$$

Clock Cycles: 1**Execution Units:** All FPU's**Exceptions:** none**Notes:**

FSNE – FLOAT SET IF NOT EQUAL

Description:

Compares two source operands Rs1 and Rs2 for equality and places the result in the destination register Rd. The result is a Boolean true or false. Positive and negative zero are considered equal. 16, 32, 64, and 128-bit immediates are supported. No rounding occurs.

Instruction Format:

FSNE Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
9 ₇	Ms ₃	Rm ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Opcode ₇		Precision	Clocks
56	H	Half precision	1
57	S	Single precision	1
58	D	Double precision	1
59	Q	Quad precision	

Operation:

$$Rd = Rs1 \neq Rs2 ? 1 : 0$$

Execution Units: All FPU's

Exceptions: none

Notes:

FSQRT – FLOATING POINT SQUARE ROOT

Description:

Take the square root of the floating-point number in register Rs1 and place the result into destination register Rd. The sign bit (bit 63) of the register is set to zero. This instruction can generate NaNs.

Instruction Format: F3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
40 ₇	Ms ₃	Rm ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Opcode ₇		Precision	Clocks
56	H	Half precision	
57	S	Single precision	
58	D	Double precision	72
59	Q	Quad precision	

Operation:

$$Rd = \text{fsqrt}(Rs1)$$

Execution Units: Floating Point

FSUB –FLOAT SUBTRACTION

Description:

Subtract two source operands and place the difference in the destination register. This is an alternate mnemonic for [FADD](#) where Rs2 is negated.

Supported Operand Sizes:**Instruction Format:** FLT

FSUB Rd, Rs1, Rs2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
4 ₇	Ms ₃	Rm ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇						

VN bit 2 must be set

Opcode ₇		Precision	Clocks
56	H	Half precision	8
57	S	Single precision	8
58	D	Double precision	8
59	Q	Quad precision	

Operation:

$$Rd = Rs1 - Rs2$$

Clock Cycles: 8

Execution Units: All FPUs

Exceptions: none

Notes:

FTOI – FLOAT TO INTEGER

Description:

This instruction converts a floating-point double value to an integer value.

Instruction Format: F3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
34_7	Ms_3	\sim_3	VN_4		0_6		\sim_6		$Rs1_6$	Rd_6		$Opcode_7$		

Clock Cycles: 2**Execution Units: All Floating Point**

FTRUNC – TRUNCATE VALUE

Description:

The FTRUNC instruction truncates off the fractional portion of the number leaving only a whole value. For instance, ftrunc(1.5) equals 1.0. Ftrunc does not change the representation of the number. To convert a value to an integer in a fixed-point representation see the FTOI instruction.

Instruction Format: F3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
53 ₇	Ms ₃	Rm ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	Opcode ₇							

Opcode ₇		Precision	Clocks
56	H	Half precision	1
57	S	Single precision	1
58	D	Double precision	1
59	Q	Quad precision	

Execution Units: All FPUs

FTX – TRIGGER FLOATING POINT EXCEPTIONS

Description:

This instruction triggers floating point exceptions. The Exceptions to trigger are identified as the bits set in register Rs1 of the instruction. Rs1 may also be a six-bit constant, controlled by the Ms₃ field of the instruction.

Instruction Format: EX

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
2 ₇	Ms ₃	~ ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	125 ₇						

Execution Units: All Floating Point

Operation:**Exceptions:**

Bit	Exception Enabled
0	global invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

ISNAN – IS NOT A NUMBER

Description:

Test the value in Rs1 to see if it's a nan (not a number) and return true or false in register Rd.

Instruction Format: F3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
46 ₇	Ms ₃	Rm ₃	VN ₄	0 ₆	~ ₆	Rs1 ₆	Rd ₆	Opcode ₇						

Clock Cycles: 1**Execution Units:** Floating Point**Example:**

fisnan Rd, Rs1

ITOF – INTEGER TO FLOAT

Description:

This instruction converts an integer value to a floating-point representation.

Instruction Format: F3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
35_7	Ms_3	\sim_3	VN_4		0_6		\sim_6		$Rs1_6$	Rd_6		$Opcode_7$		

Clock Cycles: 2**Execution Units:** All FPUs

VECTOR FLOATING-POINT OPERATIONS

VFABS – ABSOLUTE VALUE

Description:

This instruction computes the absolute value of the contents of the source vector operand Rs2 and places the result in vector Rd. The sign bit of the value is cleared. No rounding occurs. Results are masked into the destination register using Rs3. For each set bit in Rs3 the destination element will be updated, otherwise the destination element will remain the same.

The precision of the operation is determined by the value of the Float field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the Float field of the [U_VLEN](#) register.

Integer Instruction Format: VFLT

VFABS Rd, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
32 ₇	Ms ₃	Rm ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	117 ₇						

47	41	40 38	37 35	34 28	27 21	20 14	13 7	6	0
32 ₇	0 ₃	~ ₃	Rs3 ₇	Rs2 ₇	~ ₇	Rd ₇	117 ₇		

Operation:

For each element:

$$Rd = \text{Fabs}(Rs1)$$

Execution Units: All FPUs, All SAUs

Clock Cycles: 1 per element

Exceptions: none

Notes:

VFADD –FLOAT ADDITION

Description:

Add two source vector operands Rs1 and Rs2 and place the sum in the destination vector register Rd. Values are treated as floating-point values. Results are masked into the destination register using Rs3. For each set bit in Rs3 the destination element will be updated, otherwise the destination element will remain the same.

The precision of the operation is determined by the value of the Float field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the Float field of the [U_VLEN](#) register.

Supported Operand Sizes:**Instruction Format:** VFLT

VFADD Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
4 ₇	Ms ₃	Rm ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	117 ₇						

Operation:

For each element where the corresponding bit of Rs3 is set,

$$Rd = (Rs1 + Rs2)$$

Execution Units: All FMA's**Exceptions:** none**Notes:**

VFCMP - COMPARISON

Description:

Compare two source vector operands Rs1, Rs2 and place the result in the destination vector register Rd. The result is a bit vector for each element identifying the relationship between the two source operands as floating-point values. The source operands are floating-point values, the destination operand is an integer. Results are masked into the destination register using Rs3. For each set bit in Rs3 the destination element will be updated, otherwise the destination element will remain the same.

The precision of the operation is determined by the value of the Float field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the Float field of the [U_VLEN](#) register.

No rounding occurs.

Instruction Format: VFLT

VFCMP Rd, Rs1, Rs2, Rs3

47	41	40	38	37	35	34	31	30	25	24	19	18	13	12	7	6	0
13 ₇	Ms ₃	Rm ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	117 ₇									

Operation:

For each element where Rs3 bit is set:

$$Rd = Rs1 ? Rs2$$

Clock Cycles: 1

Execution Units: All SAUs, all FPUs

Exceptions: none

Rd Bit ₄	Mnem.	Meaning	Test
0	EQ	equal	!nan & eq
1	NE	not equal	!eq
2	LT	Less than	Lt & (!nan & !inf & !eq)
3	LE	Less than or equal	Eq (lt & !nan)
4	GE	greater than or equal	Eq (!nan & !lt & !inf)
5	GT	greater than	!nan & !eq & !lt & !inf
6	GL	Greater than or less than	!nan & (!eq & !inf)
7	OR	Greater than less than or equal / ordered	!nan
8		reserved	
9	UGL	Unordered or greater than or less than	Nan !eq
10	ULT	Unordered or less than	Nan (!eq & lt)
11	ULE	unordered less than or equal	Nan (eq lt)
12	UGE	Unordered or greater than or equal	Nan (!lt eq)

13	UGT	Unordered or greater than	Nan (!eq & !lt & !inf)
14		reserved	
15	UN	Unordered	Nan

VFCVTS2D – CONVERT SINGLE TO DOUBLE PRECISION

Description:

This instruction converts the contents of the source vector operand Rs1 to the equivalent of a double precision value and places the result in vector register Rd. No rounding occurs.

Integer Instruction Format: VFLT

VFCVTS2D Rd, Rs1, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
41 ₇	Ms ₃	~ ₃	VN ₄	Rs3 ₆	~ ₆	Rs1 ₆	Rd ₆	58 ₇						

Clock Cycles: 1

Operation:

For each element where Rs3 bit is set:

$$Rd = \text{Cvt}(Rs1, \text{single})$$

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

VFDP – FLOAT DOT PRODUCT

Description:

Multiply two pairs of source vector operands (Rs1, Rs2) and (Rs3, Rs4), add the products and place the result in the destination vector register Rd. This operation may not be masked. If a mask is needed the result should be placed in a temporary register, then a masked vector operation performed with the temporary result.

The precision of the operation is determined by the value of the Float field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the Float field of the [U_VLEN](#) register.

Instruction Format: VFLT4

FDP Rd, Rs1, Rs2, Rs3, Rs4

47	41	40	38	37	35	34	28	27	21	20	14	13	7	6	0
Rs4 ₇	Ms ₃	Rm ₃		Rs3 ₇		Rs2 ₇		Rs1 ₇		Rd ₇		118 ₇			

Operation:

For each element:

$$Rd = (Rs1 * Rs2) + (Rs3 * Rs4) \quad ; \text{ no masking of operation occurs}$$

Clock Cycles: 8 per element

Execution Units:

Exceptions: none

Notes:

VFMA –FLOAT MULTIPLY AND ADD

Description:

Multiply two source operands Rs1 and Rs2, add a third operand Rs3 and place the result in the destination register Rd. All register values are treated as floating-point values.

The Ms₃ field indicates where to use a seven-bit float-point constant in place of the corresponding source register. The constant contains a sign bit, three-bit exponent, and three-bit significand. The constant will be expanded to the required precision for the operation.

This operation is unmasked.

Instruction Format: F3

FMA Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
1 ₇	Ms ₃	Rm ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	117 ₇						

Operation:

$$Rd = Rs1 * Rs2 + Rs3$$

Execution Units: All FPUs

Exceptions: none

Notes:

VFMAX –FLOAT MAXIMUM VALUE

Description:

Compare two source vector operands Rs1, Rs2 and place the maximum in the destination vector register Rd. Values are treated as floating-point values. Results are masked into the destination register using Rs3. For each set bit in Rs3 the destination element will be updated, otherwise the destination element will remain the same.

The precision of the operation is determined by the value of the Float field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the Float field of the [U_VLEN](#) register.

Supported Operand Sizes:

Instruction Format: VFLT

VFMAX Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
3 ₇	Ms ₃	~ ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	117 ₇						

Operation:

For each element

If Rs1 is a signalling NaN or Rs2 is a signalling Nan
Rd = quiet NaN
Else If Rs1 is a quiet NaN and Rs2 is a quiet NaN
Rd = quiet NaN
Else if Rs1 is a quiet NaN and Rs2 is not a NaN
Rd = Rs2
Else if Rs1 is not a NaN and Rs2 is a quiet NaN
Rd = Rs1
Else if Rs1 < Rs2
Rd = Rs2
Else
Rd = Rs1

Clock Cycles: 1 per element

Execution Units: All FPU's

Exceptions: none

Notes:

VFMASK –FLOAT MASK

Description:

Copy source operand vector register Rs1 to destination operand vector register Rd under the guidance of a mask contained in scalar register Rs3.

For each set bit in Rs3 the destination element will be updated, otherwise the destination element will remain the same.

The precision of the operation is determined by the value of the Float field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the Float field of the [U_VLEN](#) register.

This instruction is useful for vector operations that do not support a mask (VFDP).

Supported Operand Sizes:

Instruction Format: VFLT

VFMASK Rd, Rs1, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
21 ₇	Ms ₃	~ ₃	VN ₄	Rs3 ₆	~ ₆	Rs1 ₆	Rd ₆	117 ₇						

Operation:

For each element where the corresponding bit is set in Rs3:

$$Rd = Rs1$$

Clock Cycles: 1 per element

Execution Units: All FPU's

Exceptions: none

Notes:

VFMIN – FLOAT MINIMUM VALUE

Description:

Compare two source vector operands Rs1, Rs2 and place the minimum in the destination vector register Rd. Values are treated as floating-point values. Results are masked into the destination register using Rs3. For each set bit in Rs3 the destination element will be updated, otherwise the destination element will remain the same.

The precision of the operation is determined by the value of the Float field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the Float field of the [U_VLEN](#) register.

Supported Operand Sizes:**Instruction Format:** VFLT

FMIN Rd, Rs1, Rs2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
2 ₇	Ms ₃	~ ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	117 ₇						

Operation:

For each element:
If Rs1 is a signalling NaN or Rs2 is a signalling Nan
 Rd = quiet NaN
Else If Rs1 is a quiet NaN and Rs2 is a quiet NaN
 Rd = quiet NaN
Else if Rs1 is a quiet NaN and Rs2 is not a NaN
 Rd = Rs2
Else if Rs1 is not a NaN and Rs2 is a quiet NaN
 Rd = Rs1
Else if Rs1 < Rs2
 Rd = Rs1
Else
 Rd = Rs2

Clock Cycles: 1 per element

Execution Units: All FPU's

Exceptions: none

Notes:

VFMUL –FLOAT MULTIPLICATION

Description:

Multiply two source operands and place the product in the destination register. All registers values are treated as floating-point values. Results are masked into the destination register using Rs3. For each set bit in Rs3 the destination element will be updated, otherwise the destination element will remain the same.

The precision of the operation is determined by the value of the Float field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the Float field of the [U_VLEN](#) register.

Instruction Format: F3

FMUL Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
6 ₇	Ms ₃	Rm ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	117 ₇						

Operation:

$$Rd = Rs1 * Rs2$$

Clock Cycles: 8

Execution Units: All FPUs

Exceptions: none

Notes:

VFNEG – NEGATIVE VALUE

Description:

This instruction computes the negative value of the contents of the source operand and places the result in Rd. The sign bit of the value is inverted. No rounding occurs. Results are masked into the destination register using Rs3. For each set bit in Rs3 the destination element will be updated, otherwise the destination element will remain the same.

The precision of the operation is determined by the value of the Float field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the Float field of the [U_VLEN](#) register.

Integer Instruction Format: FLT1

VFNEG Rd, Rs1, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
33 ₇	Ms ₃	~ ₃	VN ₄	Rs3 ₆	~ ₆	Rs1 ₆	Rd ₆	117 ₇						

Operation:

For each element where Rs3 bit is set:

$$Rd = Fneg(Rs1)$$

Execution Units: All FPUs, All SAUs

Clock Cycles: 1 per element

Exceptions: none

NaNs: propagated, not generated

Notes:

VFSCALEB –SCALE EXPONENT

Description:

Add the source operand Rs2 to the exponent of operand Rs1 and place the result in the destination register Rd. The second source operand is an integer value. No rounding occurs. Rs1 and Rd should be vector registers. Rs2 may be either a vector or a scalar register. Results are masked into the destination register using Rs3. For each set bit in Rs3 the destination element will be updated, otherwise the destination element will remain the same.

