ECS 50 RISC-V and X86-64 Reference

Nicholas Weaver, based on work by James Zhu <jameszhu@berkeley.edu>

RISC-V Instruction Set

Core Instruction Formats

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Integer Instructions

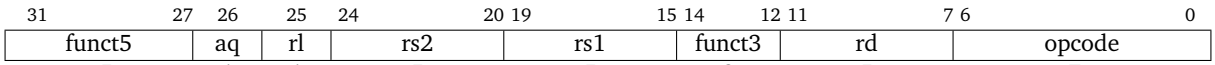
Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
`add`	ADD	R	`0110011`	`0x0`	`0x00`	`rd = rs1 + rs2`	
`sub`	SUB	R	`0110011`	`0x0`	`0x20`	`rd = rs1 - rs2`	
`xor`	XOR	R	`0110011`	`0x4`	`0x00`	`rd = rs1 ^ rs2`	
`or`	OR	R	`0110011`	`0x6`	`0x00`	`rd = rs1 | rs2`	
`and`	AND	R	`0110011`	`0x7`	`0x00`	`rd = rs1 & rs2`	
`sll`	Shift Left Logical	R	`0110011`	`0x1`	`0x00`	`rd = rs1 << rs2`	
`srl`	Shift Right Logical	R	`0110011`	`0x5`	`0x00`	`rd = rs1 >> rs2`	
`sra`	Shift Right Arith*	R	`0110011`	`0x5`	`0x20`	`rd = rs1 >> rs2`	msb-extends
`slt`	Set Less Than	R	`0110011`	`0x2`	`0x00`	`rd = (rs1 < rs2)?1:0`	
`sltu`	Set Less Than (U)	R	`0110011`	`0x3`	`0x00`	`rd = (rs1 < rs2)?1:0`	zero-extends
`addi`	ADD Immediate	I	`0010011`	`0x0`		`rd = rs1 + imm`	
`xori`	XOR Immediate	I	`0010011`	`0x4`		`rd = rs1 ^ imm`	
`ori`	OR Immediate	I	`0010011`	`0x6`		`rd = rs1 | imm`	
`andi`	AND Immediate	I	`0010011`	`0x7`		`rd = rs1 & imm`	
`slli`	Shift Left Logical Imm	I	`0010011`	`0x1`	`imm[5:11]=0x00`	`rd = rs1 << imm[0:4]`	
`srli`	Shift Right Logical Imm	I	`0010011`	`0x5`	`imm[5:11]=0x00`	`rd = rs1 >> imm[0:4]`	
`srai`	Shift Right Arith Imm	I	`0010011`	`0x5`	`imm[5:11]=0x20`	`rd = rs1 >> imm[0:4]`	msb-extends
`slti`	Set Less Than Imm	I	`0010011`	`0x2`		`rd = (rs1 < imm)?1:0`	
`sltiu`	Set Less Than Imm (U)	I	`0010011`	`0x3`		`rd = (rs1 < imm)?1:0`	zero-extends
`lb`	Load Byte	I	`0000011`	`0x0`		`rd = M[rs1+imm][0:7]`	
`lh`	Load Half	I	`0000011`	`0x1`		`rd = M[rs1+imm][0:15]`	
`lw`	Load Word	I	`0000011`	`0x2`		`rd = M[rs1+imm][0:31]`	
`lbu`	Load Byte (U)	I	`0000011`	`0x4`		`rd = M[rs1+imm][0:7]`	zero-extends
`lhu`	Load Half (U)	I	`0000011`	`0x5`		`rd = M[rs1+imm][0:15]`	zero-extends
`sb`	Store Byte	S	`0100011`	`0x0`		`M[rs1+imm][0:7] = rs2[0:7]`	
`sh`	Store Half	S	`0100011`	`0x1`		`M[rs1+imm][0:15] = rs2[0:15]`	
`sw`	Store Word	S	`0100011`	`0x2`		`M[rs1+imm][0:31] = rs2[0:31]`	
`beq`	Branch ==	B	`1100011`	`0x0`		`if(rs1 == rs2) PC += imm`	
`bne`	Branch !=	B	`1100011`	`0x1`		`if(rs1 != rs2) PC += imm`	
`blt`	Branch <	B	`1100011`	`0x4`		`if(rs1 <  rs2) PC += imm`	
`bge`	Branch ≥	B	`1100011`	`0x5`		`if(rs1 >= rs2) PC += imm`	
`bltu`	Branch < (U)	B	`1100011`	`0x6`		`if(rs1 <  rs2) PC += imm`	zero-extends
`bgeu`	Branch ≥ (U)	B	`1100011`	`0x7`		`if(rs1 >= rs2) PC += imm`	zero-extends
`jal`	Jump And Link	J	`1101111`			`rd = PC+4; PC += imm`	
`jalr`	Jump And Link Reg	I	`1100111`	`0x0`		`rd = PC+4; PC = rs1 + imm`	
`lui`	Load Upper Imm	U	`0110111`			`rd = imm << 12`	
`auipc`	Add Upper Imm to PC	U	`0010111`			`rd = PC + (imm << 12)`	
`ecall`	Environment Call	I	`1110011`	`0x0`	`imm=0x0`	`Transfer control to OS`	
`ebreak`	Environment Break	I	`1110011`	`0x0`	`imm=0x1`	`Transfer control to debugger`	

