Overview

CS01 is a thirty-two-bit processor modelled after the RISCV ISA (RV32I). Only a subset of the full RISCV instruction set is implemented. The processor features 32, 32-bit integer registers and 32, 32-bit floating-point registers. The processor has the most prominent features from real processors with some of the more complex details left out.

Programming Model

	Registers					
	31	0			31	0
	x0 / zero				f0 / zero	
	x1 / ra				f1	
	x2 / sp				…	
	x3 / gp					
	x4 / tp					
	x5 / t0 / alternate link					
	x6-x7 / t1-t2				…	
	x8 / fp / s0				…	
	x9 / s1				…	
	x10-x11 / a0-a1				…	
	x12-x17 / a2-a7				…	
	x18-x27 / s2 – s11				f30	
	x28-x31 / t3-t6				f31	
	pc					

x? refers to an integer register. f? refers to a floating-point register

Registers x0 and f0 are always zero. f0 is positive zero.

x14 / sp is the stack pointer.

pc is the program counter.

Nomenclature

The sizes of values are referred to as the following:

Size	Alternate	
1	byte	byte
2	wyde	half-word
4	tetra	word
5	penta	
8	octa	double-word
10	deci	
16	hexi	

Operating Modes

The core supports two operating modes, user and machine mode. Some instructions are available only in machine mode. Memory management including segmentation and paging address translation are applied only to user mode addressing. There is a separate integer register set for each mode.

Memory Access Alignment

The core supports unaligned data memory access; however, it does not guarantee the atomicity of the access.

Supported CSR's

The following CSR's are supported.

Number	Name	Description
000h	ustatus	user status register
001h	fflags	floating-point accrued exceptions
002h	frm	dynamic rounding mode
003h	fcsr	floating point control and status
C00h	cycle	cycle counter for RDCYCLE instruction
C01h	time	time for timer instruction
C02h	instret	instructions retired
C80h	cycleh	upper 32 bit of cycle counter
C81h	timeh	upper 32 bits of time
C82h	instreth	upper 32 bits of instructions retired
F00h	mcpuid	CPU description
F01h	mimpid	vendor ID and version number
F10h	mhartid	hardware thread id
300h	mstatus	machine status
301h	mtvec	trap handler base address ($FFFC0000)
304h	mie	machine interrupt enable
321h	mtimecmp	wall clock time compare
701h	mtime	wall clock time (same as reg 0xC01)
741h	mtimeh	upper 32 bits of wall clock time
340h	mscratch	scratchpad register
341h	mepc	machine exception program counter
342h	mcause	machine trap cause
343h	mbadaddr	machine bad address
344h	mip	interrupt pending
181h	asid	address space identifier
7C0h	regset	register set selection
7C1h	sema	machine semaphores

7C2h	tid	task id

Instruction Set Formats

Bits	31 25	24 20	19 15	14 12	11 7	6 0	
LUI	imm$_{31..12}$				Rd	55	
AUIPC	imm$_{31..12}$				Rd	23	
JAL	20 imm$_{10..1}$	11	imm$_{19..12}$		Rd	111	
CALL[1]	20 imm$_{10..1}$	11	imm$_{19..12}$		1	111	
JALR	imm$_{11..0}$		Rs1	0	Rd	103	
RET[1]	0		1	0	0	103	
BEQ	imm$_{12.10..5}$	Rs2	Rs1	0	imm$_{4..1.11}$	99	
BNE	imm$_{12.10..5}$	Rs2	Rs1	1	imm$_{4..1.11}$	99	
BLT	imm$_{12.10..5}$	Rs2	Rs1	4	imm$_{4..1.11}$	99	
BGE	imm$_{12.10..5}$	Rs2	Rs1	5	imm$_{4..1.11}$	99	
BLTU	imm$_{12.10..5}$	Rs2	Rs1	6	imm$_{4..1.11}$	99	
BGEU	imm$_{12.10..5}$	Rs2	Rs1	7	imm$_{4..1.11}$	99	
BRA[1]	imm$_{12.10..5}$	0	0	0	imm$_{4..1.11}$	99	
LB	imm$_{11..0}$		Rs1	0	Rd	3	LDB
LH	imm$_{11..0}$		Rs1	1	Rd	3	LDW
LW	imm$_{11..0}$		Rs1	2	Rd	3	LDT
LBU	imm$_{11..0}$		Rs1	4	Rd	3	LDBU
LHU	imm$_{11..0}$		Rs1	5	Rd	3	LDWU
FLW	imm$_{11..0}$		Rs1	2	FRd	7	LDFT
SB	imm$_{11..5}$	Rs2	Rs1	0	Imm$_{4..0}$	35	STB
SH	imm$_{11..5}$	Rs2	Rs1	1	Imm$_{4..0}$	35	STW
SW	imm$_{11..5}$	Rs2	Rs1	2	Imm$_{4..0}$	35	STT
FSW	imm$_{11..5}$	FRs2	Rs1	2	Imm$_{4..0}$	39	STFT
ADDI	imm$_{11..0}$		Rs1	0	Rd	19	
NOP[1]	0		0	0	0	19	
SLTI	imm$_{11..0}$		Rs1	2	Rd	19	
SLTUI	imm$_{11..0}$		Rs1	3	Rd	19	
XORI	imm$_{11..0}$		Rs1	4	Rd	19	EORI
ORI	imm$_{11..0}$		Rs1	6	Rd	19	
LDI[1]	imm$_{11..0}$		0	6	Rd	19	
ANDI	imm$_{11..0}$		Rs1	7	Rd	19	
SLLI	0	shamt	Rs1	1	Rd	19	SHLI
SRLI	0	shamt	Rs1	5	Rd	19	SHRI
SRAI	16	shamt	Rs1	5	Rd	19	ASRI
ADD	0	Rs2	Rs1	0	Rd	51	
SUB	32	Rs2	Rs1	0	Rd	51	
MUL	1	Rs2	Rs1	0	Rd	51	
SLL	0	Rs2	Rs1	1	Rd	51	SHL
SLT	0	Rs2	Rs1	2	Rd	51	
SLTU	0	Rs2	Rs1	3	Rd	51	
XOR	0	Rs2	Rs1	4	Rd	51	EOR
SRL	0	Rs2	Rs1	5	Rd	51	SHR
SRA	32	Rs2	Rs1	5	Rd	51	ASR
OR	0	Rs2	Rs1	6	Rd	51	
MOV[1]	0	0	Rs1	6	Rd	51	
AND	0	Rs2	Rs1	7	Rd	51	
FADD	0 00	FRs2	FRs1	rm	FRd	83	
FSUB	1 00	FRs2	FRs1	rm	FRd	83	
FMUL	2 00	FRs2	FRs1	rm	FRd	83	