The precision of the operation is determined by the value of the Float field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the Float field of the [U_VLEN](#) register.

Instruction Formats:

FSCALEB Rd, Rs1, Rs2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
20 ₇	Ms ₃	~ ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	117 ₇						

Operation:

Clock Cycles:

Execution Units: All FPUs

Exceptions: none

Notes:

VFSEQ – FLOAT SET IF EQUAL

Description:

Compares two source vector operands Rs1 and Rs2 for equality and places the result in the destination scalar register Rd. The result is an integer Boolean true or false. Positive and negative zero are considered equal. For FSEQ if either operand is a NaN zero the result is false. No rounding occurs. Results are masked into the destination register using Rs3. For each set bit in Rs3 the destination register bit will be updated, otherwise the destination bit will remain the same.

The precision of the operation is determined by the value of the Float field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the Float field of the [U_VLEN](#) register.

Instruction Formats: VFLT

VFSEQ Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
8 ₇	Ms ₃	Rm ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	117 ₇						

Operation:

For each element where Rs3 bit is set:

$$Rd = Rs1 == Rs2 ? 1 : 0$$

Clock Cycles: 1 per element

Execution Units: All FPU's, ALL SAUs

Exceptions: none

NaN's: a NaN for either operand results in a zero (false) result.

Notes:

VFSGNJ – FLOAT SIGN INJECT

Description:

Copy the sign of vector register Rs1 and the exponent and significand of vector register Rs2 into the destination vector register Rd. No rounding occurs. Results are masked into the destination register using Rs3. For each set bit in Rs3 the destination register element will be updated, otherwise the destination element will remain the same.

The precision of the operation is determined by the value of the Float field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the Float field of the [U_VLEN](#) register.

Instruction Format: VFLT

VFSGNJ Rd, Rs1, Rs2, Rs3

47	41	40	38	37	35	34	31	30	25	24	19	18	13	12	7	6	0
16 ₇	Ms ₃	~ ₃		VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆							117 ₇		

Operation:

For each element where Rs3 bit is set:

$$Rd = \{Rs1.\text{sign}, Rs2.\text{exp}, Rs2.\text{sig}\}$$

Clock Cycles: 1 per element

Execution Units: All FPUs, All SAUs

Exceptions: none

Notes:

VFSGNIN – FLOAT NEGATIVE SIGN INJECT

Description:

Copy the negative of the sign of vector register Rs1 and the exponent and significand of vector register Rs2 into the destination vector register Rd. No rounding occurs. Results are masked into the destination register using Rs3. For each set bit in Rs3 the destination register element will be updated, otherwise the destination element will remain the same.

The precision of the operation is determined by the value of the Float field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the Float field of the [U_VLEN](#) register.

Instruction Format:

VFSGNIN Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
17 ₇	Ms ₃	~ ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	117 ₇						

Operation:

For each element where Rs3 bit is set:

$$Rd = \{\sim R1.\text{sign}, R2.\text{exp}, R2.\text{sig}\}$$

Clock Cycles: 1

Execution Units: All FPUs, All ALUs

Exceptions: none

Notes:

VFSGNJPX – FLOAT SIGN INJECT XOR

Description:

Copy the xor of the sign of Rs1 and Rs2 and the exponent and significand of Rs2 into the destination register Rd. No rounding occurs. Results are masked into the destination register using Rs3. For each set bit in Rs3 the destination register element will be updated, otherwise the destination element will remain the same.

The precision of the operation is determined by the value of the Float field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the Float field of the [U_VLEN](#) register.

Instruction Format:

FSGNJPX Rd, Rs1, Rs2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
18 ₇	Ms ₃	~ ₃	V ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	117 ₇						

Operation:

$$Rd = \{Rs1.\text{sign} \wedge Rs2.\text{sign}, Rs2.\text{exp}, Rs2.\text{sig}\}$$

Clock Cycles: 1

Execution Units: All FPU's

Exceptions: none

Notes:

VFTRUNC – TRUNCATE VALUE

Description:

The FTRUNC instruction truncates off the fractional portion of the number leaving only a whole value. For instance, ftrunc(1.5) equals 1.0. Ftrunc does not change the representation of the number. To convert a value to an integer in a fixed-point representation see the FTOI instruction.

The precision of the operation is determined by the value of the Float field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the Float field of the [U_VLEN](#) register.

Instruction Format: F3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
53 ₇	Ms ₃	Rm ₃	VN ₄	Rs3 ₆	~ ₆	Rs1 ₆	Rd ₆	117 ₇						

Clocks: 1**Execution Units:** All FPUs

VISNAN – IS NOT A NUMBER

Description:

Test the value in Rs1 to see if it's a nan (not a number) and return true or false in register Rd. Results are masked into the destination register using Rs3. For each set bit in Rs3 the destination register element will be updated, otherwise the destination element will remain the same.

The precision of the operation is determined by the value of the Float field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the Float field of the [U_VLEN](#) register.

Instruction Format: F3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
46 ₇	Ms ₃	Rm ₃	VN ₄	Rs3 ₆	~ ₆	Rs1 ₆	Rd ₆	117 ₇						

Clock Cycles: 1

Execution Units: Floating Point

Example:

visnan Rd, Rs1

VITOF – INTEGER TO FLOAT

Description:

This instruction converts an integer value in Rs1 to a floating-point representation in Rd. Results are masked into the destination register using Rs3. For each set bit in Rs3 the destination register element will be updated, otherwise the destination element will remain the same.

The precision of the operation is determined by the value of the Float field of the [U_VELSZ](#) register. The number of elements processed is determined by the value of the Float field of the [U_VLEN](#) register.

Instruction Format: F3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
35_7	Ms_3	\sim_3	V_4	$Rs3_6$	\sim_6	$Rs1_6$	Rd_6	$Opcde_7$						

Clock Cycles: 2 per 64-bits

Execution Units: All FPUs

DECIMAL FLOATING-POINT INSTRUCTIONS

OVERVIEW

Supported operations for decimal floating-point is more limited than the operations available with binary floating-point. Basic operations include add, subtract, multiply and divide.

The core may be built to support decimal floating-point by setting the parameter number of DFP units greater than zero. NDFFU = 1. Currently the core supports only a single unit.

If one is using decimal floating-point one is likely not interested in the best floating-point performance. Hence only a single unit is supported.

PRECISION

Only 128-bit precision is supported for decimal floating point.

It is the author's opinion that the desire for decimal floating-point comes from the decimal precision of many digits. For values that are smaller it makes more sense to use binary floating-point.

REPRESENTATIONS

DECIMAL FLOATS

The core uses a 128-bit densely packed decimal double precision floating-point representation.

127	126	122	121	110	109	0
S	Combo ₅	Exponent ₁₂		Significand ₁₁₀		

Field 'S' contains the sign of the number, one for negative, zero for positive.

The Significand₁₁₀ field stores 34 densely packed decimal digits. One whole digit before the decimal point. Each group of ten bits stores three decimal digits. The Combo₅ field stores an additional decimal digit.

The exponent is a power of ten as a binary number with an offset of 6143. Range is 10^{-6143} to 10^{6144} . It is encoded in the Combo₅ field plus the Exponent₁₂ field.

Combo ₅	
00mmm	mmm = decimal digit 0 to 7, encoded as binary 000b to 111b Exponent begin with 00b
01mmm	Exponent begins with 01b
10mmm	Exponent begins with 10b
1100m	100m – decimal digits of 8, 9, m is LSB of digit, 1000b or 1001b Exponent begins with 00b
1101m	Exponent begins with 01b
1110m	Exponent begins with 10b
11110	Infinity, positive or negative depending on sign bit

11111 0	Quiet NaN – sign bit irrelevant
11111 1	Signalling NaN – sign bit irrelevant

The DPD encoding used is described in:

Densely Packed Decimal Encoding, by Mike Cowlishaw, in
IEE Proceedings – Computers and Digital Techniques, ISSN 1350-2387,
Vol. 149, No. 3, pp102–104, IEE, May 2002

See: <http://speleotrove.com/decimal/DPDecimal.html>

DFADD – ADD REGISTER-REGISTER

Description:

Add two registers Rs1 and Rs2 and place the sum in the destination register Rd. The values are treated as quad precision decimal floating-point values.

Instruction Format: DFLT

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
4 ₇	Ms ₃	Rm ₃	VN ₄	~ ₆	Rs2 ₆	Rs1 ₆	Rd ₆	17 ₇						

Execution Units: All DFPUs**Operation:**

$$Rd = Rs1 + Rs2$$

Exceptions:**Notes:**

DFMUL – MULTIPLY REGISTER-REGISTER

Description:

Multiply two registers and place the product in the destination register. The values are treated as quad precision decimal floating-point values.

Instruction Format: DFLT

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
6 ₇	Ms ₃	Rm ₃	VN ₄	~ ₆	Rs2 ₆	Rs1 ₆	Rd ₆	17 ₇						

Execution Units: All ALU's**Operation:**

$$Rd = Rs1 * Rs2$$

Exceptions:**Notes:**

DFNEG – NEGATE REGISTER

Description:

This instruction negates a quad precision decimal floating point number contained in a general-purpose register. The sign bit of the number is inverted. No rounding takes place.

Integer Instruction Format: R1

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
33_7	Ms_3	\sim_3	VN_4	\sim_6	\sim_6	$Rs1_6$	Rd_6		17_7					

Clock Cycles: 1**Execution Units:** All Floating Point**Operation:**

$$Rd = -Rs1$$

DFSEQ – DECIMAL FLOAT SET IF EQUAL

Description:

Compares two source operands Rs1 and Rs2 for equality and places the result in the destination register Rd. The result is an integer Boolean true or false. Positive and negative zero are considered equal. For DFSEQ if either operand is a NaN zero the result is false. No rounding occurs.

Instruction Formats:

DFSEQ Rd, Rs1, Rs2

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
8_7	Ms_3	Rm_3	VN_4	\sim_6	$Rs2_6$	$Rs1_6$	Rd_6		17_7					

Operation:

$$Rd = Rs1 == Rs2 ? 1 : 0$$

Clock Cycles: 1**Execution Units:** All FPU's, ALL SAUs**Exceptions:** none**Notes:**

DFSIG – GET SIGNIFICAND OF NUMBER

Description:

This instruction provides the significand of a quad precision decimal floating point number contained in a general-purpose register as a 120-bit zero extended result.

Integer Instruction Format: DF3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
39 ₇	Ms ₃	~ ₃	VN ₄	~ ₆	~ ₆	Rs1 ₆	Rd ₆	17 ₇							

Clock Cycles: 1**Execution Units:** All Floating Point**Operation:**

Rd = significand of (Rs1)

DFSUB – ADD REGISTER-REGISTER

Description:

Subtract two registers Rs2 from Rs1 and place the difference in the destination register Rd. The values are treated as quad precision decimal floating-point values.

Instruction Format: DF3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
5 ₇	Ms ₃	Rm ₃	VN ₄	~ ₆	Rs2 ₆	Rs1 ₆	Rd ₆	17 ₇							

Execution Units: All DFPUs**Operation:**

$$Rd = Rs1 - Rs2$$

Exceptions:**Notes:**

DFTOI – DECIMAL FLOAT TO INTEGER

Description:

This instruction converts a decimal floating-point quad value to an integer value.

Instruction Format: DF3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
34 ₇	Ms ₃	Rm ₃	VN ₄	~ ₆	~ ₆	Rs1 ₆	Rd ₆	17 ₇							

Clock Cycles:**Execution Units:** All Floating Point

ITODF – INTEGER TO DECIMAL FLOAT

Description:

This instruction converts an integer value to a quad precision decimal floating point representation.

Instruction Format: DF3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
35_7	Ms_3	Rm_3	VN_4		\sim_6	\sim_6		$Rs1_6$	Rd_6		17_7			

Clock Cycles:

Execution Units: All Floating Point

MEMORY OPERATION INSTRUCTIONS

OVERVIEW

There are floating-point load instructions which automatically convert the precision of the loaded value to quad precision. There are no corresponding floating-point store instructions as stores need to round quad precision values to lower precision.

ADDRESSING MODES

Load and store instructions have just a single addressing mode, scaled indexed addressing.

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*]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18...0}	Rs2 ₆	Rs1 ₆	Rd ₆	70 ₇					

ATOMIC MEMORY OPERATIONS

AMOADD – AMO ADDITION

Description:

Atomically add source operand register Rs3 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 Rs2.

Supported Operand Sizes: .h**Instruction Formats: AMO****AMOADD Rd, Rs3, [Rs1+Rs2]**

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
0 ₇	Ms ₃	Ar ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		92 ₇					

Clock Cycles:

AMOAND – AMO BITWISE ‘AND’

Description:

Atomically bitwise ‘and’ source operand register Rs3 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 index Rs2.

Supported Operand Sizes: .h**Instruction Formats: AMO****AMOAND Rd, Rs3, [Rs1+Rs2]**

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
1 ₇	Ms ₃	Ar ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		92 ₇					

Clock Cycles:

AMOASL – AMO ARITHMETIC SHIFT LEFT

Description:

Atomically shift the contents of memory to the left by one bit 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 scaled index Rs2.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOASL Rd, [Rs1+Rs2]

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
8 ₇	Ms ₃	Ar ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		92 ₇					

Clock Cycles:

AMOEOR – AMO BITWISE EXCLUSIVELY ‘OR’

Description:

Atomically bitwise exclusively ‘or’ source operand register Rs3 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 scaled index Rs2.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOEOR Rd, Rs3, [Rs1+Rs2]

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
3 ₇	Ms ₃	Ar ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		92 ₇					

Clock Cycles:

AMOLSR – AMO LOGICAL SHIFT RIGHT

Description:

Atomically shift the contents of memory to the right by one bit 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 scaled index Rs2.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOLSR Rd, [Rs1+Rs2]

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
9 ₇	Ms ₃	Ar ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		92 ₇					

Clock Cycles:

AMOMAX – AMO MAXIMUM

Description:

Atomically determine the maximum of source operand register Rs3 and a 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 scaled index Rs2.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOMAX Rd, Rs3, [Rs1+Rs2]

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
5 ₇	Ms ₃	Ar ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		92 ₇					

AMOMAXU – AMO UNSIGNED MAXIMUM

Description:

Atomically determine the maximum of source operand register Rs3 and a value from memory and store the result back to memory. Values are treated as unsigned integers. The original value of the memory cell is stored in register Rd. The memory address is the sum of Rs1 and scaled index Rs2.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOMAX Rd, Rs3, [Rs1+Rs2]

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
13 ₇	Ms ₃	Ar ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		92 ₇					

AMOMIN – AMO MINIMUM

Description:

Atomically determine the minimum of source operand register Rs3 and a 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 scaled index Rs2.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOAND Rd, Rs3, [Rs1+Rs2]

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
4 ₇	Ms ₃	Ar ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		92 ₇					

AMOMINU – AMO UNSIGNED MINIMUM

Description:

Atomically determine the minimum of source operand register Rs3 and a value from memory and store the result back to memory. Values are treated as unsigned integers. The original value of the memory cell is stored in register Rd. The memory address is the sum of Rs1 and scaled index Rs2.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOAND Rd, Rs1, [Rs1+Rs2]

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
12 ₇	Ms ₃	Ar ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		92 ₇					

AMOOR – AMO BITWISE ‘OR’

Description:

Atomically bitwise ‘and’ source operand register Rs3 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 scaled index Rs2.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOOR Rd, Rs3, [Rs1+Rs2]

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
2 ₇	Ms ₃	Ar ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	92 ₇						

Clock Cycles:

AMOROL – AMO ROTATE LEFT

Description:

Atomically rotate the contents of memory to the left by one bit 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 scaled index Rs2.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOROL Rd, [Rs1+Rs2]

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
10 ₇	Ms ₃	Ar ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	92 ₇						

Clock Cycles:

AMOROR – AMO ROTATE RIGHT

Description:

Atomically rotate the contents of memory to the right by one bit 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 scaled index Rs2.