Standard Extensions

RV32M Multiply Extension

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
mul	MUL	R	0110011	0x0	0x01	rd = (rs1 * rs2)[31:0]
mulh	MUL High	R	0110011	0x1	0x01	rd = (rs1 * rs2)[63:32]
mulsu	MUL High (S) (U)	R	0110011	0x2	0x01	rd = (rs1 * rs2)[63:32]
mulu	MUL High (U)	R	0110011	0x3	0x01	rd = (rs1 * rs2)[63:32]
div	DIV	R	0110011	0x4	0x01	rd = rs1 / rs2
divu	DIV (U)	R	0110011	0x5	0x01	rd = rs1 / rs2
rem	Remainder	R	0110011	0x6	0x01	rd = rs1 % rs2
remu	Remainder (U)	R	0110011	0x7	0x01	rd = rs1 % rs2

RV32A Atomic Extension



Inst	Name	FMT	Opcode	funct3	funct5	Description (C)
lr.w	Load Reserved	R	0101111	0x2	0x02	rd = M[rs1], reserve M[rs1]
sc.w	Store Conditional	R	0101111	0x2	0x03	if (reserved) { M[rs1] = rs2; rd = 0 } else { rd = 1 }
amoswap.w	Atomic Swap	R	0101111	0x2	0x01	rd = M[rs1]; swap(rd, rs2); M[rs1] = rd
amoadd.w	Atomic ADD	R	0101111	0x2	0x00	rd = M[rs1] + rs2; M[rs1] = rd
amoand.w	Atomic AND	R	0101111	0x2	0x0C	rd = M[rs1] & rs2; M[rs1] = rd
amoor.w	Atomic OR	R	0101111	0x2	0x0A	rd = M[rs1] | rs2; M[rs1] = rd
amoxor.w	Atomix XOR	R	0101111	0x2	0x04	rd = M[rs1] ^ rs2; M[rs1] = rd
amomax.w	Atomic MAX	R	0101111	0x2	0x14	rd = max(M[rs1], rs2); M[rs1] = rd
amomin.w	Atomic MIN	R	0101111	0x2	0x10	rd = min(M[rs1], rs2); M[rs1] = rd