FDIV	3	00	FRs2	FRs1	rm	FRd	83	
FMIN	5	00	FRs2	FRs1	0	FRd	83	
FMAX	5	00	FRs2	FRs1	1	FRd	83	
FSQRT	11	00	0	FRs1	rm	FRd	83	
FSGNJ	16	00	FRs2	FRs1	0	FRd	83	
FMOV[1]	16	00	FRs1	FRs1	0	FRd	83	
FSGNJN	16	00	FRs2	FRs1	1	FRd	83	
FNEG[1]	16	00	FRs1	FRs1	1	FRd	83	
FSGNJX	16	00	FRs2	FRs1	2	FRd	83	
FABS[1]	16	00	FRs1	FRs1	2	FRd	83	
FEQ	20	00	FRs2	FRs1	2	Rd	83	
FLT	20	00	FRs2	FRs1	1	Rd	83	
FLE	20	00	FRs2	FRs1	0	Rd	83	
FCVT.W.S	24	00	0	FRs1	rm	Rd	83	
FCVT.WU.S	24	00	1	FRs1	rm	Rd	83	
FCVT.S.W	25	00	0	Rs1	rm	FRd	83	
FCVT.S.WU	25	00	1	Rs1	rm	FRd	83	
FMV.X.S	28	00	0	FRs1	0	Rd	83	
FCLASS	28	00	0	FRs1	1	Rd	83	
FMV.S.X	30	00	0	Rs1	0	FRd	83	
FENCE	0	pred	succ	0	0	0	15	
FENCE.I	0	0	0	0	1	0	15	
ECALL	000h			0	0	0	115	
EBREAK	001h			0	0	0	115	
ERET	100h			0	0	0	115	
MRET	302h			0	0	0	115	
RDCYCLE	C00h			0	1	Rd	115	
RDCYCLEH	C80h			0	1	Rd	115	
RDTIME	C01h			0	1	Rd	115	
RDTIMEH	C81h			0	1	Rd	115	
RDINSTRET	C02h			0	1	Rd	115	
RDINSTRETH	C82h			0	1	Rd	115	
CSRRW	CSR_{12}			Rs1	1	Rd	115	
CSRRS	CSR_{12}			Rs1	2	Rd	115	
CSRRC	CSR_{12}			Rs1	3	Rd	115	
CSRRWI	CSR_{12}			imm_5	5	Rd	115	
CSRRSI	CSR_{12}			imm_5	6	Rd	115	
CSRRCI	CSR_{12}			imm_5	7	Rd	115	
WFI	101h			0	0	0	115	
Custom Instructions								
PFI[2]	103h			0	0	0	115	
MVSEG[2]	0		Rs2	Rs1	0	Rd	13	
MVMAP[2]	1		Rs2	Rs1	0	Rd	13	
PALLOC[2]	4		0	0	0	Rd	13	
PFREE[2]	5		0	Rs1	0	0	13	
PFREEALL[2]	6		0	0	0	0	13	
PSTAT	7		0	Rs1	0	Rd	13	
SETTO	8		Rs2	Rs1	0	0	13	
GETTO	9		0	Rs1	0	Rd	13	
GETZL	10		0	0	0	Rd	13	
DECTO	11		0	0	0	0	13	
PUSHQ	12		Rs2	Rs1	0	0	13	
POPQ	13		0	Rs1	0	Rd	13	

PEEKQ	14		0	Rs1	0	Rd	13	
MRET2	16		0	0	0	0	13	
GCSUB	32		Rs2	Rs1	0	Rd	13	
LxX	Rs3	Sc	F3L	Rs1	1	Rd	13	
SxX	Rs3	Sc	Rs2	Rs1	2	F3S	13	
GCADDI	$Imm_{11..0}$			Rs1	3	Rd	13	
MAJ	Rs3	0	Rs2	Rs1	4	Rd	13	

1. an extended mnemonic for another instruction
2. instruction is a green-field extension to the RISCV instruction set

Integer Instructions

ADD – Addition

Description:

Add two values using two's complement addition, which are in Rs1 and Rs2 or an immediate value and place the sum in the destination register Rd.

Instruction Format: R2, RI

Exceptions: none

AND – Bitwise And

Description:

Bitwise 'and' two values which are in Rs1 and Rs2 or an immediate value and place the result in the destination register Rd. A bitwise operation operates on each bit of the register individually. By carefully managing values the bitwise and may also be used as a logical and.

Instruction Format: R2, RI

Exceptions: none

AUIPC – Add Upper Immediate to PC

Description:

This instruction adds the upper 20 bits of the program counter to an immediate supplied by the instruction and stores the result in the destination register Rd. This instruction may be used to generate addresses relative to the program counter.