Supported Operand Sizes: .h**Instruction Formats: AMO****AMOROR Rd, [Rs1+Rs2]**

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
11 ₇	Ms ₃	Ar ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	92 ₇						

Clock Cycles:

AMOSWAP – AMO SWAP

Description:

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

Supported Operand Sizes: .h**Instruction Formats: AMO****AMOSWAP Rd, Rs3, [Rs1+Rs2]**

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
6 ₇	Ms ₃	Ar ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	92 ₇						

Clock Cycles:

CAS – COMPARE AND SWAP

Description:

If the contents of the addressed memory cell equals the contents of Rs2 then a 64-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 performed by the memory controller.

Instruction Format:

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
0 ₇	Ms ₃	Ar ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		93 ₇					

Operation:

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

Assembler:

```
CAS Rd, Rs2, Rs3, [Rs1]
```

Note:

LOAD OPERATIONS

FLDD RN, <EA> - FLOAT LOAD DOUBLE

Description:

Load register Rd with a double precision value from memory. The memory address is the value in register Rs1 plus the value in register Rs2 scaled by 2^n plus a 19-bit displacement.

The capabilities tag bit of the register is cleared.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18..0}	Rs2 ₆	Rs1 ₆	Rd ₆	70 ₇					LS – load / store

Execution Units: AGEN, MEM

Exceptions:

Notes:

FLDH RN, <EA> - FLOAT LOAD HALF PRECISION

Description:

Load register Rd with a floating-point half precision value from memory. The value is converted to a quad precision format.

The memory address is the value in register Rs1 plus the value in register Rs2 scaled by 2^n plus a 19-bit displacement.

The capabilities tag bit of the register is cleared.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18..0}	Rs2 ₆	Rs1 ₆	Rd ₆	74 ₇					LS – load / store

Execution Units: AGEN, MEM

Exceptions:

Notes:

FLDQ RN, <EA> - FLOAT LOAD QUAD

Description:

Load register Rd with a quad precision value from memory. The memory address is the value in register Rs1 plus the value in register Rs2 scaled by 2^n plus a 19-bit displacement.

The register flags bits are cleared.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18..0}	Rs2 ₆	Rs1 ₆	Rd ₆	78 ₇					LS – load / store

Execution Units: AGEN, MEM

Exceptions:

Notes:

FLDS RN, <EA> - FLOAT LOAD SINGLE PRECISION

Description:

Load register Rd with a floating-point single precision value from memory. The value is one extended to the machine width with the sign bit of the single precision number preserved (see [NaN boxing](#)).

The memory address is the value in register Rs1 plus the value in register Rs2 scaled by 2^n plus a 19-bit displacement.

The capabilities tag bit of the register is cleared.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18..0}	Rs2 ₆	Rs1 ₆	Rd ₆	76 ₇					LS – load / store

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDB RN, <EA> - LOAD BYTE

Description:

Load register Rd with a byte from source. The source value is sign extended to the machine width. The memory address is the value in register Rs1 plus the value in register Rs2 scaled by 1,2,4,8,16,32, 64, or 128 plus a nineteen-bit displacement.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18...0}	Rs2 ₆	Rs1 ₆	Rd ₆	64 ₇					LS – load / store

If Ms = 0 then displacement is Disp_{15...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{15...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDBU RN, <EA> - LOAD UNSIGNED BYTE

Description:

Load register Rd with a byte from source. The source value is zero extended to the machine width. The memory address is the value in register Rs1 plus the value in register Rs2 scaled by 1,2,4,8,16,32, 64, or 128 plus a 19-bit displacement.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18...0}	Rs2 ₆	Rs1 ₆	Rd ₆	65 ₇					LS – load / store

If Ms = 0 then displacement is Disp_{15...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{15...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDO RN, <EA> - LOAD OCTABYTE

Description:

Load register Rd with an octa-byte value from memory. The memory address is the value in register Rs1 plus the value in register Rs2 scaled by 2^n plus a 19-bit displacement. A cache-line constant may be substituted for the displacement.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18...0}	Rs2 ₆	Rs1 ₆	Rd ₆	70 ₇					LS – load / store

If Ms = 0 then displacement is Disp_{15...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{15...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Execution Units: AGEN, MEM

Exceptions:

Notes:

LOAD RN, <EA> - LOAD OCTABYTE

Description:

Load register Rd with an octa-byte value from memory. The memory address is the value in register Rs1 plus the value in register Rs2 scaled by 2^n plus a 19-bit displacement. A cache-line constant may be substituted for the displacement.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18...0}	Rs2 ₆	Rs1 ₆	Rd ₆	70 ₇					LS – load / store

If Ms = 0 then displacement is Disp_{15...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{15...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDT RN, <EA> - LOAD TETRA

Description:

Load register Rd with a tetra from memory. The memory value is sign extended to the machine width. The memory address is the value in register Rs1 plus the value in register Rs2 scaled by 1,2,4,8,16,32, 64, or 128 plus a nineteen-bit displacement.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18...0}	Rs2 ₆	Rs1 ₆	Rd ₆	68 ₇					LS – load / store

If Ms = 0 then displacement is Disp_{15...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{15...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDTU RN, <EA> - LOAD UNSIGNED TETRA

Description:

Load register Rd with a tetra from memory. The memory value is zero extended to the machine width. The memory address is the value in register Rs1 plus the value in register Rs2 scaled by 1,2,4,8,16,32, 64, or 128 plus a nineteen-bit displacement.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18...0}	Rs2 ₆	Rs1 ₆	Rd ₆	69 ₇					LS – load / store

If Ms = 0 then displacement is Disp_{15...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{15...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Execution Units: AGEN, MEM

Exceptions:

Notes:

LDW RN, <EA> - LOAD WYDE

Description:

Load register Rd with a wyde from source. The source value is sign extended to the machine width. The memory address is the value in register Rs1 plus the value in register Rs2 scaled by 1,2,4,8,16,32, 64, or 128 plus a nineteen-bit displacement.

The capabilities tag bit of the register is cleared.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18...0}	Rs2 ₆	Rs1 ₆	Rd ₆	66 ₇					LS – load / store

If Ms = 0 then displacement is Disp_{15...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{15...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Exceptions:

Notes:

LDWU RN, <EA> - LOAD UNSIGNED WYDE

Description:

Load register Rd with a wyde from source. The source value is zero extended to the machine width. The memory address is the value in register Rs1 plus the value in register Rs2 scaled by 1,2,4,8,16,32, 64, or 128 plus a sixteen-bit displacement.

The capabilities tag bit of the register is cleared.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18...0}	Rs2 ₆	Rs1 ₆	Rd ₆	67 ₇					LS – load / store

If Ms = 0 then displacement is Disp_{15...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{15...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Execution Units: AGEN, MEM

Exceptions:

Notes:

STORE OPERATIONS

STB RS, <EA> - STORE BYTE

Description:

Store a byte from register Rs to memory. The memory address is the value in register Rs1 plus the value in register Rs2 scaled by 1,2,4,8,16,32, 64, or 128 plus a nineteen-bit displacement.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18...0}	Rs ₂₆	Rs ₁₆	Rs ₆	80 ₇					LS – load / store

If Ms = 0 then displacement is Disp_{15...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{15...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

STI RS, <EA> - STORE IMMEDIATE TO MEMORY

Description:

Store a constant value located on the cache line to memory. The size of the constant store is determined by the SzLoc₆ bits of the instruction.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18..0}	Rs2 ₆	Rs1 ₆	SzLoc ₆	84 ₇					LS – load / store

If Ms = 0 then displacement is Disp_{15..0} otherwise Disp_{3..0} locate the constant on the cache-line and Disp_{5..4} indicate the size.

Ms	Disp _{5..4}	Displacement
0	~	Disp _{15..0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Size ₂	Mnemonic	
0	STIB	8-bits (stored as 16-bits on cache line)
1	STIW	16-bits
2	STIT	32-bits
3	STIO	64-bits

STO RS, <EA> - STORE OCTABYTE TO MEMORY

Description:

Store a value from register Rs to memory.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18...0}	Rs ₂ ₆	Rs ₁ ₆	Rs ₆	83 ₇					LS – load / store

If Ms = 0 then displacement is Disp_{15...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{15...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

STORE RS, <EA> - STORE TO MEMORY

Description:

This is an alternate mnemonic for STO. Store a value from register Rs to memory.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18...0}	Rs ₂ ₆	Rs ₁ ₆	Rs ₆	83 ₇					LS – load / store

If Ms = 0 then displacement is Disp_{15...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{15...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

STT RS, <EA> - STORE TETRA

Description:

Store a tetra from register Rs to memory.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18...0}	Rs2 ₆	Rs1 ₆	Rs ₆	82 ₇					LS – load / store

If Ms = 0 then displacement is Disp_{15...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{15...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

STW RS, <EA> - STORE WYDE

Description:

Store a wyde from register Rs to memory.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18...0}	Rs ₂ ₆	Rs ₁ ₆	Rs ₆	81 ₇					LS – load / store

If Ms = 0 then displacement is Disp_{15...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{15...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

MISCELLANEOUS MEMORY OPERATIONS

CACHE <CMD>,<EA>

Description:

Issue command to cache controller.

Instruction Format: d[Rs1+Rs2*scale]

47	45	44	43	28	27	26	21	20	19	14	13	12	7	6	0
Sc ₃	Ms		Disp _{15...0}	N2	Rs2 ₆	N1	Rs1 ₆	0	Cmd ₆	75 ₇					

Notes:

Cmd ₆	Cache	
??000	Ins.	Invalidate cache
??001	Ins.	Invalidate line
??010	TLB	Invalidate TLB
??011	TLB	Invalidate TLB entry
000???	Data	Invalidate cache
001???	Data	Invalidate line
010???	Data	Turn cache off
011???	Data	Turn cache on
100100	LSQINV	Invalidate LSQ entry

Command 100100b invalids the LSQ entry associated with the specified address.

LOADA RN, <EA> - LOAD ADDRESS

Description:

Load register Rd with the computed memory address. The memory address is the value in register Rs1 plus the value in register Rs2 scaled by 2^n plus a 19-bit displacement.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18..0}	Rs2 ₆	Rs1 ₆	Rd ₆	20 ₇					LS – load / store

If Ms = 0 then displacement is Disp_{15..0} otherwise Disp_{3..0} locate the constant on the cache-line and Disp_{5..4} indicate the size.

Ms	Disp _{5..4}	Displacement
0	~	Disp _{15..0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Clock Cycles: 1

Execution Units: All SAUs

Exceptions: none

Notes:

POP – POP REGISTERS FROM STACK

Description:

This instruction pops up to five registers from the stack. Note ‘N’ may only vary between one and five. The stack pointer to use is specified in Rd. Typically this would be r37. Use r38 to pop from the safe stack.

Instruction Format: PUSH

47	45	44 43	42	37	36	31	30	25	24	19	18	13	12	7	6	0
N ₃	0 ₂	Rs5 ₆	Rs4 ₆	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	55 ₇								

Operation:

If (N > 0) Rs1 = Mem[Rd]
If (N > 1) Rs2 = Mem[Rd+8]
If (N > 2) Rs3 = Mem[Rd+16]
If (N > 3) Rs4 = Mem[Rd+32]
If (N > 4) Rs5 = Mem[Rd+40]
Rd = Rd + N * 8

PUSH – PUSH REGISTERS ON STACK

Description:

This instruction pushes up to five registers onto the stack. ‘N’ encodes the register count, 1 to 5. The stack pointer to use is specified in Rd. Typically this would be r37. Use r38 to push to the safe stack.

Instruction Format: PUSH

47	45	44 42	41	37	36	32	31	27	26	22	21	17	16	12	11	7	6	0
N ₃	0 ₃	Rs7 ₅	Rs6 ₅	Rs5 ₅	Rs4 ₅	Rs3 ₅	Rs2 ₅	Rs1 ₅	54 ₇									

47	45	44 43	42	37	36	31	30	25	24	19	18	13	12	7	6	0	
N ₃	0 ₂	Rs5 ₆	Rs4 ₆	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	54 ₇									

Operation:

if (N > 0) Memory₈[Rd-8] = Rs1
if (N > 1) Memory₈[Rd-16] = Rs2
if (N > 2) Memory₈[Rd-24] = Rs3
if (N > 3) Memory₈[Rd-32] = Rs4

if ($N > 4$) $\text{Memory}_8[\text{Rd}-40] = \text{Rs5}$

$\text{SP} = \text{SP} - N * 8$

V2P RN, <EA> - TRANSLATE VIRTUAL TO PHYSICAL ADDRESS

Description:

Load register Rd with the computed physical memory address. The memory address is the value in register Rs1 plus the value in register Rs2 scaled by 2^n plus a 19-bit displacement translated to a physical address. The operation works in a manner similar to a load operation, but the physical address is returned instead of the data.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms			Disp _{18...0}	Rs2 ₆	Rs1 ₆	Rs ₆	90 ₇				LS – load / store

If Ms = 0 then displacement is Disp_{15...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{15...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Clock Cycles:

Execution Units: All SAUs

Exceptions: none

Notes:

VECTOR MEMORY OPERATIONS

VVN2P RN, <EA> - TRANSLATE VIRTUAL TO PHYSICAL ADDRESS VECTORS

Description:

Load vector register Rd with the computed physical memory addresses. The memory addresses are the value in register Rs1 plus the offset value in vector register Rs2 scaled by 2^n plus a 10-bit displacement translated to a physical address. The operation works in a manner like an indexed vector load operation, but the physical addresses are returned instead of the data.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	34	33 31	30	25	24	19	18	13	12	7	6	0
Sc ₃	Ms	Disp _{9...0}		l ₃	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆		91 ₇					VLS – vector load / store

If Ms = 0 then displacement is Disp_{5...0} otherwise Disp_{5...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{5...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Clock Cycles:

Execution Units: All SAUs

Exceptions: none

Notes:

VV2P RN, <EA> - TRANSLATE VIRTUAL TO PHYSICAL ADDRESS VECTORS

Description:

Load vector register Rd with the computed physical memory addresses. The memory addresses are the value in register Rs1 plus the value in register Rs2 scaled by 2^n multiplied by the lane number, plus a 10-bit displacement translated to a physical address. The operation works in a manner like a vector load operation, but the physical addresses are returned instead of the data.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	34	33 31	30	25	24	19	18	13	12	7	6	0
Sc ₃	Ms	Disp _{9...0}	0 ₃	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	91 ₇	VLS – vector load / store						

If Ms = 0 then displacement is Disp_{5...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{5...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Clock Cycles:

Execution Units: All SAUs

Exceptions: none

Notes:

VXLOAD RN, <EA>, RM - LOAD VECTOR STRIDED

Description:

Load vector register Rd with a vector from memory. The memory address is the value in register Rs1 plus the value in scalar register Rs2 scaled by 2^n multiplied by the lane number, plus a 10-bit displacement. A cache-line constant may be substituted for the displacement.