RV32F / D Floating-Point Extensions

Inst	Name	FMT	Opcode	funct3	funct5	Description (C)
flw	Flt Load Word	*				rd = M[rs1 + imm]
fsw	Flt Store Word	*				M[rs1 + imm] = rs2
fmadd.s	Flt Fused Mul-Add	*				rd = rs1 * rs2 + rs3
fmsub.s	Flt Fused Mul-Sub	*				rd = rs1 * rs2 - rs3
fnmadd.s	Flt Neg Fused Mul-Add	*				rd = -rs1 * rs2 + rs3
fnmsub.s	Flt Neg Fused Mul-Sub	*				rd = -rs1 * rs2 - rs3
fadd.s	Flt Add	*				rd = rs1 + rs2
fsub.s	Flt Sub	*				rd = rs1 - rs2
fmul.s	Flt Mul	*				rd = rs1 * rs2
fdiv.s	Flt Div	*				rd = rs1 / rs2
fsqrt.s	Flt Square Root	*				rd = sqrt(rs1)
fsgnj.s	Flt Sign Injection	*				rd = abs(rs1) * sgn(rs2)
fsgnjn.s	Flt Sign Neg Injection	*				rd = abs(rs1) * -sgn(rs2)
fsgnjx.s	Flt Sign Xor Injection	*				rd = rs1 * sgn(rs2)
fmin.s	Flt Minimum	*				rd = min(rs1, rs2)
fmax.s	Flt Maximum	*				rd = max(rs1, rs2)
fcvt.s.w	Flt Conv from Sign Int	*				rd = (float) rs1
fcvt.s.wu	Flt Conv from Uns Int	*				rd = (float) rs1
fcvt.w.s	Flt Convert to Int	*				rd = (int32_t) rs1
fcvt.wu.s	Flt Convert to Int	*				rd = (uint32_t) rs1
fmv.x.w	Move Float to Int	*				rd = *((int*) &rs1)
fmv.w.x	Move Int to Float	*				rd = *((float*) &rs1)
feq.s	Float Equality	*				rd = (rs1 == rs2) ? 1 : 0
flt.s	Float Less Than	*				rd = (rs1 < rs2) ? 1 : 0
fle.s	Float Less / Equal	*				rd = (rs1 <= rs2) ? 1 : 0
fclass.s	Float Classify	*				rd = 0..9

RV32C Compressed Extension

15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0	
funct4		rd/rs1		rs2		op	CR-type
funct3	imm	rd/rs1		imm		op	CI-type
funct3	imm			rs2		op	CSS-type
funct3	imm				rd’	op	CIW-type
funct3	imm	rs1’		imm	rd’	op	CL-type
funct3	imm	rd’/rs1’		imm	rs2’	op	CS-type
funct3	imm	rs1’		imm		op	CB-type
funct3	offset					op	CJ-type

Inst	Name	FMT	OP	Funct	Description
c.lwsp	Load Word from SP	CI	10	010	lw rd, (4*imm)(sp)
c.swsp	Store Word to SP	CSS	10	110	sw rs2, (4*imm)(sp)
c.lw	Load Word	CL	00	010	lw rd’, (4*imm)(rs1’)
c.sw	Store Word	CS	00	110	sw rs1’, (4*imm)(rs2’)
c.j	Jump	CJ	01	101	jal x0, 2*offset
c.jal	Jump And Link	CJ	01	001	jal ra, 2*offset
c.jr	Jump Reg	CR	10	1000	jalr x0, rs1, 0
c.jalr	Jump And Link Reg	CR	10	1001	jalr ra, rs1, 0
c.beqz	Branch == 0	CB	01	110	beq rs’, x0, 2*imm
c.bnez	Branch != 0	CB	01	111	bne rs’, x0, 2*imm
c.li	Load Immediate	CI	01	010	addi rd, x0, imm
c.lui	Load Upper Imm	CI	01	011	lui rd, imm
c.addi	ADD Immediate	CI	01	000	addi rd, rd, imm
c.addi16sp	ADD Imm * 16 to SP	CI	01	011	addi sp, sp, 16*imm
c.addi4spn	ADD Imm * 4 + SP	CIW	00	000	addi rd’, sp, 4*imm
c.slli	Shift Left Logical Imm	CI	10	000	slli rd, rd, imm
c.srli	Shift Right Logical Imm	CB	01	100x00	srli rd’, rd’, imm
c.srai	Shift Right Arith Imm	CB	01	100x01	srai rd’, rd’, imm
c.andi	AND Imm	CB	01	100x10	andi rd’, rd’, imm
c.mv	MoVe	CR	10	1000	add rd, x0, rs2
c.add	ADD	CR	10	1001	add rd, rd, rs2
c.and	AND	CS	01	10001111	and rd’, rd’, rs2’
c.or	OR	CS	01	10001110	or rd’, rd’, rs2’
c.xor	XOR	CS	01	10001101	xor rd’, rd’, rs2’
c.sub	SUB	CS	01	10001100	sub rd’, rd’, rs2’
c.nop	No OPeration	CI	01	000	addi x0, x0, 0
c.ebreak	Environment BREAK	CR	10	1001	ebreak