EOR – Bitwise Exclusive Or

Description:

This is an alternate mnemonic supported by the assembler for the XOR instruction. Bitwise 'exclusive or' two values which are in Rs1 and Rs2 or an immediate value and place the result in the destination register Rd. A bitwise operation operates on each bit of the register individually. By carefully managing values the bitwise eor may also be used as a logical eor.

Instruction Format: R2, RI

Exceptions: none

LDI – Load Immediate

Description:

This is an alternate mnemonic for the 'or' instruction where Rs1 is assumed to be x0. This has the effect of simply loading the constant into integer register Rd.

Instruction Format: RI

Exceptions: none

LUI – Load Upper Immediate

Description:

The LUI instruction sets the upper 20 bits of the destination register Rd to the constant supplied in the instruction and zeros out the lower 12 bits of the destination register.

MOV – Move Register

Description:

There are two instructions that share the mov mnemonic, one for moving within a register set, and one for moving between register sets. Moving between register sets is implemented as a custom instruction not available to the user operating level.

If moving within the same register set, this is an alternate mnemonic for the ‘or’ instruction where Rs2 is assumed to be x0. The value in Rs1 is then simply copied to destination register Rd.

If moving between register sets the s field of the instruction determines the source register set, while the d field of the instruction determines the destination register set. Code 00b is for the user register set, code 11b for the machine register set.

Instruction Format: R2, MOV

Exceptions: none

Examples:

```
mov     $a0,$v0          ; move v0 to a0 in same register set
mov     u:$v0,m:$v0      ; move machine register v0 to user register v0
```

Notes:

When moving between register sets a register set indicator prefixes the register to move. Register set prefixes are m: for machine, and u: for user.

OR – Bitwise Inclusive Or

Description:

Bitwise ‘inclusive or’ two values which are in Rs1 and Rs2 or an immediate value and place the result in the destination register Rd. A bitwise operation operates on each bit of the register individually. By carefully managing values the bitwise or may also be used as a logical or.

Instruction Format: R2, RI

Exceptions: none

SLL – Shift Left Logical

Description:

Shift left the value in Rs1 by the value in Rs2 or an immediate value and place the result in the destination register Rd. Low order bits are filled with zeros.

Instruction Format: R2, RI

Exceptions: none

SLT – Set if Less Than

Description:

Compare two two’s complement signed values which are in Rs1 and Rs2 or an immediate value. If Rs1 is less than the second operand then store a one in register Rd, otherwise store a zero in register Rd.

Instruction Format: R2, RI

Exceptions: none

SRA – Shift Right Arithmetic

Description:

Shift right the value in Rs1 by the value in Rs2 or an immediate value and place the result in the destination register Rd. High order bits are filled with the original sign bit, preserving the sign of the number.

Instruction Format: R2, RI

Exceptions: none

SRL – Shift Right Logical

Description:

Shift right the value in Rs1 by the value in Rs2 or an immediate value and place the result in the destination register Rd. High order bits are filled with zeros.

Instruction Format: R2, RI

Exceptions: none

SUB – Subtract

Description:

Subtract Rs2 or an immediate value from Rs1 and place the result in the destination register Rd. A bitwise operation operates on each bit of the register individually. By carefully managing values the bitwise xor may also be used as a logical xor.

Instruction Format: R2, RI

Exceptions: none

XOR – Bitwise Exclusive Or

Description:

Bitwise ‘exclusive or’ two values which are in Rs1 and Rs2 or an immediate value and place the result in the destination register Rd. A bitwise operation operates on each bit of the register individually. By carefully managing values the bitwise xor may also be used as a logical xor.

Instruction Format: R2, RI

Exceptions: none

Control Flow Instructions

BEQ – Branch if Equal

Description:

This instruction tests if two registers are equal and branches if they are otherwise program execution continues with the next instruction. The branch target is calculated as the sum of the program counter and a sign extended displacement value found in the instruction.

Instruction Format: BCC

Exceptions: none

BEQZ – Branch if Equal to Zero

Description:

This an alternate mnemonic for the BEQ instruction where the second register is assumed to be x0. This instruction tests if two registers are equal and branches if they are otherwise program execution continues with the next instruction. The branch target is calculated as the sum of the program counter and a sign extended displacement value found in the instruction.

Instruction Format: BCC

Exceptions: none

BGE – Branch if Greater Than or Equal

Description:

This instruction tests if two registers and branches if Rs1 is greater than or equal to Rs2; otherwise program execution continues with the next instruction. The values in registers Rs1 and Rs2 are treated as two’s complement signed numbers. The branch target is calculated as the sum of the program counter and a sign extended displacement value found in the instruction.

Instruction Format: BCC

Exceptions: none

BGEU – Branch if Greater Than or Equal Unsigned

Description:

This instruction tests if two registers and branches if Rs1 is greater than or equal to Rs2; otherwise program execution continues with the next instruction. The values in registers Rs1 and Rs2 are treated as unsigned numbers. The branch target is calculated as the sum of the program counter and a sign extended displacement value found in the instruction.

Instruction Format: BCC

Exceptions: none

BLE – Branch if Less Than or Equal

Description:

This is an alternate mnemonic for the BGE instruction. It's the same instruction except the order of the registers is reversed. This instruction tests if two registers and branches if Rs2 is less than or equal to Rs1; otherwise program execution continues with the next instruction. The values in registers Rs1 and Rs2 are treated as two's complement signed numbers. The branch target is calculated as the sum of the program counter and a sign extended displacement value found in the instruction.