The size of the memory operation is determined by the [U_VELSZ](#) field corresponding to the selected data format.

The load operation is guided by the mask register Rs3. Each set bit in Rs3 indicates to load that lane number.

Instruction Format: d[Rs1+Rs2*Sc*Lane]

47	45	44	43	34	33 31	30	25	24	19	18	13	12	7	6	0
Sc ₃	Ms	Disp _{9...0}	Dt ₃	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	71 ₇	VLS – vector load / store						

If Ms = 0 then displacement is Disp_{5...0} otherwise Disp_{5...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{5...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Dt ₃	Mnemonic	Data Type
0	VLOAD	Integer
1	VFLOAD	Floating-point
2	VXLOAD	Fixed-point
3	VALOAD	Address
4	VCLOAD	Character

Execution Units: AGEN, MEM

Exceptions:

Notes:

VXLOADN RN, <EA>, RM - LOAD VECTOR, VECTOR INDEXED

Description:

Load vector register Rd with a vector from memory. The memory address is the value in register Rs1 plus the value in vector register Rs2, plus a 10-bit displacement. A cache-line constant may be substituted for the displacement. Each lane of vector Rs2 is used to form the memory address for that vector lane. Vector Rs2 will be defined by the address component of the VLEN and VELSZ registers. The length of the address vector should be at least as great as the length of the vector being loaded. With an address lane size independent of the data lane size, it is possible for instance to use only an eight-bit address offset with larger data.

The size of the memory operation is determined by the U_VELSZ field corresponding to the selected data format.

A vector of eight 64-bit floats may be stored using an address vector of eight 8-bit offsets. In which case the address vector will consume only a single register, while the data uses eight registers.

The load operation is guided by the mask register Rs3. Each set bit in Rs3 indicates to load that lane number.

Instruction Format: d[Rs1+Rs2[Lane]]

47	45	44	43	34	33 31	30	25	24	19	18	13	12	7	6	0
Sc ₃	Ms	Disp _{9...0}	Dt ₃	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	79 ₇	VLS – vector load / store						

If Ms = 0 then displacement is Disp_{5...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{5...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Dt ₃	Mnemonic	Data Type
0	VLOADN	Integer
1	VFLOADN	Floating-point
2	VXLOADN	Fixed-point
3	VALOADN	Address
4	VCLOADN	Character

Execution Units: AGEN, MEM

Exceptions:

Notes:

VXSTORE RN, <EA>, RM - STORE VECTOR STRIDED

Description:

Store vector register Rd with to memory. The memory address is the value in register Rs1 plus the value in scalar register Rs2 scaled by 2^n multiplied by the lane number, plus a 10-bit displacement. A cache-line constant may be substituted for the displacement.

The size of the memory operation is determined by the [VELSZ](#) field corresponding to the selected data format.

The store operation is guided by the mask register Rs3. Each set bit in Rs3 indicates to store that lane number.

Instruction Format: d[Rs1+Rs2*Sc*Lane]

47	45	44	43	34	33 31	30	25	24	19	18	13	12	7	6	0
Sc ₃	Ms	Disp _{9...0}	Dt ₃	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	88 ₇	VLS – vector load / store						

If Ms = 0 then displacement is Disp_{5...0} otherwise Disp_{5...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{5...0}
1	0	Reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Dt ₃	Mnemonic	Data Type
0	VSTORE	Integer
1	VFSTORE	Floating-point
2	VXSTORE	Fixed-point
3	VASTORE	Address
4	VCSTORE	Character

Execution Units: AGEN, MEM

Exceptions:

Notes:

VXSTOREN RN, <EA>, RM - STORE VECTOR, VECTOR INDEXED

Description:

Store vector register Rd to memory. The memory address is the value in register Rs1 plus the value in vector register Rs2, plus a 10-bit displacement. A cache-line constant may be substituted for the displacement. Each lane of vector Rs2 is used to form the memory address for that vector lane. Vector Rs2 will be defined by the address component of the [U_VLEN](#) and [U_VELSZ](#) registers. The length of the address vector should be at least as great as the length of the vector being stored. With an address lane size independent of the data lane size, it is possible for instance to use only an eight-bit address offset with larger data.

The size of the memory operation is determined by the [U_VELSZ](#) field corresponding to the selected data format.

A vector of eight 64-bit floats may be stored using an address vector of eight 8-bit offsets. In which case the address vector will consume only a single register, while the data uses eight registers.

The store operation is guided by the mask register Rs3. Each set bit in Rs3 indicates to store that lane number.

Instruction Format: d[Rs1+Rs2[Lane]]

47	45	44	43	34	33 31	30	25	24	19	18	13	12	7	6	0
Sc ₃	Ms	Disp _{9...0}	Dt ₃	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	89 ₇	VLS – vector load / store						

If Ms = 0 then displacement is Disp_{5...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{5...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Dt ₃	Mnemonic	Data Type
0	VSTOREN	Integer
1	VFSTOREN	Floating-point
2	VXSTOREN	Fixed-point
3	VASTOREN	Address
4	VCSTOREN	Character

Execution Units: AGEN, MEM

Exceptions:

Notes:

BLOCK INSTRUCTIONS

OVERVIEW

Block instructions perform operations on blocks of memory and take place in the background. A block operation is queued. A handle for the block operation is returned by block operate instructions. The handle will be zero if the operation could not be queued.

Summary of Block Operations

Op ₅	Operation Performed
0	Get Status
1	Get count
2	Cancel operation
3	store
4	move
8 to 15	compare
16 to 23	find
24 to 31	reserved

B_COMPARE – BLOCK COMPARE

Description:

This instruction compares data from the memory location addressed by Rs1 to the memory location addressed by Rs2 until the loop counter Rs3 reaches zero or until a mismatch occurs. Rs1 and Rs2 increment by the specified amount. The block status will be zero if the entire block matches the compare condition; otherwise, status will be non-zero. A handle for the operations is returned in register Rd. If the block operation could not be queued the handle returned is zero.

Instruction Format:

63	57	56	54	5352	5149	4847	46	43	42	41	34	33	32	25	24	23	16	15	14	7	6	0
Imm ₇	Pr ₃	~	Prc ₃	Id ₂	Op ₄	Nc	Rc ₈	Nb	Rb ₈	Na	Ra ₈	Nt	Rt ₈	109 ₇								

Prc ₃	Precision
0	Byte
1	Wyde
2	Tetra
3	Octa
4	reserved
others	reserved

Op ₄			
0	==	BCMP.EQ	equal
1	!=	BCMP.NE	not equal
2	<	BCMP.LT	signed
3	<=	BCMP.LE	signed
4	<	BCMP.LTU	unsigned
5	<=	BCMP.LEU	unsigned

Assembler Example

```
LDI Rs3,200
BCMPO.EQ Rd, [Rs1], [Rs2], Rs3, +8, +8
BCOUNT Rs3, Rd
SUBF Rs3, Rs3, 200      ; get index of difference
```

Execution Units: Memory

Operation:

while Rs3 <> 0 and mem[Rs2] op mem[Rs1]==1

 Rs1 = Rs1 + amt

 Rs2 = Rs2 + amt

 Rs3 = Rs3 - 1

BCANCEL – CANCEL BLOCK OPERATION

Description:

This instruction cancels the block operation specified by register Rs1.

Instruction Format:

63	57	56	54	5352	5149	4847	46	43	42	41	34	33	32	25	24	23	16	15	14	7	6	0
~ ₇	Pr ₃	~	~ ₃	~ ₂	2 ₄	~	~ ₈	~	~ ₈	~	~ ₈	Na	Ra ₈	Nt	Rt ₈	111 ₇						

Execution Units: Memory, ALU, Flow Control

Operation:**Assembler Example**

```
LDI LC,200  
BSTOREB R1, R2, [R3], LC, +1  
BCOUNT R5, R1
```

BCOUNT – GET BLOCK COUNT

Description:

This instruction gets the current count for the block operation whose handle is in register Rs1. The count is returned in register Rd. The count is only approximate as the block operation continues in the background.

Instruction Format:

63	57	56	54	5352	51 49	48 47	46	43	42	41	34	33	32	25	24	23	16	15	14	7	6	0
~ ₇	Pr ₃	~	~ ₃	~ ₂	1 ₄	~	~ ₈	~	~ ₈	~	~ ₈	Na	Ra ₈	Nt	Rt ₈	111 ₇						

Execution Units: Memory, ALU, Flow Control

Operation:

Rd = count (block operation[Rs1])

Assembler Example

```
LDI LC,200  
BSTOREB R1, R2, [R3], LC, +1  
BCOUNT R5, R1
```

BFIND – BLOCK FIND

Description:

This instruction compares data from the memory location in Rs1 to the data in register Rs2. If the data is found matching the specified function, Rs3 will be non-zero. A handle for the operations is returned in register Rd.

Instruction Format:

63	57	56	54	5352	51 49	48 47	46	43	42	41	34	33	32	25	24	23	16	15	14	7	6	0
Imm ₇	Pr ₃		~	Prc ₃	Id ₂	Op ₄	Nc	Rc ₈	Nb	Rb ₈	Na	Ra ₈	Nt	Rt ₈		108 ₇						

Prc ₃	Precision
0	Wyde
1	Tetra
2	Octa
3	Reserved
4	Byte
others	Reserved

Op ₄			
0	==	BFIND.EQ	equal
1	!=	BFIND.NE	not equal
2	<	BFIND.LT	signed
3	<=	BFIND.LE	signed
4	<	BFIND.LTU	unsigned
5	<=	BFIND.LEU	unsigned

Execution Units: Memory

Operation:

BMOVE –BLOCK MOVE

Description:

This instruction moves a data from the memory location addressed by Rs1 to the memory location addressed by Rs2 until the loop counter Rs3 reaches zero. Rs3 is a count of the number of bytes to move. A handle for the operation is returned in register Rd.

Instruction Format:

63	57	56	54	5352	51 49	48 47	46	43	42	41	34	33	32	25	24	23	16	15	14	7	6	0
\sim_7	Pr ₃		~	\sim_3	Id ₂	4 ₄	Nc	Rc ₈	Nb	Rb ₈	Na	Ra ₈	Nt	Rt ₈		111 ₇						

Assembler Example

```
LDI LC,200  
BMOVE Rd, [Rs1], [Rs2], Rs3
```

Execution Units: Memory**Operation:**

```
while Rs3 <> 0  
    t0 = mem[Rs1]  
    mem[Rs2] = t0  
    Rs1 = Rs1 + Imma  
    Rs2 = Rs2 + Immb  
    Rs3 = Rs3 - 1
```

BSTATUS – BLOCK OPERATION STATUS

Description:

This instruction gets the status of the block operation whose handle is in register Rs1. The status is returned in register Rd.

Instruction Format:

63	57	56	54	5352	51 49	48 47	46	43	42	41	34	33	32	25	24	23	16	15	14	7	6	0
~7	Pr ₃	~	~ ₃	~ ₂	0 ₄	~	~ ₈	~	~ ₈	~	~ ₈	Na	Ra ₈	Nt	Rt ₈	111 ₇						

Execution Units: Memory, ALU, Flow Control

Result in Rd:

Bit	
0	0=operation complete, 1=in progress
1	0=count is zero, 1=count is non-zero

Operation:

Rd = status (block operation[Rs1])

Assembler Example

LDI LC,200

BSTOREB Rd, Rs2, [Rs1], LC, +1

BSTORE – BLOCK STORE

Description:

This instruction stores data contained in register Rs2 to consecutive memory locations beginning at the address in Rs1 until register Rs3 reaches zero. A handle for the operations is returned in register Rd.

Instruction Format:

63	57	56	54	5352	5149	4847	46	43	42	41	34	33	32	25	24	23	16	15	14	7	6	0
\sim_7	Pr ₃	~	Prc ₃	Id ₂	3 ₄	Nc	Rc ₈	Nb	Rb ₈	Na	Ra ₈	Nt	Rt ₈								111 ₇	

Id: 0 = increment, 1 = decrement

Prc ₃	Precision
0	Byte
1	Wyde
2	Tetra
3	Octa
4	reserved
others	reserved

Execution Units: Memory, ALU, Flow Control

Operation:

```
if Rs3 <> 0
    mem[Rs1] = Rs2
    Rs1 = Rs1 + -
    Rs3 = Rs3 - 1
```

Assembler Example

```
LDI LC,200
BSTOREB Rd, Rs2, [Rs1], LC, +1
```

VECTOR SPECIFIC INSTRUCTIONS

OVERVIEW

VGMASK – GENERATE VECTOR MASK

Description:

Create a vector mask based on the number of lanes to enable. The number of lanes to enable is the lesser of Rs1 or Rs2.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
58 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇							R3

Operation: R3

Rd = generate mask (minu (Rs1, Rs2))

Clock Cycles: 1

VECTOR SPECIFIC INSTRUCTIONS

VEINS / VMOVSV – VECTOR ELEMENT INSERT

Synopsis

Vector element insert.

Description

A general-purpose register Rs1 is transferred into one element of a vector register Rd. The element to insert is identified by Rs2.