Control and Status Registers

Instruction	Operation	Notes
`csrrw rd csr rs`	`csr = rs; rd = csr`	`rd=x0` only does the write
`csrrs rd csr rs`	`csr = rs | csr; rd = csr`	`rs=x0` only does the read
`csrrc rd csr rs`	`csr = r̂s & csr; rd = csr`	`rs=x0` only does the read
`csrrwi rd csr imm`	`csr = imm, rd = csr`	`rd=x0` only writes. Immediate is 5 bit, zero extended
`csrrsi rd csr imm`	`csr = imm | csr, rd = csr`	`imm=0` only reads. Immediate is 5 bit, zero extended
`csrrci rd csr imm`	`csr = îmm & csr, rd = csr`	`imm=0` only reads. Immediate is 5 bit, zero extended

Pseudo-instructions `csrw` and `csrr` are converted to `csrrw` and `csrrs` respectively. All CSR instructions are I-type with the immediate specifying the CSR. The immediate CSRs use the source register as a 5-bit, zero-extended immediate. Traps are returned from using `mret`.

The listing of CSRs is deliberately incomplete but it is designed for both a ECS 50 (assembly) class and an ECS 150 (operating systems) type class as a quick reference.

CSR	Meaning and Access
`misa`	Machine ISA: Allows checking what extensions are supported and disabling them.
`mvendorid`	Manufacturer ID number
`marchid`	Architeture ID number
`mhartid`	Hart (HARdware Thread) number
`mstatus`	The lower 32b of machine status.
`mstatush`	The upper 32b of machine status.
`mtvec`	Address of the trap handler. If the last bit is set the interrupt portion is vectored.
`medeleg` and `mideleg`	Delegation of interrupts
`mip` and `mie`	Pending interrupts and enabled interrupts
`mepc`	Machine Exception Program Counter. The PC that triggered a trap.
`mcause`	Machine Cause. The cause of the current trap/interrupt.
`mtval`	Additional value information. For memory faults, the address fetched that triggered the fault.
`pmpcfg0-pmpcfg15`	Physical Memory Protection configuration
`pmpaddr0-pmpaddr63`	Physical Memory Protection Addresses
`sapt`	System Page Table pointer

Comments on particular CSRs

`mhartid`: RISC-V specifies a "Hart" as a Hardware thread of execution. In the absence of symmetric multithreading, this is the CPU core number. If there is hardware multithreading it specifies the distinct hardware thread where multiple threads can share a single CPU core.

`mstatus` and `mstatush`: These CSRs (`mstatush` only exists on RV32, RV64 just has a combined `mstatus`) state both the current state of the machine in terms of operating mode and other operating data. This includes whether interrupts are enabled, what mode the system is currently in, and what mode the system will be returned to with `mret` or `sret` when returning from the trap handler. This CSR is very complicated with lots of details for those implementing operating systems or similar low level code.

`mtvec`: This is the address of the trap handler. When an interrupt or trap occurs the CPU will update the operating mode and jump to the trap handler code, disabling interrupts so the trap handler won't be interrupted during the process. If the last two bits are `01b` the interrupts are vectored: jumping to `mtvec + 4*`*Interrupt number*.

`medeleg` and `mideleg`: RISC-V supports up to 32 distinct traps and interrupts. This register allows machine mode (M-mode) to delegate specific traps and interrupts to the S-mode, so that those traps are instead handled by the S-mode trap handler. Only traps that occur in S-mode or U-mode are delegated, any trap or interrupt that occurs while in M-mode will never be delegated to the S-mode trap handler.

`mscratch`: This register generally holds a pointer to space for the trap handler. The common convention for the trap handler is to perform an atomic swap with `sp`, giving the trap handler its own stack space. The trap handler can then save all registers. Once this is complete the trap handler can now operate normally. When exiting the trap handler the trap handler should restore all the registers, restore `sp` by again swapping it with `mscratch`, and then execute `mret`.

`mepc`: The address of the instruction triggering a trap. For an exception this is the instruction which triggered the exception (and is not yet executed) and for an interrupt it is the first instruction not yet executed. If the trap handler wants to retry the instruction (e.g. because it was an interrupt or because it was a corrected memory protection error), `mret` will just return to the address in `mepc`. If the trap handler instead corrected the exception (e.g. a misaligned load), it should first increment `mepc` to the next instruction before returning.