Instruction Format: BCC

Exceptions: none

BLT – Branch if Less Than

Description:

This instruction tests if two registers and branches if Rs1 is less than Rs2; otherwise program execution continues with the next instruction. The values in registers Rs1 and Rs2 are treated as two's complement signed numbers. The branch target is calculated as the sum of the program counter and a sign extended displacement value found in the instruction.

Instruction Format: BCC

Exceptions: none

BLTU – Branch if Less Than Unsigned

Description:

This instruction tests if two registers and branches if Rs1 is less than Rs2; otherwise program execution continues with the next instruction. The values in registers Rs1 and Rs2 are treated as unsigned numbers. The branch target is calculated as the sum of the program counter and a sign extended displacement value found in the instruction.

Instruction Format: BCC

Exceptions: none

BNE – Branch if Not Equal

Description:

This instruction tests if two registers are unequal and branches if they are; otherwise program execution continues with the next instruction. The branch target is calculated as the sum of the program counter and a sign extended displacement value found in the instruction.

Instruction Format: BCC

Exceptions: none

BRA – Branch Always

Description:

This instruction is an alternate mnemonic for the BEQ instruction where both registers are assumed to be x0. Hence the branch is always taken. The branch target is calculated as the sum of the program counter and a sign extended displacement value found in the instruction.

Instruction Format: BRA

Exceptions: none

CALL – Call Subroutine

Description:

This is an alternate mnemonic for the JAL instruction where the destination register is assumed to be the $ra register.

Instruction Format: JAL, JALR

Exceptions: none

FENCE[.I]

Description:

With this core the fence instruction is a nop operation. Memory instructions are not buffered and always execute in order. Fencing is used to control order on machines where the order of memory operation may not be in program order.

JAL – Jump and Link

Description:

The JAL instruction jumps to the target address determined by adding a signed extended immediate constant in the instruction to the program counter. The constant is shifted left once before the addition. The two LSB's of the target address are set to zero. The address of the next instruction after the JAL is stored in the destination register Rd. The address range of the JAL instruction is approximately +/- 1 MB.

Instruction Format: JAL

Exceptions: none

JALR – Jump and Link Register

Description:

The JALR instruction jumps to the target address determined by adding a signed extended immediate constant in the instruction to integer register Rs1. The two LSB's of the target address are set to zero. The address of the next instruction after the JALR is stored in the destination register Rd. The address range is all of memory.

Instruction Format: JALR

Exceptions: none

RET – Return from Subroutine

Description:

RET is an alternate mnemonic for the JALR instruction where the constant is assumed to be zero and the source register is the return address register x1. The RET instruction is common to many instruction sets. Another mnemonic for this instruction is RTS.

Instruction Format: JALR

Exceptions: none

Memory Instructions

Address Modes

The processor supports only a single address mode – register indirect with displacement. Any other desired addressing of data must be built up out of instructions using this address mode.

Unaligned Accesses

If there is an unaligned access for data larger than a byte, the processor will automatically run two bus cycles to load or store the data. The processor doesn't care what address is used for the data; however, using aligned accesses results in faster program execution as only single bus cycles are required.

FLW – Float Load Word (32 bits)

Description:

FLW loads 32-bit data from memory and loads it into the floating-point destination register Rd. The memory address to load from is calculated as the sum of integer register Rs1 and an immediate constant in the instruction.

Instruction Format: ML

Exceptions: none

FSW – Float Store Word (32 bits)

Description:

FSW stores 32-bit data to memory from the floating-point destination source register Rs2. The memory address to store to is calculated as the sum of integer register Rs1 and an immediate constant in the instruction.

Instruction Format: MS

Exceptions: none

LB – Load Byte (8 bits)

Description:

LB loads a byte of data from memory, sign extends it to the width of the machine, and loads it into the integer destination register Rd. The memory address to load from is calculated as the sum of Rs1 and an immediate constant in the instruction.

Instruction Format: ML

Exceptions: none

LBU – Load Byte Unsigned (8 bits)

Description:

LBU loads a byte of data from memory, zero extends it to the width of the machine, and loads it into the integer destination register Rd. The memory address to load from is calculated as the sum of Rs1 and an immediate constant in the instruction.

Instruction Format: ML

Exceptions: none

LDB – Load Byte (8 bits)

Description:

LDB is an alternate mnemonic for the LB instruction. It loads a byte of data from memory, sign extends it to the width of the machine, and loads it into the integer destination register Rd. The memory address to load from is calculated as the sum of Rs1 and an immediate constant in the instruction.

Instruction Format: ML

Exceptions: none

LDBU – Load Byte Unsigned (8 bits)

Description:

LDBU is an alternate mnemonic for LBU. LDBU loads a byte of data from memory, zero extends it to the width of the machine, and loads it into the integer destination register Rd. The memory address to load from is calculated as the sum of Rs1 and an immediate constant in the instruction.

Instruction Format: ML

Exceptions: none

LDT – Load Tetra (32 bits)

Description:

LDT is an alternate mnemonic for the LW instruction which loads 32-bit data from memory and loads it into the integer destination register Rd. The memory address to load from is calculated as the sum of Rs1 and an immediate constant in the instruction.

Instruction Format: ML

Exceptions: none

LDW – Load Wyde (16 bits)

Description:

LDW is an alternate mnemonic for the LH instruction. LDW loads 16-bit data from memory, sign extends it to the width of the machine, and loads it into the integer destination register Rd. The memory address to load from is calculated as the sum of Rs1 and an immediate constant in the instruction.

Instruction Format: ML

Exceptions: none

LDWU – Load Wyde Unsigned (16 bits)

Description:

LDWU is an alternate mnemonic for LHU. LHU loads 16-bit data from memory, zero extends it to the width of the machine, and loads it into the integer destination register Rd. The memory address to load from is calculated as the sum of Rs1 and an immediate constant in the instruction.