Instruction Format: R2

47	41	4039	38	36	35	34	33	29	28	27	26	22	21	20	19	15	14	13	12	8	7	6	0
51 ₇	~ ₂	~ ₃	~	~	~ ₅	Nb	Vb	Rb ₅	Na	Va	Ra ₅	Nt	Vt	Rt ₅	0	2 ₇							

Operation

$$Rd[Rs2] = Rs1$$

Exceptions: none

VEX / VMOVS – VECTOR ELEMENT EXTRACT

Synopsis

Vector element extract.

Description

A vector register element from Va is transferred into a general-purpose register Rd. The element to extract is identified by Rs2. Rs2 and Rd are scalar registers.

Instruction Format: R2

47	41	4039	38	36	35	34	33	29	28	27	26	22	21	20	19	15	14	13	12	8	7	6	0
50 ₇	~ ₂	~ ₃	~	~	~ ₅	Nb	Vb	Rb ₅	Na	Va	Va ₅	Nt	Vt	Rt ₅	0	2 ₇							

Operation

$$Rd = Va[Rs2]$$

Exceptions: none

VGNDX – GENERATE INDEX

Description

A value in a register Rs1 is multiplied by the element number and added to a value in Rs2 and copied to elements of vector register Vt guided by a vector mask register. Rs1 is a scalar register. This operation may be used to compute memory addresses for a subsequent vector load or store operation. Only the low order 24-bits of Rs1 are involved in the multiply. The result of the multiply is a product less than 41 bits in size. The multiply is a fast 24x16 bit multiply.

Instruction Format: R2

47	41	4039	38	36	35	34	33	29	28	27	26	22	21	20	19	15	14	13	12	8	7	6	0
52 ₇	Msk ₂	~ ₃	~	~	~ ₅	Nb	Vb	Rb ₅	Na	Va	Ra ₅	Nt	Vt	Vt ₅	1	2 ₇							

Operation

$$y = 0$$

for x = 0 to VL - 1

if (Pr[x])

$$Vt[y] = Rs1 * y + Rs2$$

$$y = y + 1$$

VMFILL – VECTOR MASK FILL

Description:

Fill the contents of a vector mask register with a mask of ones beginning at $Rs2_7$ and ending at $Rs3_7$, inclusive. Fill the remainder of the register with zeros.

Instruction Format:

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
59_7	Ms_3	Op_3		VN_4	$Rs3_6$	$Rs2_6$	\sim_6		Rd_6	$Opcode_7$					R3

1 clock cycle

Exceptions: none

VMFIRST – FIND FIRST SET BIT

Description

This is an alternate mnemonic for the count trailing zeros, [CNTTZ](#), instruction.

The position of the first bit set in the mask register is copied to the destination register. If no bits are set the value is -1. The search begins at the least significant bit and proceeds to the most significant bit.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
26_7	Ms_3	Op_3		VN_4	$Rs3_6$	6_6		$Rs1_6$	Rd_6	$Opcode_7$					R3

Operation

$Rd = \text{first set bit number of } (Rs1)$

Exceptions: none**Execution Units:** SAU #0

VMLAST – FIND LAST SET BIT

Description

The position of the last bit set in the mask register is copied to the destination register. If no bits are set the value is -1. The search begins at the most significant bit of the mask register and proceeds to the least significant bit.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
26_7	Ms_3	1_3	VN_4	$Rs3_6$	\sim_6	$Rs1_6$	Rd_6	$Opcode_7$	R3						

Operation

$$Rd = \text{last set bit number of } (Vm)$$

Exceptions: none

Execution Units: SAU #0

BRANCH / FLOW CONTROL INSTRUCTIONS

OVERVIEW

There are two basic types of branches: conditional branches and unconditional branches.

Conditional branches follow the fused compare-and-branch paradigm. Two registers (or cache-line immediate values) are compared then a branch takes place if the comparison result is true. There are two formats for conditional branches, one where the branch target address is a relative displacement away from the branch instruction address, and one where the branch target address is contained in a register.

Unconditional branches always branch.

MNEMONICS

There are mnemonics for specifying the comparison method. Floating-point comparisons prefix the branch mnemonic with ‘F’ as in FBEQ. Decimal-floating point comparisons prefix the branch mnemonic with ‘DF’ as in DFBEQ. And finally posit comparisons prefix the branch mnemonic with a ‘P’ as in ‘PB EQ’. There is no prefix for integer branches.

CONDITIONAL BRANCH FORMAT

Branch target address is displaced from branch instruction address:

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	Cnd ₄	Opcode ₇	BR – conditional branch				

Branch target address is contained in a register, Rs3:

4746	45	44	31	30	25	24	19	18	13	1211	10	7	6	0
N ₂	1	~ ₁₃	Rs3 ₇	Rs2 ₆	Rs1 ₆	Ms ₂	Cnd ₄	Opcode ₇	BRR – branch to register					

CONDITIONS

Conditional branches branch to the target address only if the condition is true. The condition is determined by the comparison or logical / arithmetic operation of two general-purpose registers (or cache-line constants).

*The original Thor machine used instruction predicates to implement conditional branching.
Another instruction was required to set the predicate before branching. Combining compare and branch in a single instruction may reduce the dynamic instruction count. An issue with comparing and branching in a single instruction is that it may lead to a wider instruction format.*

BRANCH CONDITIONS

The branch opcode and condition field determines the condition under which the branch will execute.

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs ₂ ₆	Rs ₁ ₆	Ms ₂	Cnd ₄	Opcode ₇	BR – conditional branch				

Type	Unsigned Integer				Signed Integer			
Precision	8	16	32	64	8	16	32	64
Opcode	24	25	26	27	28	29	30	31
Type	Float-point				DFP	Posit		
Precision	16	32	64	128	128	64		
Opcode	40	41	42	43	44	45		

Integer / Address Conditions

Fn ₄	Unsigned	Signed	Comparison Test
0	EQ / ENORB	ENOR	Equal
1	NE / EORB	EOR	not equal
2	LTU	LT	less than
3	LEU	LE	less than or equal
4	GEU	GE	greater or equal
5	GTU	GT	greater than
6	BC		Bit clear
7	BS		Bit set
8	NANDB	NAND	And zero
9	ANDB	AND	And non-zero
10	NORB	NOR	Or zero
11	ORB	OR	Or non-zero
12	PTR		Rs1 or Rs2 flagged a pointer
13	OV		Rs1 or Rs2 overflow
14	BTWU	BTW	Three way branch
15	IRQ		IPL > SR.IM
others			reserved

Float Conditions

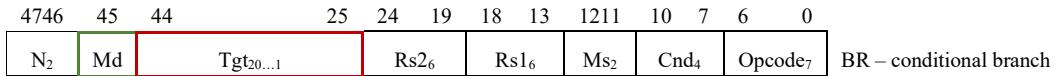
Cnd ₄	Mnem.	Meaning	Test
0	BEQ	equal	!nan & eq
1	BNE	not equal	!eq
2	BLT	Less than	Lt & (!nan & !inf & !eq)
3	BLE	Less than or equal	Eq (lt & !nan)
4	BGE	greater than or equal	Eq (!nan & !lt & !inf)
5	BGT	greater than	!nan & !eq & !lt & !inf
6	BGL	Greater than or less than	!nan & (!eq & !inf)
7	BOR	Greater than less than or equal / ordered	!nan
8		reserved	
9	BUGL	Unordered or greater than or less than	Nan !eq
10	BULT	Unordered or less than	Nan (!eq & lt)
11	BULE	unordered less than or equal	Nan (eq lt)
12	BUGE	Unordered or greater than or equal	Nan (!lt eq)
13	BUGT	Unordered or greater than	Nan (!eq & !lt & !inf)
14		reserved	
15	BUN	Unordered	Nan

BRANCH TARGET

CONDITIONAL BRANCHES

For conditional branches, the target address is formed in one of two ways. **One**, as the sum of the instruction pointer and a constant specified in the instruction. Relative branches have a range of approximately $\pm 1\text{MB}$ or 21 displacement bits. **Two**, the target address may come from the contents of register Rs3.

The target 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. The author decided to use a division by two since instructions are six bytes in size and the target must be a multiple of six bytes away from the branches IP. Dividing by two effectively adds a displacement bit.



Md	Mode
0	Relative
1	Register Indirect (Rs3)

UNCONDITIONAL BRANCHES

There are two forms of unconditional branches or jump. A target routine that may be at any wyde (16-bit aligned) address. The first form has a 36-bit displacement or address directly encoded in the instruction. The target displacement field is large enough to accommodate a $\pm 2^{35}$ range. This is adequate for many applications. The second form uses a register indirect address with displacement.



BAND –BRANCH IF LOGICAL AND TRUE

BAND Rs1, Rs2, label

Description:

Branch if the logical and of source operands results in a non-zero value. The displacement is relative to the address of the branch instruction. This ‘and’ operation reduces the source operand values to a Boolean true or false before performing the operation. A non-zero value is considered true, zero is considered false.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	9 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BANDB –BRANCH IF BITWISE AND TRUE

BANDB Rs1, Rs2, label

Description:

Branch if the bitwise and of source operands results in a non-zero value. The displacement is relative to the address of the branch instruction.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	9 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BBC – BRANCH IF BIT CLEAR

Description:

This instruction branches to the target address if bit Rs2 of Rs1 is clear, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	6 ₄	Opcode ₇	BR – conditional branch				

Operation:

If (Rs1.bit[Rs2] == 0)

IP = IP + Constant

Execution Units: Branch**Exceptions:** none**Notes:**

BBS – BRANCH IF BIT SET

Description:

This instruction branches to the target address if bit Rs2 of Rs1 is set, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	7 ₄	Opcode ₇	BR – conditional branch				

Operation:

If (Rs1.bit[Rs2] == 1)

IP = IP + Constant

Execution Units: Branch

Exceptions: none

Notes:

BENOR –BRANCH IF EQUAL

BENOR Rs1, Rs2, label

Description:

This is an alternate mnemonic for BEQ. Branch if source operands are equal (the exclusive OR is zero or false). This ‘enor’ operation reduces the source operand values to a Boolean true or false before performing the operation. A non-zero value is considered true, zero is considered false. The displacement is relative to the address of the branch instruction.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	0 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BEOR –BRANCH IF NOT EQUAL

BEOR Rs1, Rs2, label

Description:

This is an alternate mnemonic for BNE. Branch if source operands are not equal (the exclusive OR is non-zero or true). This ‘eor’ operation reduces the source operand values to a Boolean true or false before performing the operation. A non-zero value is considered true, zero is considered false. The displacement is relative to the address of the branch instruction.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	l ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BEQ –BRANCH IF EQUAL

BEQ Rs1, Rs2, label

Description:

Branch if source operands are equal. Source operands are treated as integers of the specified precision.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	0 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BGE –BRANCH IF GREATER THAN OR EQUAL

BGE Rm, Rn, label

Description:

Branch if the first source operand is greater than or equal to the second. Both operands are treated as signed integer values.

Instruction Format: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	4 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BGEU –BRANCH IF UNSIGNED GREATER THAN OR EQUAL

BGEU Rm, Rn, label

Description:

Branch if the first source operand is greater than or equal to the second. Both operands are treated as unsigned integer values.

Instruction Format: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	4 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BGT –BRANCH IF GREATER THAN

BGT Rs1, Rs2, label

Description:

Branch if the first source operand is greater than the second. Both operands are treated as signed integer values.

Instruction Format: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	5 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BGTU –BRANCH IF UNSIGNED GREATER THAN

BGTU Rs1, Rs2, label

Description:

Branch if the first source operand is greater than the second. Both operands are treated as unsigned integer values.

Instruction Format: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	5 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BHI –BRANCH IF HIGHER

BHI Rs1, Rs2, label

Description:

This is an alternate mnemonic for [BGTU](#). Branch if the first source operand is greater than the second. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

Instruction Format: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	5 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BLE –BRANCH IF LESS THAN OR EQUAL

BLE Rs1, Rs2, label

Description:

Branch if the first source operand is less than or equal to the second. Both operands are treated as signed integer values.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	3 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BLEU –BRANCH IF UNSIGNED LESS THAN OR EQUAL

BLEU Rs1, Rs2, label

Description:

Branch if the first source operand is less than or equal to the second. Both operands are treated as unsigned integer values.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	3 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BLT –BRANCH IF LESS THAN

BLT Rs1, Rs2, label

Description:

Branch if the first source operand is less than the second. Both operands are treated as signed integer values.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	2 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BLTU –BRANCH IF UNSIGNED LESS THAN

BLTU Rs1, Rs2, label

Description:

Branch if the first source operand is less than the second. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	2 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BNAND –BRANCH IF LOGICAL AND FALSE

BNAND Rs1, Rs2, label

Description:

Branch if the logical ‘and’ of source operands results in a zero value. This ‘nand’ operation reduces the source operand values to a Boolean true or false before performing the operation. A non-zero value is considered true, zero is considered false.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	8 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BNE –BRANCH IF NOT EQUAL

BNE Rs1, Rs2, label

Description:

Branch if source operands are not equal.

Instruction Format: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	l ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BNOR –BRANCH IF LOGICAL OR FALSE

BNOR Rs1, Rs2, label

Description:

Branch if the logical ‘or’ of source operands results in a non-zero value. This ‘nor’ operation reduces the source operand values to a Boolean true or false before performing the operation. A non-zero value is considered true, zero is considered false.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	l ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BOR –BRANCH IF LOGICAL OR TRUE

BOR Rs1, Rs2, label

Description:

Branch if the logical or of source operands results in a non-zero value. This ‘or’ operation reduces the source operand values to a Boolean true or false before performing the operation. A non-zero value is considered true, zero is considered false.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	11 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BORB –BRANCH IF BITWISE OR TRUE

BORB Rs1, Rs2, label

Description:

Branch if the bitwise ‘or’ of source operands results in a non-zero value. The displacement is relative to the address of the branch instruction.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	11 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

BTW –BRANCH THREE-WAY

BTW Rs1, Rs2, labelLT, labelEQ, labelGT

Description:

Perform a three-way branch based on whether the comparison of register Rs1 value is less than, equal to, or greater than register Rs2 value.

Formats Supported: BR

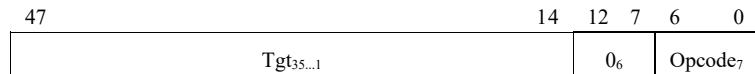
47	28	27	21	20	14	13	1211	10	7	6	0
TgtLT _{20...1}		Rs2 ₇		Rs1 ₇	0	Ms ₂	14 ₄		Opcode ₇		
TgtGT _{20...1}				TgtEQ _{20...1}			~		127 ₇		

Clock Cycles: 13

BRA – BRANCH ALWAYS

Description:

This instruction always branches to the target address. The target address range is 36 bits.