`mcause`: The particular trap/interrupt number that triggered the exception. Interrupts have the MSB set (allowing an easy signed comparison with 0).

`mtval`: The value depends on the particular trap or interrupt. For memory related faults (e.g. misaligned load, page fault), it always contains the virtual address of the memory who's access triggered the fault.

`pmpcfg*` and `pmpaddr*`: These CSRs are to implement a low level Physical Memory Protection scheme, allowing M-mode to control what memory is accessed in S-Mode or U-Mode. Its primary usage is on small embedded processor that don't run a full virtual-memory supporting operating system but still want some level of protection.

The config registers use 8 bits to indicate the status of each corresponding address register. These include whether S/U mode can Read data, Write data, eXecute data, the Address mode for the address register, and whether to Lock out control.

The two primary address modes are to specify a Naturally Aligned Power of 2 (NAPOT) in the corresponding address register or whether two addresses should be taken together to specify a valid range.

`satp`: This is the pointer to the page table. For efficiency reasons it points to a physical page rather than a specific memory location, because page table entries are aligned on page-boundaries. It only exists if the processor supports S mode, and it can be applied for both S-mode and U-mode accesses.

The page table itself is a 2-level radix structure: Pages in RISC-V are 4 kB which means the page offset a 12b value and the virtual address is 20b. Since pages are 4 kB, and page table entries are 4B, this means that we can fit 1024 page table entries in a physical page.

The SATP pointer points to the root physical page: The upper 10 bits of the virtual address are used to look up the individual entry. If the entry is invalid it triggers a trap. If it is valid the resulting entry specifies another page to evaluate the next 10 bits of the vitrual address. Once the proper page table entry is found the physical address then replaces the virtual address before the actual memory fetch occurs.

One particular note is that the S and U modes can use the same page table. In that case the system's pages are usually marked as both "global" for efficiently 23098and unaccessable to U mode.

Pseudo Instructions

Pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if $=$ zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if $>$ zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`beqz rs, offset`	`beq rs, x0, offset`	Branch if $=$ zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if $>$
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, rs, 0`	Jump register
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	Call far-away subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine
`fence`	`fence iorw, iorw`	Fence on all memory and I/O

Registers

Register	ABI Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP args/return values	Caller
f12-17	fa2-7	FP args	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

X86-64 Registers

64B register	32B	16B	8B	Description	Saver
rax	eax	ax	al	Return Value	Caller
rbx	ebx	bx	bl	Saved Register	Callee
rcx	ecx	cx	cl	Arg 4	Caller
rdx	edx	dx	dl	Arg 3	Caller
rsi	esi	si	sil	Arg 2	Caller
rdi	edi	di	dil	Arg 1	Caller
rsp	esp	sp	sil	Stack Pointer	N/A
rbp	ebp	bp	bpl	Frame Pointer	Callee
r8	r8d	r8w	r8b	Arg 5	Caller
r9	r9d	r9w	r9b	Arg 6	Caller
r10	r10d	r10w	r10b	Scratch	Caller
r11	r11d	r11w	r11b	Scratch	Caller
r12	r12d	r12w	r12b	Saved Register	Callee
r13	r13d	r13w	r13b	Saved Register	Callee
r14	r14d	r14w	r14b	Saved Register	Callee
r15	r15d	r15w	r15b	Saved Register	Callee

Most x86 instructions are a 2-argument form, with one of the arguments capable of being a memory location, not just a register or an immediate.

Mode	Meaning	Examples
Immediate	A 32b immediate value	`0x381` 42
Register	A register	`rax` `r11b`
Memory Location	Memory location	`[rax]` `[r11 + 32]`
Indexed Memory Location	The second register is multiplied by the stride (1, 2, 4, or 8 only)	`[rax + 8 * rbp]` [r11 + 4 * r12 + 32]

We use Intel syntax, which automatically inferrs the bit width for operations based on the register involved. For the rare case where one wishes to use an instruction with just an immediate and memory, we need to specify the bit width explicitly either by writing the immediate into a register first or by annotating the memory location with `byte ptr`, `word ptr`, `dword ptr`, or `qword ptr` for 1, 2, 4, and 8 bits respectively.