Instruction Format: ML

Exceptions: none

LH – Load Half (16 bits)

Description:

LH loads 16-bit data from memory, sign extends it to the width of the machine, and loads it into the integer destination register Rd. The memory address to load from is calculated as the sum of Rs1 and an immediate constant in the instruction.

Instruction Format: ML

Exceptions: none

LHU – Load Half Unsigned (16 bits)

Description:

LHU loads 16-bit data from memory, zero extends it to the width of the machine, and loads it into the integer destination register Rd. The memory address to load from is calculated as the sum of Rs1 and an immediate constant in the instruction.

Instruction Format: ML

Exceptions: none

LW – Load Word (32 bits)

Description:

LW loads 32-bit data from memory and loads it into the integer destination register Rd. The memory address to load from is calculated as the sum of Rs1 and an immediate constant in the instruction.

Instruction Format: ML

Exceptions: none

SB – Store Byte (8 bits)

Description:

SB stores a byte of data to memory from the low order eight bits of source register Rs2. The memory address to load from is calculated as the sum of Rs1 and an immediate constant in the instruction.

Instruction Format: MS

Exceptions: none

SH – Store Half (16 bits)

Description:

SH stores 16-bits of data to memory from the low order sixteen bits of source register Rs2. The memory address to load from is calculated as the sum of Rs1 and an immediate constant in the instruction.

Instruction Format: MS

Exceptions: none

STB – Store Byte (8 bits)

Description:

STB is an alternate mnemonic of the SB instruction. SB stores a byte of data to memory from the low order eight bits of source register Rs2. The memory address to load from is calculated as the sum of Rs1 and an immediate constant in the instruction.

Instruction Format: MS

Exceptions: none

STT – Store Tetra (32 bits)

Description:

STT is an alternate mnemonic for the SW instruction. SW stores 32-bits of data to memory from source register Rs2. The memory address to load from is calculated as the sum of Rs1 and an immediate constant in the instruction.

Instruction Format: MS

Exceptions: none

STW – Store Wyde (16 bits)

Description:

STW is an alternate mnemonic of the SH instruction. SH stores 16-bits of data to memory from the low order sixteen bits of source register Rs2. The memory address to load from is calculated as the sum of Rs1 and an immediate constant in the instruction.

Instruction Format: MS

Exceptions: none

SW – Store Word (32 bits)

Description:

SW stores 32-bits of data to memory from source register Rs2. The memory address to load from is calculated as the sum of Rs1 and an immediate constant in the instruction.

Instruction Format: MS

Exceptions: none

Floating-Point

Rounding mode

The rounding mode to use for floating point instructions may be one specified in the instruction or a dynamic rounding mode specified in the rounding mode register. If the rounding mode specified in the instruction is '111' then the dynamic rounding mode register will be used to determine the rounding mode.

Floating-Point Exceptions

Underflow occurs when the result is a de-normal number having an exponent of zero. Underflow sets the uf bit in the floating-point status register.

Overflow occurs when the result becomes infinite (positive or negative); the exponents is all ones and the mantissa is zero. Overflow sets the of bit in the floating-point status register.

Inexact occurs during normalization if there were bits in the intermediate result that were non-zero to the right of the LSB of the result. Inexact sets the nx bit in the floating-point status register.

Divide by zero occurs if an attempt is made to divide a number by zero or an attempt is made to take the square root of zero. Divide by zero sets the dz bit in the floating-point status register.

Invalid operation occurs if there is an attempt to take the square root of a negative number. An invalid operation sets the nv bit in the floating-point status register.

FABS – Absolute Value

Description:

FABS is an alternate mnemonic for FSGNJX which copies the value in Rs1 into the destination register Rd then sets the sign of Rd equal to the xor of the sign of Rs1 and Rs2. Rs1 and Rs2 are encoded as the same register by the assembler.

Instruction Format: FSGNJ

Exceptions: none

FADD – Addition

Description:

FADD adds two floating-point values in floating-point registers Rs1 and Rs2 and store the result in floating-point register Rd. If either operand is a Nan then the result is a Nan.

Instruction Format: FLT

Exceptions: uf, of, nx

FDIV – Division

Description:

FDIV divides two floating-point values in floating-point registers Rs1 and Rs2 and stores the result in floating-point register Rd. If either operand is a Nan then the result is a Nan.

Instruction Format: FLT

Exceptions: uf, of, nx, dz

FEQ – Float Test for Equality

Description:

This instruction tests two floating-point values in registers Rs1 and Rs2 for equality. If the condition is true a one is returned in integer register Rd. Rs1 and Rs2 are floating-point registers. Positive zero and negative zero are assumed to be equal. If either operand is a Nan, then this test will return false.

Instruction Format: FLT

Exceptions: none

FLE – Float Test for Less Than or Equal

Description:

This instruction tests two floating-point values in registers Rs1 and Rs2 for Rs1 less than or equal to Rs2. If Rs1 is less than or equal to Rs2 then Rd is set to one. Otherwise Rd is set to zero. Rd is an integer register, Rs1 and Rs2 are floating-point registers. Positive zero and negative zero are assumed to be equal. If either operand is a Nan, then this test will return false. This instruction may also be used to test for greater than or equal by swapping the operands.

Instruction Format: FLT

Exceptions: none

FLT – Float Test for Less Than

Description:

This instruction tests two floating-point values in registers Rs1 and Rs2 for Rs1 less than Rs2. If Rs1 is less than Rs2 then Rd is set to one. Otherwise Rd is set to zero. Rd is an integer register, Rs1 and Rs2 are floating-point registers. Positive zero and negative zero are assumed to be equal. If either operand is a Nan, then this test will return false. This instruction may also be used to test for greater than by swapping the operands.