Formats Supported: BSR**Operation:**

$$IP = IP + \text{Constant} * 2$$

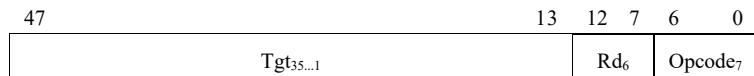
Execution Units: Integer ALU #0**Clock Cycles:** 4**Exceptions:** none**Notes:**

BSR – BRANCH TO SUBROUTINE

Description:

This instruction always jumps to the target address. The address of the next instruction is stored in a link register. The target address range is 36 bits.

Formats Supported: BSR



Operation:

Rd = next IP

IP = IP + Constant * 2

Execution Units: Integer ALU #1

Exceptions: none

Notes:

FBEQ –BRANCH IF EQUAL

FBEQ Rs1, Rs2, label

Description:

Branch if two source operands are equal. Both operands are treated as floating-point values. Positive and negative zero are considered equal. If either operand is a NaN the branch will not be taken.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	0 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

FBGE –BRANCH IF GREATER THAN OR EQUAL

FBGE Rs1, Rs2, label

Description:

Branch if source operand Rs1 is greater than or equal to Rs2. Both operands are treated as floating-point values. Positive and negative zero are considered equal. If either operand is a NaN the branch will not be taken. The displacement is relative to the address of the branch instruction.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	4 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

FBGT –BRANCH IF GREATER THAN

FBGT Rs1, Rs2, label

Description:

Branch if source operand Rs1 is greater than Rs2. Both operands are treated as floating-point values. Positive and negative zero are considered equal. If either operand is a NaN the branch will not be taken. The displacement is relative to the address of the branch instruction.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	5 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

FBLE –BRANCH IF LESS THAN OR EQUAL

FBLE Rs1, Rs2, label

Description:

Branch if source operand Rs1 is less than or equal to Rs2. Both operands are treated as floating-point values. Positive and negative zero are considered equal. If either operand is a NaN the branch will not be taken. The displacement is relative to the address of the branch instruction.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	3 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

FBLT –BRANCH IF LESS THAN

FBLT Rs1, Rs2, label

Description:

Branch if source operand Rs1 is less than Rs2. Both operands are treated as floating-point values. Positive and negative zero are considered equal. If either operand is a NaN the branch will not be taken. The displacement is relative to the address of the branch instruction.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	2 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

FBNE –BRANCH IF NOT EQUAL

FBNE Rs1, Rs2, label

Description:

Branch if two source operands are not equal. Both operands are treated as floating-point values. Positive and negative zero are considered equal.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	l ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

FBUGT –BRANCH IF UNORDERED OR GREATER THAN

FBUGT Rs1, Rs2, label

Description:

Branch if source operand Rs1 is greater than Rs2 or if the comparison is unordered. Both operands are treated as floating-point values. Positive and negative zero are considered equal.

Formats Supported: BR

4746	45	44	25	24	19	18	13	1211	10	7	6	0
N ₂	0	Tgt _{20...1}	Rs2 ₆	Rs1 ₆	Ms ₂	13 ₄	Opcode ₇	BR – conditional branch				

Clock Cycles: 13

JMP – JUMP TO TARGET

Description:

This instruction always jumps to the target address. The target address range is $\pm 2^{35}$.

Formats Supported: BSR

47	13	12	7	6	0
Tgt _{35...1}		0 ₆		33 ₇	

Operation:

$$IP = \text{sign extend Constant} * 2$$

Execution Units: Integer ALU #1, Branch

Exceptions: none

Scaled Indexed Form:

This instruction always jumps to the target address. A scaled indexed address is calculated using Rs1, Rs2 and a constant and loaded into the IP.

Instruction Format: d[Rs1+Rs2*Sc]

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18...0}		Rs2 ₆		Rs1 ₆		0 ₆		38 ₇	

If Ms = 0 then displacement is Disp_{15...0} otherwise Disp_{3...0} locate the constant on the cache-line and Disp_{5...4} indicate the size.

Ms	Disp _{5...4}	Displacement
0	~	Disp _{15...0}
1	0	reserved
	1	32-bit cache-line constant
	2	48-bit cache-line constant
	3	64-bit cache-line constant

Operation:

$$IP = Rs1 + \text{Constant} + Rs2 * \text{scale}$$

Exceptions: none

Notes:

Memory Indirect Form:

Description:

This instruction always jumps to the target address. The target address is loaded from memory at the sum of a register and an immediate constant. Low order bits of the instruction pointer may be loaded while keeping the higher order bits constant. This allows efficient implementation of jump tables.

Instruction Format: JSR

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms		Disp _{18...0}	Rs2 ₆	Rs1 ₆	Rd ₆		36 ₇				

4746	4544	43	28	27	21	20	14	13	7	6	0
Op ₂	Sc ₂		Disp _{15...0}	Rs2 ₇	Rs1 ₇	0 ₇		36 ₇			

Op ₂	Operation
0	Load only the low order 16-bits of the instruction pointer, upper bits remain the same
1	Load only the low order 32-bits of the instruction pointer, upper bits remain the same
2	Load only the low order 64-bits of the instruction pointer, upper bits remain the same
3	Load entire instruction pointer

Operation:

$$IP = \text{Memory}[Rs1 + \text{sign extend (Constant)} + Rs2 * \text{scale}]$$

Execution Units: Integer ALU #0

Exceptions: none

Notes:

JSR – JUMP TO SUBROUTINE

Description:

This instruction always jumps to the target address. The address of the next instruction is stored in register Rd.

Direct Address Form:

This instruction always jumps to the target address. The address of the next instruction is stored in register Rd. The target address field is shifted left once before use. The target address range is 36 bits.

Formats Supported: BSR

47	13	12	7	6	0
Tgt _{35...1}		Rd ₆		33 ₇	

Operation:

Rd = next IP

IP = sign extend Constant * 2

Scaled Index Form:

This instruction always jumps to the target address. The IP of the next instruction is stored in register Rd. A scaled indexed address is calculated using Rs1, Rs2 and a constant and loaded into the IP.

Instruction Format: JSR

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms			Disp _{18...0}		Rs2 ₆		Rs1 ₆		Rd ₆		38 ₇

Operation:

Rd = next IP

IP = Rs1 + Constant + Rs2 * scale

Exceptions: none

JTT – JUMP THROUGH TABLE

Description:

This instruction jumps to the target address calculated from a value in a table. The address of the next instruction is stored in register Rd. The target address is calculated from a value loaded from a table in memory whose base address is contained in register Rs1. The value is loaded from memory at the base address plus an index calculated as the scaled contents of register Rs2. The table index must be greater than or equal to zero and less than the limit set in the instruction. If the index is greater than or equal to the limit, then the entry at the limit will be jumped to.

Instruction Format: JTT

47	45	44	43	25	24	19	18	13	12	7	6	0
Sc ₃	Ms	Limit _{18..0}		Rs2 ₆	Rs1 ₆	Rd ₆	36 ₇					

Abs/rel	Sc ₃	Operation
0	0	
0	1	Load only the low order 16-bits of the instruction pointer, upper bits remain the same
0	2	Load only the low order 32-bits of the instruction pointer, upper bits remain the same
0	3	Load only the low order 64-bits of the instruction pointer, upper bits remain the same
1	4	
1	5	Add 16-bit value to instruction pointer
1	6	Add 32-bit value to instruction pointer
1	7	Add 64-bit value to instruction pointer

Operation:

Rd = next IP

If (Rs2 > limit or Rs2 < 0)

IP = IP + Memory[Rs1 + limit * scale]

else

IP = IP + Memory[Rs1 + Rs2 * scale]

Execution Units: Branch

Exceptions: none

Notes:

NOP – NO OPERATION

NOP

Description:

This instruction does not perform any operation. Any value for bits 8 to 47 may be used.

Instruction Format:

47	8	7	6	0
\sim_{40}	1	127 ₇		

Notes:

RET – RETURN FROM SUBROUTINE

Description:

This instruction returns from a subroutine by transferring program execution to the address stored in a register specified by Rs1. RET and RTS are synonymous.

Formats Supported: RTD

47	46	45	19	18	13	12	7	6	0
3 ₂		0 ₂₇		Rs1 ₆		0 ₆		35 ₇	

Operation:

IP <= Rs1

Execution Units: Branch**Exceptions:** none**Notes:**

Return address prediction hardware may make use of the RET instruction.

RTD – RETURN FROM SUBROUTINE AND DEALLOCATE

Description:

This instruction returns from a subroutine by transferring program execution to the address stored in register Rs1. Additionally, the stack pointer Rd is incremented by the amount specified.

Formats Supported: RTD

47	46	45	19	18	13	12	7	6	0
3 ₂		Immediate ₂₇		Rs1 ₆		Rd ₆		35 ₇	

Operation:

IP <= Rs1

SP = SP + sign extend Constant

Execution Units: Branch**Exceptions:** none**Notes:**

Return address prediction hardware may make use of the RTS instruction.

RTS – RETURN FROM SUBROUTINE

Description:

An alternate mnemonic for the [RTD](#) instruction where the stack pointer adjustment is zero. This instruction returns from a subroutine by transferring program execution to the address stored in a register specified by Rs1 plus an offset constant. RTS and RET are synonymous.

Formats Supported: RTD

47	46	45	19	18	13	12	7	6	0
3 ₂			0 ₂₇		Rs1 ₆		0 ₆		35 ₇

Operation:

IP <= Rs1

Execution Units: Branch**Exceptions:** none**Notes:**

Return address prediction hardware may make use of the RTS instruction.

GRAPHICS INSTRUCTIONS

BLEND – BLEND COLORS

Description:

This instruction blends two colors whose values are in Rs1 and Rs2 according to an alpha value in Rs3. The resulting color is placed in register Rd. The alpha value is a ten-bit value assumed to be a fixed-point number with one whole digit and nine fraction digits. The same alpha value should be placed in each RGB component location of Rs3. The color values in Rs1 and Rs2 are assumed to be RGB10.10.10 format colors. The result is a RGB10.10.10 format color. Note that a close approximation to $1.0 - \text{alpha}$ is used. Each component of the color is blended independently. Component overflow saturates towards white.

Instruction Format: R3

BLEND Rd, Rs1, Rs2, Rs3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
0 ₇	Ms ₃	Op ₃	VN ₄	Rs3 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	11 ₇						

Operation:

$$Rd.R = (Rs1.R * \text{alpha.R}) + (Rs2.R * \sim\text{alpha.R})$$

$$Rd.G = (Rs1.G * \text{alpha.G}) + (Rs2.G * \sim\text{alpha.G})$$

$$Rd.B = (Rs1.B * \text{alpha.B}) + (Rs2.B * \sim\text{alpha.B})$$

Clock Cycles: 2

TRANSFORM – TRANSFORM POINT

Description:

The point transform instruction transforms a point from one location to another using a transform function. The transform function has 12 co-efficients in the form of a matrix used in the calculation.

Points are represented in 18.18 fixed-point format.

Instruction Format:

47	41	40	38	37	35	34	28	27	21	20	14	13	7	6	0
Op ₇	Ms ₃	Op ₃		Rs3 ₇		Rs2 ₇		Rs1 ₇		Rd ₇		22 ₇			

Op ₇	Operation
0	Blend
1	Transform: Return new X
2	Transform: Return new Y
3	Transform: Return new Z
4	Transform: Get coefficient
5	Transform: Set coefficient

To set a coefficient Rs1 specifies which coefficient to set, Rs2 specifies the value.

Rx	Ry Co-efficient
0	aa
1	ab
2	ac
3	tx
4	ba
5	bb
6	bc
8	ty
9	ca
10	cb
11	cc
12	tz

Clock Cycles: 4

Operation:

Input matrix M:

$$M = \begin{vmatrix} aa & ab & ac & tx \\ ba & bb & bc & ty \\ ca & cb & cc & tz \end{vmatrix}$$

Input point P:

$$P = \begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix}$$

Output point P':

$$P' = MP = \begin{vmatrix} x' \\ y' \\ z' \end{vmatrix} = \begin{vmatrix} aa*x + ab*y + ac*z + tx \\ ba*x + bb*y + bc*z + ty \\ ca*x + cb*y + cc*z + tz \end{vmatrix}$$

Clock Cycles: 3

PROCESSOR QUEUE MANAGEMENT INSTRUCTIONS

OVERVIEW

There are several hardware queues in the processor. Queues are accessible with instructions dedicated to queue management described further in this section of the document.

Queue Number	Usage
0	Graphics queue
14	NaN trace queue
15	Instruction trace queue

PEEKQ – PEEK AT QUEUE / STACK

Description:

This instruction returns the top value into Rd from the hardware queue specified in Rs2. The hardware queue position is not advanced. Unused value bits should read as zero. Used the STATQ instruction to get the queue status.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
104 ₇	Ms ₃	0 ₃	VN ₄	0 ₆	Rs2 ₆	0 ₆	Rd ₆	Opcode ₇						

Exceptions: none

POPQ – POP FROM QUEUE / STACK

Description:

This instruction pops a value into Rd from the hardware queue specified in Rs2. The hardware queue position is advanced. Unused value bits should read as zero. To check the queue status, use the STATQ instruction.

63	0
Value	

Value: the value that was pushed to the queue

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
105 ₇	Ms ₃	0 ₃	VN ₄	0 ₆	Rs2 ₆	0 ₆	Rd ₆	Opcode ₇						

Exceptions: none**Notes:**

Queue #14 is the NaN trace queue

Queue #15 is the instruction trace queue

PUSHQ – PUSH ON QUEUE / STACK

Description:

This instruction pushes an N-bit value in Rs1 onto the hardware queue specified in Rs2. Where N is implementation defined between 1 and 64 bits. To check the queue status, use the STATQ instruction.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
106 ₇	Ms ₃	0 ₃	VN ₄	0 ₆	Rs2 ₆	0 ₆	Rd ₆	Opcode ₇							

Exceptions: none

READQ – READ FROM QUEUE / STACK

Description:

This instruction reads a specific queue entry identified by Rs1 into Rd from the hardware queue specified in Rs2. Unused value bits should read as zero. Used the STATQ instruction to get the queue status.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
110 ₇	Ms ₃	0 ₃	VN ₄	0 ₆	Rs2 ₆	Rs1 ₆	Rd ₆	Opcode ₇							

Exceptions: none

RESETQ – RESET QUEUE / STACK

Description:

This instruction resets the hardware queue specified by Rs2.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
107 ₇	Ms ₃	0 ₃	VN ₄	0 ₆	Rs2 ₆	0 ₆	0 ₆	Opcode ₇						

Exceptions: none

STATQ – GET STATUS OF QUEUE / STACK

Description:

This instruction returns a queue status value into Rd from the hardware queue specified in Rs2. The hardware queue position is not advanced. Unused value bits should read as zero.