One can see the large variety possible through addressing in the following eamples for `mov`.

Example	Meaning
`mov rdi, rsi`	Move the contents of register `rsi` into `rdi`
`mov rdi, [rsi]`	Load the memory pointed to by `rsi` (equivalent to RISC-V `lw`
`mov rdi, 0xf00d`	Load the immediate into `rdi`.
`mov [rsi + 8], rdi`	Store the contents of `rdi` into the memory location `rsi + 8`
`mov [rsi + rdx * 8 + 32], rdi`	Store the contents of `rdi` into the memory location `rsi + rdi * 8 + 32`
`mov dword ptr [rsi + 32], -42`	Store the 32b immediate value -42 into the memory location `rsi + 32`

1 x86-64 Instructions

For our x86-64 programming we are using a deliberately RISC-like subset of x86, specifically excluding instructions like `push` and `pop` that manipulate the stack.

Instruction	Meaning
`add dst, src`	Add the source and destination together, storing the result in the destination
`and dst, src`	Performs a logical AND between the source and destination, storing the results in the destination
`call target`	Control flow transfer to label or register, decrements rsp by 8, storing rip into [rsp]
`cmov`*cc* `dst, src`	Conditional move from source to dst. See condition codes
`cmp src1, src2`	Compare the two values and set the flags for conditional move and branch
`div arg`	Unsigned division: This takes a 128b number composed of `rdx:rax` (that is, the upper 64b are in `rdx`) and uses the arg as the divisor, treating the arguments as signed. The quotient is placed in `rax` while the remainder is in `rdx`
`idiv arg`	Signed division: This takes a 128b number composed of `rdx:rax` (that is, the upper 64b are in `rdx`) and uses the arg as the divisor, treating the arguments as signed. The quotient is placed in `rax` while the remainder is in `rdx`
`imul dst, src`	Unsigned multiplication: This multiplies the source and destination together, writing it in the destination. The destination must be a register.
`imul dst, src, imm`	Unsigned multiplication: This multiplies the source by the 32b sign-extended immediate, writing it in the destination. The destination must be a register.
`j`*cc* `dst`	Conditional jump: Jump to the destination if the condition is true. See condition codes
`jmp dst`	Transfer control flow to the destination
`lea dst, src`	Load Effective Immediate: Load the address referred to in src into dst.
`mul src`	Signed multiplication: This multiples the source by the contents of the `rax` register, storing the lower 64b of the results in `rax` and the upper 64b in `rdx`.
`mov dst, src`	Move src into destination
`neg dst`	2s complement negation of the destination
`not dst`	Bitwise negation of the destination
`or dst, src`	Performs a logical OR between the source and destination, writing the results in the destination
`ret`	Loads rip as [rsp], increments rsp by 8, transfers control flow to rip
`shl dst, src`	Shifts the destination to the left by src bits
`sar dst, src`	Shifts the destination to the right by src bits, shifting in the sign bit
`shr dst, src`	Shifts the destination to the right by src bits, shifting in 0 bits on the left
`sub dst, src`	Calculates dst - src, writing the results into dst
`xor dst, src`	Performs a logical XOR between the source and destination, writing the results in the destination

X86 uses a "compare and conditional operation" motif for branches and conditional moves. Many instructions set various condition flags, but the most common is `cmp` which performs a subtraction but only sets the flags. This is not all of the mnemonics but it is the most important set.

Code	Meaning	Examples
`a`	Above (unsigned >)	`cmova, ja`
`ae`	Above or equal (unsigned >=)	`cmovae, jae`
`b`	Below (unsigned <)	`cmovb, jb`
`be`	Below or equal (unsigned <=)	`cmovbe, jbe`
`e`	Equal	`cmove, je`
`g`	Greater (signed >)	`cmovg, jg`
`ge`	Greater or equal (signed >=)	`cmovge, jge`
`l`	Less than (signed <)	`cmovl, jl`
`le`	Less than or equal (signed <=)	`cmovle, jle`
`ne`	Not equal	`cmovne, jne`
`ns`	Not sign (result was positive)	`cmovns, jns`
`nz`	Not zero	`cmovnz, jnz`
`s`	Sign (result was negative)	`cmovs, js`
`z`	Zero (result was zero)	`cmovz, jz`