Instruction Format: FLT

Exceptions: none

FMOV – Move Register

Description:

FMOV is an alternate mnemonic for FSGNJ which copies the value in Rs1 into the destination register Rd then sets the sign of Rd equal to the sign of Rs2. Rs1 and Rs2 are encoded as the same register by the assembler. Both the source and destination registers are part of the floating-point register file. To move directly between the integer and floating-point register files see the FMV instruction.

Instruction Format: FSGNJ

Exceptions: none

FMUL – Multiplication

Description:

FMUL multiplies two floating-point values in floating-point registers Rs1 and Rs2 and stores the result in floating-point register Rd. If either operand is a Nan then the result is a Nan.

Instruction Format: FLT

Exceptions: uf, of, nx

FMV – Move Register

Description:

The FMV instruction moves a value directly between integer and floating-point registers without performing any conversions. FMV.X.S moves from a floating-point register Rs1 to an integer register Rd. FMV.S.X moves an integer register Rs1 to a floating-point register Rd.

Instruction Format: FMV

Exceptions: none

FNEG – Negate

Description:

FNEG is an alternate mnemonic for FSGNJN which copies the value in Rs1 into the destination register Rd then sets the sign of Rd equal to the complement of the sign of Rs2. Rs1 and Rs2 are encoded as the same register by the assembler.

Instruction Format: FSGNJ

Exceptions: none

FSGNJ – Sign Injection

Description:

FSGNJ copies the value in Rs1 into the destination register Rd then sets the sign of Rd equal to the sign of Rs2.

Instruction Format: FSGNJ

Exceptions: none

FSGNJN – Sign Injection Invert

Description:

FSGNJ copies the value in Rs1 into the destination register Rd then sets the sign of Rd equal to the complement of the sign of Rs2.

Instruction Format: FSGNJ

Exceptions: none

FSGNJX – Sign Injection Xor

Description:

FSGNJX copies the value in Rs1 into the destination register Rd then sets the sign of Rd equal to the xor of the sign of Rs1 and Rs2.

Instruction Format: FSGNJ

Exceptions: none

FSUB – Subtraction

Description:

FSUB subtracts two floating-point values in floating-point registers Rs1 and Rs2 and stores the result in floating-point register Rd. If either operand is a Nan then the result is a Nan.

Instruction Format: FLT

Exceptions: uf, of, nx

Machine Mode Instructions

EBREAK – Debug Environment Call

Description:

This instruction transfers control back to the debug environment. The processor is switched to machine mode with interrupts disabled. The machine mode register set is selected. An ERET instruction should be used to return from an environment call.

ECALL – Environment Call

Description:

This instruction invokes environment (operating system) processing. The processor is switched to machine mode with interrupts disabled. The machine mode register set is selected. An ERET instruction should be used to return from an environment call.

ERET – Return from Exception

Description:

This instruction returns to user mode from an exception handler. The previous interrupt mask setting is restored. The previous register set is also restored.

MRET – Return from Machine Exception

Description:

This instruction returns to user mode from an exception handler. The previous interrupt mask setting is restored. The previous register set is also restored.

WFI – Wait for Interrupt

Description:

This instruction causes the processor to pause and wait for an interrupt signal before continuing. While waiting for an interrupt the processor clock is stopped to reduce power consumption. Only the wall-clock time is updated. If an interrupt occurs and interrupts are enabled, then the interrupt service routine will begin. Otherwise if an interrupt occurs and interrupts are not enabled, then program execution will continue with the next instruction.

Instruction Format: WFI

Exceptions: none

Custom Instructions

The following instructions are not part of the RISCV standard.

DECTO – Decrement Timeout

Description:

Decrements the timeout value for an array of 32 timeouts. All 32 timeouts are decremented. This instruction requires over 32 clock cycles to perform the decrement, however the instruction operates asynchronously, and effectively is a single cycle operation. If the instruction is repeated before the previous cycle completed the second operation will be ignored.

Green-Field Extension

This instruction is a green-field extension to the base RISCV instruction set and not likely to be present in other implementations.

Instruction Format: R2

Exceptions: none

GCSUB – Garbage Collect Subtract

Description:

Subtract Rs2 or an immediate value from Rs1 and place the result in the destination register Rd. Also clear the garbage collect interrupt enable bit in the user interrupt enable CSR (CSR $004) and load a lockout count into an internal instruction count register. Once the lockout count has expired the interrupt enable bit will be set enabling GC interrupts. The value loaded into the lockout count is four plus the value in Rs2 or the immediate value shift right twice.

Green-Field Extension

This instruction is a custom instruction not part of the RISCV standard.

Instruction Format: R2, RI

Exceptions: none

MAJ – Majority Logic

Description:

This instruction determines the majority in a bitwise fashion for three input values.

Operation:

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

Green-Field Extension

This instruction is a green-field extension to the base RISCV instruction set and not likely to be present in other implementations.

Instruction Format: MAJ

Exceptions: none

MVMAP – Move Mapping Register

Description:

MVMAP instruction is used for mapping memory pages into the address space of a task.

MVMAP works in a manner similar to the CSR instruction, but is applied for mapping register access only. Register Rs2 indirectly identifies the map register to access. Note that Rs2 is an integer register that contains the map register number. Rs1 identifies new source data for the map register, and Rd specifies the register to put the current map register value into. New source data and the current data in the map register are swapped in an atomic fashion.

Specifying Rs1 as x0 causes the map move operation to only output the current map value without updating it.

The Rs2 field specifies a 32-bit value broken into two fields. The low order nine bits are a map register number for a given task. Bits 16 to 19 specify the task number for which the map is updated. The mapping register is only nine bits wide. Upper bits from the source register are ignored.