63	62	61		16	15	0
Qe	Dv		~46		Data Count	

Fields

Qe: empty. If set, this bit indicates that the queue/stack is empty.

Dv: data valid. If this bit is set it indicates that valid data is present at the queue.

Dc: data count: The number of items left in the queue

XD: The high order 48 bits of the data stored by the queue if the queue is wider than 64 bits.

Instruction Format: R3

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
108 ₇	Ms ₃	0 ₃	VN ₄	0 ₆	Rs2 ₆	0 ₆	Rd ₆	Opcode ₇						

Exceptions: none

WRITEQ – WRITE TO QUEUE / STACK

Description:

This instruction writes a specific queue entry identified by Rs1 from Rd to the hardware queue specified in Rs2. Used the STATQ instruction to get the queue status.

Instruction Format: R3

47	41	40 38	37	35	34 31	30	25	24	19	18	13	12	7	6	0
111_7	Ms_3	0_3	VN_4	0_6	$Rs2_6$	$Rs1_6$	Rd_6	$Opcode_7$							

Exceptions: none

SYSTEM INSTRUCTIONS

BRK – BREAK

Description:

This instruction initiates the processor debug routine. The processor enters debug mode. The cause code register is set to indicate execution of a BRK instruction. Interrupts are disabled. The instruction pointer is reset to the vector located from the contents of the kernel vector corresponding to the operating mode and instructions begin executing. The address of the BRK instruction is stored in the EIP.

Instruction Format: BRK

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
~7	~3	~3	~4	~6	~6	~6	~6	0 ₆	0 ₆	0 ₇				

Operation:

CAUSE = 0

PUSH SR

PUSH IP

EIP = IP

IP = kernel_vector[operating mode]

Execution Units: Branch

Clock Cycles:

Exceptions: none

Notes:

ESC – ESCALATE EXCEPTION

Description:

This instruction escalates an exception, triggering an exception at the next higher operating mode.

Instruction Format: ESC

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
7 ₇	Ms ₃	~ ₃	V ₄	~ ₆	~ ₆	Rs1 ₆	~ ₆	124 ₇						

Operation:

Cause = Rs1

PUSH SR

PUSH IP

EIP = IP

IP = vector at (kernel_vector[operating mode+1])

Execution Units: Branch**Clock Cycles:****Exceptions:** none**Notes:****Sample Code:**

CSR RD r127, CAUSE, \$0	; get current cause code
ESC r127	; escalate the exception to the next mode ; exception processing will return here

FENCE – SYNCHRONIZATION FENCE

Description:

All instructions for a particular unit before the FENCE are completed and committed to the architectural state before instructions of the unit type after the FENCE are issued. This instruction is used to ensure that the machine state is valid before subsequent instructions are executed.

Instruction Format:

47	41	40	27	26	24	23	16	15	12	11	8	7	6	0
0 ₇		~ ₁₄		Op ₃		Mask _{7..0}		Aft ₄		Bef ₄	~		123 ₇	

Mask Bit	Access		
0	Wr	Before	
1	Rd		
2	Out		
3	In		
4	Wr	After	
5	Rd		
6	Out		
7	In		

Aft ₄ / Bef ₄	Unit
0	MEM
1	ALU
2	FPU
3	Branch
4 to 14	Reserved
15	All units

Op ₃	Fence Type
0	Normal
1 to 6	reserved
7	TSO fence

IRQ – GENERATE INTERRUPT

Description:

Generate interrupt. This instruction invokes the system exception handler. The return address is pushed onto an internal stack.

The return address stored is the address of the interrupt instruction, not the address of the next instruction. To call system routines use the [SYS](#) instruction.

The level of the interrupt is checked and if the interrupt level in the instruction specified by Rs1 is less than or equal to the current interrupt level then the instruction will be ignored. The cause code is set the value of Rs2.

Instruction Format:

47	41	40 38	37	35	34	28	27	21	20	14	13	7	6	0
2 ₇	Ms ₃	~ ₃	~ ₇		Rs2 ₇	Rs1 ₇	0 ₇		124 ₇					

Operation:

PUSH SR

PUSH IP

CAUSE = Rs2

IP = vector(tvec[om])

Execution Units: Branch

JMPX – JUMP TO EXCEPTION HANDLER

Description:

This instruction jumps to an exception routine by transferring program execution to the address specified as the sum of a displacement and register Rs1. The instruction pointer and status register are pushed onto an internal stack.

Formats Supported: RTE

39	19	18 17	16	12	11 10	9	7	6	0
Disp ₂₁		2 ₂	Ra ₆		3 ₂	~ ₃		35 ₇	

Operation:

PUSH IP on internal stack

PUSH SR on internal stack

$$IP = Rs1 + \text{disp}$$

Execution Units: Branch**Exceptions:** none**Notes:**

MEMDB – MEMORY DATA BARRIER

Description:

All memory accesses before the MEMDB command are completed before any memory accesses after the data barrier are started. This is an alternate mnemonic for the [FENCE](#) instruction.

Instruction Format:

31	30	29	27	26	24	23	16	15	12	11	8	7	6	0
Fmt ₂	Pr ₃	0 ₃		255 _{7..0}		0 ₄	0 ₄	~		114 ₇				

Clock Cycles: 1**Execution Units:** Memory

MEMSB – MEMORY SYNCHRONIZATION BARRIER

Description:

All instructions before the MEMSB command are completed before any memory access is started. This is an alternate mnemonic for the [FENCE](#) instruction.

Instruction Format:

31	30	29	27	26	24	23	16	15	12	11	8	7	6	0
Fmt ₂	Pr ₃	0 ₃		192 _{7..0}		0 ₄	15 ₄	~		114 ₇				

Clock Cycles: 1**Execution Units:** Memory

PFI – POLL FOR INTERRUPT

Description:

The poll for interrupt instruction polls the interrupt status lines and performs an interrupt service if an interrupt is present. Otherwise, the PFI instruction is treated as a NOP operation. Polling for interrupts is performed by managed code. PFI provides a means to process interrupts at specific points in running software. Rd is loaded with the cause code in the low order eight bits, and the interrupt level is set in bits twelve to fourteen of the register.

Instruction Format: OSR2

47	41	40	38	37	35	34	33	28	27	26	21	20	19	14	13	12	7	6	0
1 ₇	~ ₃	~ ₃	~	~ ₆	Nd	Rd ₆	124 ₇												

Clock Cycles: 1 (if no exception present)

Operation:

```
if (irq <> 0)
    Rd[7:0] = cause code
    Rd[14:12] = irq level
    PMSTACK = (PMSTACK << 4) | 6
    EIP = IP
    ESR = SR
    IP = tvec[3]
```

Execution Units: Branch

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 the bitwise union of the PL₈ field and the value in register Rs1. Otherwise the privilege level will remain unchanged.

Instruction Format: EX

47	41	4039	38	36	35	34	33	29	28	22	21	20	19	15	14	13	12	10	12	8	7	6	0
26 ₇	~ ₂	~ ₃	S	PL ₇	13 ₅		PL _{6...0}	Na	Va	Ra ₅	0	0	~ ₃	Tm ₂	v	Opc ₇							

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.

RTE – RETURN FROM EXCEPTION

Description:

This instruction returns from an exception routine by transferring program execution to the address stored in an internal stack. This instruction may perform a two-up level return.

Formats Supported: RTE

47	35	34	33	28	27	26	21	20	19	16	15	14	13	12	7	6	0
0 ₁₃	~	~ ₆	~	~ ₆	~	~ ₃	1 ₂	~ ₂	Offs ₆	35 ₇							

Formats Supported: RTE – Two up level return.

47	35	34	33	28	27	26	21	20	19	16	15	14	13	12	7	6	0
0 ₁₃	~	~ ₆	~	~ ₆	~	~ ₃	2 ₂	~ ₂	Offs ₆	35 ₇							

Operation:

Optionally pop the status register and always pop the instruction pointer from the internal stack. Add Const bytes to the instruction pointer. If returning from an application trap the status register is not popped from the stack.

Execution Units: Branch**Exceptions:** none**Notes:**

SYS – SYSTEM CALL

Description:

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

Instruction Format:

47	41	40 38	37	35	34	33	28	27	26	21	20	19	14	13	12	7	6	0
3 ₇	Ms ₃	~ ₃	~	~ ₆	0	0 ₆	124 ₇											

Operation:

PUSH SR onto internal stack

PUSH IP + 6 onto internal stack

IP = tvec[3]

Execution Units:

Branch

Clock Cycles:

Exceptions: none

Notes:

STOP – STOP PROCESSOR

Description:

The STOP instruction waits for an external interrupt to occur before proceeding. While waiting for the interrupt, the processor clock is slowed down or stopped placing the processor in a lower power mode. A sixteen-bit constant is provided for the CPU's stop instruction.

Instruction Format: STOP

47	41	40	38	37	35	34	33	28	27	26	21	20	19	14	13	12	7	6	0
0 ₇	Ms ₃	Op ₃	N3	Rs3 ₆	N2	Rs2 ₆	N1	Rs1 ₆	Nd	Rd ₆	124 ₇								

Clock Cycles: 1 (if no exception present)

Execution Units: Branch

TRAP – TRAP

Description:

Execute trap. The data field is loaded into the specified destination register, Rd. The trap number to execute comes from the contents of register Rs1. The trap number is loaded into the CAUSE CSR. The trap number must be between 1 and 255. Trap numbers below 64 are reserved for the system. Trap numbers 64 and above may be used by applications.

Trap routines should return using an [RTE](#) instruction.

Instruction Format:

47	41	40 38	37 35	34 31	30	25	24	19	18	13	12	7	6	0
\sim_7	\sim_3	\sim_3	\sim_4	\sim_6										

Clock Cycles: 1

Operation:

The program counter and the status register are pushed on an internal stack. Next the vector is fetched from the exception vector table and jumped to.

CAUSE = Rs1

PUSH SR

PUSH IP

EIP = IP + 6

IP = kernel_vector[operating mode]

SUBROUTINE INSTRUCTIONS

ENTER – ENTER ROUTINE

Description:

This instruction is used for subroutine linkage at entrance into a subroutine. First it pushes the frame pointer and return address onto the safe stack, next the stack pointer is loaded into the frame pointer and finally the stack space is allocated. This instruction is code dense, replacing five or more other instructions with a single instruction.

A maximum of 16MB may be allocated on the stack. To allocate more a second ADD instruction must be used.

Note that the constant must be a negative number and a multiple of eight.

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: RI

47	25	24	19	18	13	12	7	6	0
Immediate _{25..3}	LR ₆	FP ₆	SP ₆	52 ₇					

Operation:

$$\text{SafeSP} = \text{SafeSP} - 32$$

$$\text{Memory}[\text{SafeSP}] = \text{FP}$$

$$\text{Memory8}[\text{SafeSP}] = \text{LR}$$

$$\text{FP} = \text{SP}$$

$$\text{SP} = \text{SP} + \text{constant}$$

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 moves the frame pointer to the stack pointer deallocating any stack memory allocations. Next the frame pointer and return address are popped off the safe stack. The stack pointer is adjusted by the amount specified in the instruction. Then a jump is made to the return address. This instruction is code dense, replacing six other instructions with a single instruction.

Instruction Format: EXIT

47	25	24	19	18	13	12	7	6	0
Immediate _{25...3}	LR ₆	FP ₆	SP ₆	53 ₇					

Operation:

$$SP = FP$$

$$FP = \text{Memory}[\text{SafeSP}]$$

$$LR = \text{Memory8}[\text{SafeSP}]$$

$$\text{SafeSP} = \text{SafeSP} + 32$$

$$SP = SP + \text{Constant}$$

$$IP = LR$$

CONSTANT SUPPORT

CZ – CONSTANT ZONE

Description:

Embed a constant zone into the instruction stream. Constant zone instructions are 48-bits in size (the size of an instruction). Constant zones are concatenated together to form a larger zone. Up to four constant zones may be concatenated together for up to 160 bits of constant data.

The instruction prior to the zone references constant data within the zone.

The zone may contain any number of constants which are a multiple of 16-bits in size in any order.

Constant zones typically follow inline after instructions requiring constants. Constants in the zone are referenced in the instruction as offsets from the instruction. The four-bit constant location field in the instruction is multiplied by sixteen to determine the bit location of the constant in the zone. A constant size field in the instruction determines the number of constant bits to extract from the zone.

Constant zones are detected by hardware and the corresponding instruction space is marked as non-executable.

Instruction Format: CZ

47	8	7	6	0
Constant Zone ₄₀	0	127 ₇		

Operation:

NOP

Execution Units:

Clock Cycles:

Exceptions:

Notes:

MODIFIERS / POSTFIXES

ATOM

Description:

Treat the following sequence of instructions as an “atom”. The instruction sequence is executed with interrupts masked off. Interrupts may be disabled for up to twelve instructions. The non-maskable interrupt may not be masked.

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

47	21	19	8	7	6	0
~28		Mask ₁₂	0	122 ₇		

Scope Modifier	Mask Bit	
	0	Instruction zero
	1	Instruction one
	2	Instruction two
	3	Instruction three
	4	Instruction four
	5	Instruction five
	6	Instruction six
	7	Instruction seven
	8	Instruction eight
	9	Instruction nine
	10	Instruction ten

Assembler Syntax:

Example:

```
ATOM "MMMMMM"  
LOAD a0,[a3]  
SLT t0,a0,a1  
PRED t0,"TTF"  
STORE a2,[a3]  
LDI a0,1
```

LDI a0,0

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

PRED

Description:

Apply the predicate to following instructions according to a bit mask. The predicate may be applied to a maximum of eight instructions. The PRED instruction may be applied to vector instructions and act to mask off operation of specific lanes of the vector. If the ‘Z’ bit is set, destination 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

47	46	45	44	29	28	27	26	22	21	20	19	15	14	13	12	8	7	6	0
~2	Z			Mask _{15..0}	0	Vn ₅	0	Rp ₅	~2	~5	0	121 ₇							

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

Mask Bit	Meaning
00	Always execute (ignore predicate)
01	Execute only if predicate bit is true
10	Execute only if predicate bit is false

Assembler Syntax:

After the instruction mnemonic the register containing the predicate flags is specified. Next a character string containing ‘T’ for True, ‘F’ for false, or ‘I’ for ignore for the next five instructions is present.

Example:

```
PRED r2,"TIFIII"  
; execute one if true, ignore one, next execute if false, one after always execute  
MUL r3,r4,r5      ; executes if True  
ADD r6,r3,r7      ; always executes  
ADD r6,r6,#1234    ; executes if FALSE  
DIV r3,r4,r5      ; always executes
```

REXT – EXTENDED REGISTER SELECTION

Description:

This postfix extends the register selection. A second destination register and two more source operand registers are specified.

Instruction Format: REXT

47	41	40 38	37	31	30 28	27 25	24	19	18	13	12	7	6	0
\sim_7	Ms_3	\sim_5	V_3	N_3	$Rs5_6$	$Rs4_6$	$Rd2_6$	120_7						

ROUND

Description:

Set the rounding mode for following eight instructions. Note that postfixes do not count as instructions.

Round modes are specified with a field in the instruction now.

Instruction Format: ATOM

39 38	37 35	34 31	30	7	6	0
Fmt ₂	Pr ₃	~ ₄		Mask ₂₄	116 ₇	

Scope Modifier	Mask Bit	
	0 to 2	Instruction zero
	3 to 5	Instruction one
	6 to 8	Instruction two
	9 to 11	Instruction three
	12 to 14	Instruction four
	15 to 17	Instruction five
	18 to 20	Instruction six
	21 to 23	Instruction seven

BINARY FLOAT ROUNDING MODES

Rm3	Rounding Mode
000	Round to nearest ties to even
001	Round to zero (truncate)
010	Round towards plus infinity
011	Round towards minus infinity
100	Round to nearest ties away from zero
101	Reserved
110	Reserved
111	Use rounding mode in float control register

Assembler Syntax:

Example:

OPCODE MAPS

QUPLS ROOT OPCODE

	0	1	2	3	4	5	6	7
0x	0 BRK	1 CAP {CAP}	2 {R3.128}	3 {BFLD}	4 ADDI	5 SUBFI	6 MULI	7 CSR
	8 ANDI	9 ORI	10 EORI	11 CMPI	12 {EXTD}	13 DIVI	14 MULUI	15 ADDIPI
1x	16 {SHIFT}	17 {DFLT}	18 {PST}	19 CMPUI	20 LOADA	21 DIVUI	22 BLEND	23 MOVMR
	24 BccU.8	25 BccU.16	26 BccU.32	27 BccU.64	28 Bcc.8	29 Bcc.16	30 Bcc.32	31 Bcc.64
2x	32 BRA	33 JMP addr	34 CAP CJMP adr	35 RTD JMPN	36 {CAP}	37 JMP reg	38 CJMP reg	39 CAP CJSR reg
	BSR	JSR addr	CJSR adr	IRET JMPX	JSRN	JSR reg		
3x	40 FBcc.16	41 FBcc.32	42 FBcc.64	43 FBcc.128	44 DFBcc.128	45 PBcc	46	47 CHK
	48 {FLTP.16}	49 {FLTP.32}	50 {FLTP.64}	51 {FLT.128}	52 ENTER	53 LEAVE	54 PUSH	55 POP
4x	56 {FLT.16}	57 {FLT.32}	58 {FLT.64}	59 {FLT.128}	60	61	62	63
	64 LDB	65 LDBU	66 LDW	67 LDWU	68 LDT	69 LDTU	70 LDO	71 Vector LDV
5x	72 CLOAD	73 FLDH	74 CACHE	75 FLDS	76 FLDD	77	78	79 Vector LDVN
	80 STB	81 STW	82 STT	83 STO	84 STI	85 CSTORE	86 STPTR	87
6x	88 Vector STV	89 Vector STVN	90 V2P	91 VV2P	92 AMO	93 CAS	94	95
	96 FDP.16	97 FDP.32	98 FDP.64	99 FDP.128	100	101	102	103
7x	104 {R3.8}	105 {R3.16}	106 {R3.32}	107 {R3.64}	108 BFIND	109 BCMP	110	111 {BLOCK}
	112 SIMD {R3P.8}	113 SIMD {R3P.16}	114 SIMD {R3P.32}	115 SIMD {R3P.64}	116 Vector R3P	117 Vector FLTP	118 Vector VFDP	119
	120 REXT	121 PRED	122 ATOM	123 FENCE	124 STOP / PFI IRQ / SYS	125 FEX	126	127 NOP / CZ

{CAP} OPERATIONS

	0	1	2	3	4	5	6	7
	8 CRetd	9	10	11	12 CInvoke	13	14	15
	16	17	18	19	20	21	22	23
	24	25	26	27	28	29	30	31
	32 CAndPerm	33 CBuildCap	34 CCopyType	35 CIncOffs	36 CSeal	37 CSetAddr	38 CSetBounds	39 CSetFlags
	40 CSetHigh	41 CSetOffs	42 CSpeicalRW	43 CUnseal	44	45	46	47
	48	49	50	51	52	53	54	55
	56	57	58	59	60	61	62	63
	64 CClearTag	65 CGetFlags	66 CGetHlgh	67 CGetLen	68 CGetOffs	69 CGetPerms	70 CGetTag	71 CGetTop
	72 CGetType	73 CLoadTags	74 CAlignMsk	75 CRoundLen	76 CSealEntry	77 CGetBase	78	79
	80	81	82	83	84	85	86	87
	88	89	90	91	92	93	94	95
	96	97	98	99	100	101	102	103
	104	105	106	107	108	109	110	111
	112	113	114	115	116	117	118	119
	120	121	122	123	124	125	126	127

{R1} OPERATIONS

	0	1	2	3	4	5	6	7
0x	0 CNTLZ	1 CNTLO	2 CNTPOP	3 LOPOS	4 SQRT	5 REVBIT	6 CNTTZ	7 NOT
	8 NNA_TRIG	9 NNA_STAT	10 NNA_MFACT	11	12 MKBOOL	13 REX	14 SM3P0	15 SM3P1
1x	16	17	18 AES64DS	19 AES64DSM	20 AES64ES	21 AES64ESM	22 AES64IM	23
	24 SHA256 SIG0	25 SHA256 SIG1	26 SHA256 SUM0	27 SHA256 SUM1	28 SHA512 SIG0	29 SHA512 SIG1	30 SHA512 SUM0	31 SHA512 SUM1
2x	32 BNDX	33	34 CNTNPOP					

{R3, R3P} OPERATIONS

	0	1	2	3	4	5	6	7
0	0 AND	1 OR	2 EOR	3 CMP	4 ADD	5 SUB	6 CMPU	7 CPUID
	8 NAND	9 NOR	10 ENOR	11 {CMOV}	12 SATADD	13	14	15 MOVE
1	16 MUL	17 DIV	18 {MINMAX}	19 MULU	20 DIVU	21 MULSU	22 DIVSU	23 {MINMAXU}
	24 MULW	25 MOD	26 {R1}	27 MULUW	28 MODU	29 MULSUW	30 MODSU	31
2	32 PTRDIF	33 MUX	34 BMM	35 BMAP	36 DIF	37 CHARNDX	38 CHARNDX	39 CHARNDX
	40 NNA MTWT	41 NNA MTIN	42 NNA MTBIAS	43 NNA MTFB	44 NNA MTMC	45 NNA MTBC	46	47
3	48 V2BITS	49 BITS2V	50 VEX	51 VEINS	52 VGNDX	53	54 VSHLV	55 VSHRV
	56 V2BITSP	57 PBITS2V	58 VSETMASK	59 VMFILL	60	61	62 VSHLVI	63 VSHRVI
4	64 AES64K1I	65 AES64KS2	66 SM4ED	67 SM4KS	68	69	70 CLMUL	71
	72	73	74	75	76	77	78	79
5	80 ROL_OP	81 ROR_OP	82 ASR	83 ASL_OP	84 LSR_OP	85 NASR	86 NLSR	87 DIVMOD
	88 CLR	89 COM	90 EXTZ	91 EXT	92 DEP	93 SET	94	95 DIVMODU
6	96 REDAND	97 REDOOR	98 REDEOR	99 REDMINU	100 REDSUM	101 REDMAXU	102 REMIN	103 REDMAX
	104 PEEKQ	105 POPQ	106 PUSHQ	107 RESETQ	108 STATQ	109	110 READQ	111 WRITEQ
7	112 {MINMAX2}	113 {MINMAXU2}	114	115	116	117	118	119
	120 SEQ	121 SNE	122 SLT	123 SLE	124 SLTU	125 SLEU	126	127 MVVR

{EXTD} EXTENDED PRECISION INSTRUCTIONS

	0	1	2	3	4	5	6	7
12	0 ADC	1 SBC	2 ASLC	3 ASRC	4 LSRC	5 BPACK	6 VSHLV	7 VSHRV

{CMOV} CONDITIONAL MOVE INSTRUCTIONS

	0	1	2	3	4	5	6	7
11	0 CMOVNZ	1 CMOVLTZ	2 CMOVLEZ	3 CMOVEVN	4 CMOVEVN	5 CMOVEVN	6 CMOVEVN	7 CMOVEVN

{FLT, FLTP} OPERATIONS

	0	1	2	3	4	5	6	7
16	0 FNOP	1 {FMxx}	2 FMIN	3 FMAX	4 FADD	5 FSUB	6 FMUL	7 FDIV
	8 FSEQ	9 FSNE	10 FSLT	11 FSLE	12 	13 FCMP	14 FNXT	15 FREM
	16 FSGNJ	17 FSGNIN	18 FSGNIX	19 	20 FSCALEB	21 FMASK	22 FSTAT	23
	24 	25 	26 	27 	28 	29 	30 	31
16	32 FABS	33 FNEG	34 FTOI	35 ITOF	36 FCONST	37 	38 FSIGN	39 FSIG
	40 FSQRT	41 FCVTS2D	42 FCVTS2Q	43 FCVTD2Q	44 FCVTH2S	45 FCVTH2D	46 ISNAN	47 FINITE
	48 FCVTQ2H	49 FCVTQ2S	50 FCVTQ2D	51 	52 FCVTH2Q	53 FTRUNC	54 FRSQRTE	55 FRES
	56 	57 FCVTD2S	58 	59 	60 	61 	62 FCLASS	63 FRM
16	64 FSIN	65 FCOS	66 FTAN	67 	68 	69 	70 	71
	72 	73 	74 FATAN	75 	76 	77 	78 	79
	80 FSIGMOID	81 	82 	83 	84 	85 	86 	87
	88 	89 	90 	91 	92 	93 	94 	95

{DFLT3} OPERATIONS

	0	1	2	3	4	5	6	7
17	0 FMA	1 FMS	2 FNMA	3 FNMS	4 VFMA	5 VFMS	6 VFNMA	7 VFNMS
	8 {DFLT2}	9	10	11	12 {VDFLT2}	13 {VSFLT2}	14	15

{DFLT} OPERATIONS

	0	1	2	3	4	5	6	7
17	0 DFSCALEB	1 {DFLT1}	2 DFMIN	3 DFMAX	4 DFADD	5 DFSUB	6 DFMUL	7 DFDIV
	8 DFSEQ	9 DFSNE	10 DFSLT	11 DFSLE	12	13 DFCMP	14 DFNXT	15 DFREM
	16 DFSGNJ	17 DFSGNJJN	18 DFSGNJJX	19	20	21	22	23
	24	25	26	27	28	29	30 FNMUL	31
17	32	33 DFNEG	34 DFTOI	35 ITODF	36	37	38	39 DFSIG

{AMO} – ATOMIC MEMORY OPS

	0	1	2	3	4	5	6	7
92	0 AMOADD	1 AMOAND	2 AMOOR	3 AMOEOR	4 AMOMIN	5 AMOMAX	6 AMOSWAP	7
	8 AMOASL	9 AMOLSR	10 AMOROL	11 AMOROR	12 AMOMINU	13 AMOMAXU	14	15

{EX} EXCEPTION INSTRUCTIONS

	0	1	2	3	4	5	6	7
2	0 IRQ	1	2 FTX	3 FCX	4 FDX	5 FEX	6	7 ESC
	8	9	10	11	12	13	14	15

MPU HARDWARE

PIC – PROGRAMMABLE INTERRUPT CONTROLLER

OVERVIEW

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

The programmable interrupt controller presents an interrupt signal bus to the CPU core(s). The PIC may be used in a multi-CPU system as a shared interrupt controller. The PIC can guide the interrupt to the specified core(s). The PIC 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. PIC supports 63 different priority levels.

The PIC 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 PIC 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		
MSI message specifications						
040	32	R/W		MSI “data” field Bits 0 to 11 = vector number Bits 14, 15 = needed operating mode Bits 16 to 31 = data field for interrupt table		
044	32	R/W		MSI “adr” field Bits 0 to 15 = bus/device/function Bit 16 to 28 = segment Bit 29 to 31 = software stack needed		
048	32	R/W		MSI “tid” field Bits 0 to 5 = interrupt priority Bits 7 to 12 = interrupt core number		
04C	32	R/W		MSB= IRQ trigger		
080 to OFF	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 PIC 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.

PIT – PROGRAMMABLE 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 PIT is a 64-bit peripheral. The PIT 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 PIT is given a 64kB-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 \$FFFFFFFEE4xxxx

PIT#2 is located at \$FFFFFFFEE5xxxx

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 OFF	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 the I/O address select input, cs_io, will have bits 24 and above in its decode and that a 64kB page is required for the device, matching the MMU page size.

The controller will respond with a mask of 0x00FF0000 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 PIT 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 PIT 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 are 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 to 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.

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.

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.

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 dealt 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.

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).

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

An acronym 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 Francisco, 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 Asanović 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

IBM® is a registered trademark of International Business Machines Corporation. Intel® is a registered trademark of Intel Corporation. HP® is a registered trademark of Hewlett-Packard Development Company. "SPARC® is a registered trademark of SPARC International, Inc.

WISHBONE COMPATIBILITY DATASHEET

The Qupls 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:	128 bit
Data port, granularity:	8 bit
Data port, maximum operand size:	128 bit
Data transfer ordering:	Little Endian
Data transfer sequencing	any (undefined)
Clock frequency constraints:	tm_clk_i must be \geq 10MHz
Supported signal list and cross reference to equivalent WISHBONE signals	Signal Name: WISHBONE Equiv. Resp.ack_i ACK_I Req.adr_o(31:0) ADR_O() clk_i CLK_I resp.dat(127:0) DAT_I() req.dat(127:0) DAT_O() req.cyc CYC_O req.stb STB_O req.wr WE_O req.sel(7:0) SEL_O

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

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.

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	
Stb	1	Data strobe	
We	1	Write enable	
Asid	16	Address space id	
Vadr	32/64	Virtual address	
Padr	32/64	Physical address	
Sel	16	Byte lane selects	
Data1	128	First data item	
Data2	128	Second data item	
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	1	An error occurred	
Pri	4	Transaction priority	
Adr	32/64	Physical address	
Dat	128	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.

Ordinal		
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_ASIL	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	