Green-Field Extension

This instruction is a custom instruction not part of the RISCV standard.

Instruction Format: MTU

Exceptions: none

MVSEG – Move Segment Register

Description:

MVSEG works in a manner similar to the CSR instruction, but is applied for segment register access only. Register Rs2 indirectly identifies the segment register to access. Note that Rs2 is an integer register that contains the segment register number. Rs1 identifies source data for the segment register, and Rd specifies the register to put the current segment register value into. New source data and the current data in the segment register are swapped in an atomic fashion.

Green-Field Extension

This instruction is a custom instruction not part of the RISCV standard.

Instruction Format: MVSEG

Exceptions: none

PALLOC – PAM Allocate

Description:

This instruction accesses an internal page allocation map (PAM) and will find a bit set to zero and set it to a one, returning the bit number in Rd. The search takes place a configuration defined number of bits in parallel (typically 32). The palloc function "remembers" the last position of the bit and begins it's search at the last position. This improves performance. The size of the PAM is a configuration defined number of bits. There should be a bit for each possible memory page in the system.

Green-Field Extension

This instruction is a green-field extension to the base RISCV instruction set and not likely to be present in other implementations.

Instruction Format: PALLOC

Exceptions: none

PEEKQ – Peek at Queue

Description:

This instruction returns the top value into Rd from the hardware queue specified in Rs1. The hardware queue position is not advanced. The value returned in Rd includes status bits in addition to the value pushed. The value field is an N-bit field between 1 and 20 bits in size. Unused value bits should read as zero.

31	30	29 26	25 20	19 0
Qe	Dv	~	DC	Value

Fields

Dv: data valid. If this bit is set it indicates that the N-bit value field is a valid queue data.

Qe: queue empty.If set, this bit indicates that the queue is empty.

Dc: data count: The number of items left in the queue

Value: the value that was pushed to the queue

Green-Field Extension

This instruction is a green-field extension to the base RISCV instruction set and not likely to be present in other implementations.

Instruction Format: PUSHQ

Exceptions: none

PFREE – PAM Free

Description:

This instruction accesses an internal page allocation map (PAM) and will set the bit specified in Rs1 to zero. This bit would be a value returned by the PALLOC instruction.

Green-Field Extension

This instruction is a green-field extension to the base RISCV instruction set and not likely to be present in other implementations.

Instruction Format: PFREE

Exceptions: none

PSTAT – PAM Status Get / Set

Description:

This instruction accesses an internal page allocation map (PAM) and will get the bit specified in Rs1. The specified bit may be set to zero or one, or not set according to the value in Rs2. A value in Rs2 of zero sets the bit to zero, a value of one sets the bit to one, a value of two ignores the value in Rs2.

Green-Field Extension

This instruction is a green-field extension to the base RISCV instruction set and not likely to be present in other implementations.

Instruction Format: R2

Exceptions: none

PFI – Poll for Interrupt

Description:

This instruction causes the processor to check for the presence of an interrupt then perform interrupt processing if an interrupt is present. Otherwise program execution continues with the next instruction. Interrupts do no have to be enabled for the PFI instruction to perform interrupt processing. Effectively PFI temporarily enables interrupts for the duration of the instruction.

Green-Field Extension

This instruction is a green-field extension to the base RISCV instruction set and not likely to be present in other implementations. An equivalent action may be performed using a minimum sequence of two CSR instructions to enable then disable interrupts.

Instruction Format: PFI

Exceptions: none

POPQ – Pop from Queue

Description:

This instruction pops a value into Rd from the hardware queue specified in Rs1. The hardware queue position is advanced. The value returned in Rd includes status bits in addition to the value pushed. The value field is an N-bit field between 1 and 20 bits in size. Unused value bits should read as zero.

31	30	29 26	25 20	19 0
Qe	Dv	~	DC	Value

Fields

Dv: data valid. If this bit is set it indicates that the N-bit value field is a valid queue data.

Qe: queue empty.If set, this bit indicates that the queue is empty.

Dc: data count: The number of items left in the queue

Value: the value that was pushed to the queue

Green-Field Extension

This instruction is a green-field extension to the base RISCV instruction set and not likely to be present in other implementations.

Instruction Format: PUSHQ

Exceptions: none

PUSHQ – Push on Queue

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 20 bits.

Green-Field Extension

This instruction is a green-field extension to the base RISCV instruction set and not likely to be present in other implementations.

Instruction Format: PUSHQ

Exceptions: none

SETTO – Set Timeout

Description:

This instruction updates the timeout counter specified in Rs1 with the value in Rs2. There is an array of 32 timers which are all decremented by the DECTO instruction.

Green-Field Extension

This instruction is a green-field extension to the base RISCV instruction set and not likely to be present in other implementations.

Instruction Format: R2

Exceptions: none

Machine Mode Programming Model

Machine mode has its own integer register file.

	Registers	
	31 0	
	x0 / zero	
	x1 / ra	
	x2 / sp	
	x3 / gp	
	x4 / tp	
	x5 / t0 / alternate link	
	x6-x7 / t1-t2	
	x8 / fp / s0	
	x9 / s1	
	x10-x11 / a0-a1	
	x12-x17 / a2-a7	
	x18-x27 / s2 – s11	
	x28-x31 / t3-t6	
	pc	

x? refers to an integer register.

Register x0 is always zero.

x14 / sp is the stack pointer.

pc is the program counter.

Register Set Usage

Set No.	ABI Usage
0 to 27	Applications
28	Reset
29	Ecall
30	Interrupts
31	Debug mode

Register Set Selection

Which register file (Machine mode or user mode) is used for each of Rs1, Rs2, Rs3, and Rd in an instruction is controlled by the register set (regset) CSR. If the bit in the regset CSR is clear then the machine register is selected, otherwise the previous register set is selected.

CSR $7C0 - Regset

31 4	3	2	1	0
reserved	Rs3	Rs2	Rs1	Rd

Updating the regset CSR returns the current value of the CSR in the newly selected Rd register set.

CSR $7C3 – Regset selector stack

The register set selector stack keeps track of the register set selector currently active and previously active in the form of an six-entry stack. An exception will shift the register to the left by five bits and insert the current register set selector into the low order five bits. An exception return instruction will shift the register to the right by five bits.

29 10	9 5	4 0
<more stacked rs>	prs	crs

Reset Operation

The RISCV spec pretty much leaves it up to the implementor to set the reset address. There are generally two used areas for the reset address, a high address or a low address. Ram memory often begins at a low address, so the author chose a high address for the reset address. A small rom is placed at $FFFC0000 in the upper range of addresses. Often the rom contains just enough code to load an OS into memory from a I/O device such as disk, or memory card.

On reset the processor begins executing instructions at $FFFC0100 in machine mode. Interrupts are disabled. All other state is undefined.

Interrupt Programming Model

The interrupt programming model allows for fast (low latency) interrupt handling. A dedicated integer register file is available for interrupt processing. This register file is selected automatically when a hardware interrupt occurs. This eliminates the need to save and restore integer registers during interrupt handling.

Hardware interrupts have their own integer register file. This register file is automatically selected when a hardware interrupt occurs. On return from interrupt by executing the MRET instruction the register set prior to the interrupt is selected.

	Registers	
	31 0	
	x0 / zero	
	x1 / ra	
	x2 / sp	
	x3 / gp	
	x4 / tp	
	x5 / t0 / alternate link	
	x6-x7 / t1-t2	
	x8 / fp / s0	
	x9 / s1	
	x10-x11 / a0-a1	
	x12-x17 / a2-a7	
	x18-x27 / s2 – s11	
	x28-x31 / t3-t6	
	pc	

x? refers to an integer register.

Register x0 is always zero.

x14 / sp is the stack pointer.

pc is the program counter.

Simplified System MMU (SSMMU)

Introduction

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

The simplified system MMU provides minimalistic base and bound and paging capabilities for a small to mid size system. Base bound and paging are applied only to user mode apps. In machine mode the system sees a flat address space with no restrictions on access. Base address generation is applied to virtual addresses first to generate a linear address which is then mapped using a paged mapping system. Access rights are governed by the base register since all pages in the based on the same address are likely to require the same access. Support for access rights is optional if it's desired to reduce the hardware cost. To simplify hardware there are no bound registers. Bounds are determined by what memory is mapped into the base address area.

Base Registers

The upper address bits of a virtual or effective address are not used for addressing memory and are available to select base register. The SSMMU includes 16 base registers. The base register in use is selected by the upper nybble of the virtual address. In the case of the program address, program counter bits 30 and 31 are used to select one of four registers. If the program address has all ones in bits 24 to 31 then base addressing is bypassed. This provides a shared program area containing the BIOS and OS code.

Base Regno	Usage	Selected By
0 to 7	data	bits 28 to 31 of effective address
8, 9	reserved	bits 28 to 31 of effective address
10	Stack	bits 28 to 31 of effective address
11	I/O	bits 28 to 31 of effective address
12 to 15	code	bits 30, 31 of pc

Base Register Format

31 … 4	3 … 0
Base Address$_{28}$	RWX

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

R: 1 = segment readable

W: 1 = segment writeable

X: 1 = segment executable

Linear Address Generation

The base address value contained in the upper 28 bits of a base register is shifted left 10 bits before being added to the virtual address. This gives potentially a 38-bit address space.

Note there is no limit field. Access is limited by what is mapped into the segment.

The Page Map

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

In addition to based addresses, memory is divided up into 1kB pages which are mapped. There are 32 memory maps available. A memory map represents an address space; a five-bit address space identifier is in use. Address spaces will need to be shared if more than 32 apps are running in the system. The desire is to keep the mapping tables small so they may fit into a small number of standard memory blocks. For instance, for the sample system there are 512 pages required to map the 512kB address space. Any individual app is limited to maximum of 256kB (one half of the memory available). The virtual page number is used to lookup the physical page in the page mapping table. Addresses with the top eight bits set are not mapped to allow access to the system ROM.

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

ASID_4	Virtual Page	Physical Page
0	0	10
	1	11
	…	
	254	18
	255	19
1	0	
	1	
	…	
	254	
	255	
… 30 more address spaces		

The low order 10 bits of an address pass through both linear address generation and paging unchanged.

The 1kB Page

Many memory systems use a 4kB page size. That size was not chosen here as the available memory is assumed to be small and a 4kB page size would result in too few pages (128) of memory to support multiple tasks. A smaller page size results in less wasted space which is important with a small memory system. It’s a careful balance, an even smaller page size would waste less memory but would require a much larger page mapping ram.

MVMAP

Rs1:

31 20	20 16	15 0
Unused - should be zero	$ASID_5$	Virtual page number 16 bits max

Exercises

Write a program to load two numbers into x1 and x2 and store the sum in x3.

Write a program to compute the rom checksum. The rom checksum is the sum of all the bytes in the rom.

Compute the clocks per instruction using the tick count and instructions retired CSR registers. Compute the result using floating point instructions.

Where does the processor go when system mode is entered?

The processor continues on from where it was last before entering user mode. Since user mode is entered with an eret instruction, the address is the next address after the eret. However, at reset the processor begins running in system mode at the reset address of $FFFC0100.