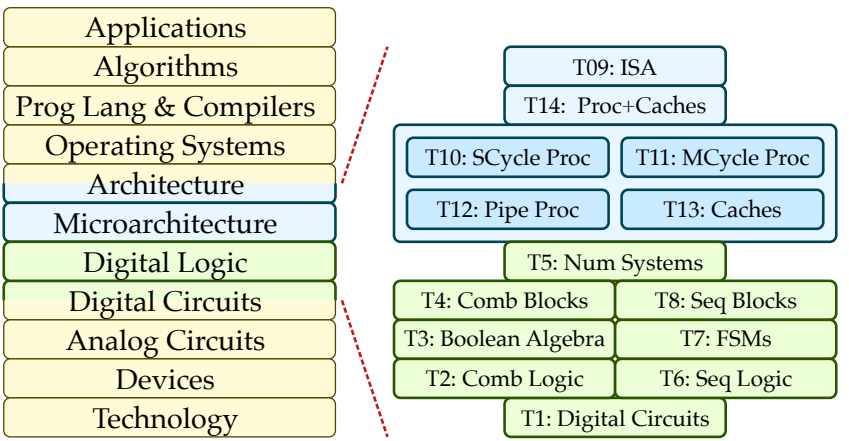
ECE 2300 Digital Logic and Computer Organization Fall 2025

Topic 9: Instruction Set Architecture

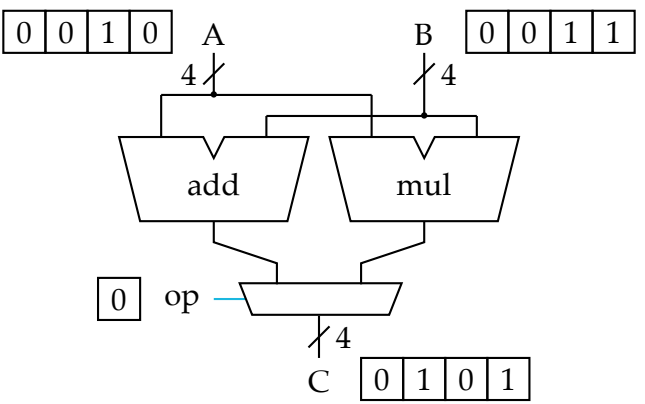
School of Electrical and Computer Engineering
Cornell University

revision: 2025-10-30-10-48



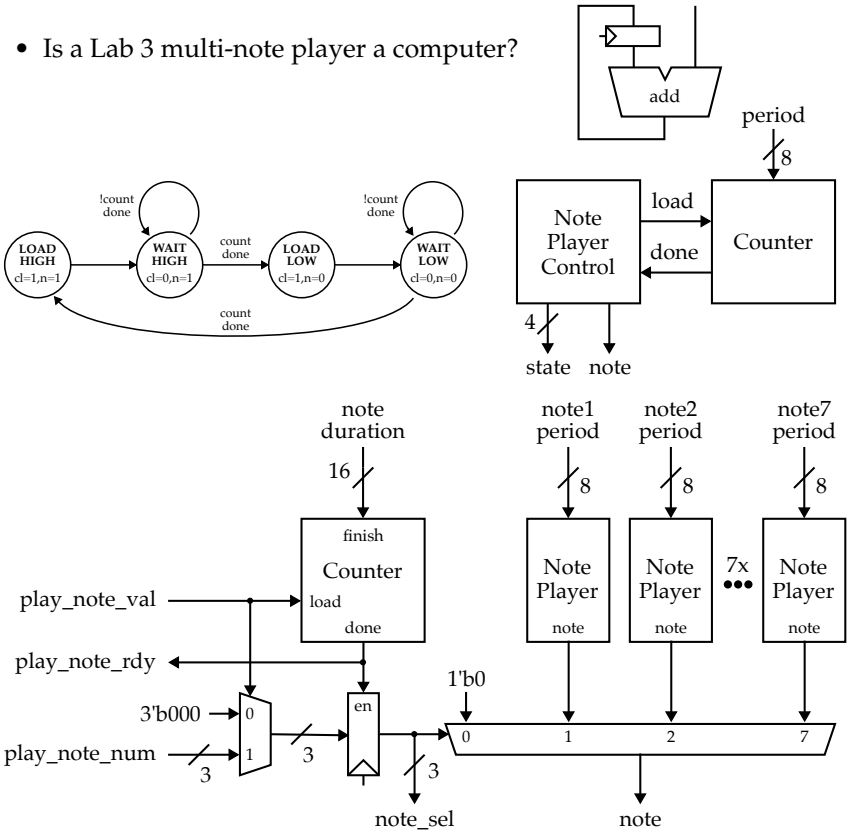
1. Computer Organization

- A **computer** is "an electronic device for processing and storing data (typically in binary form), according to general-purpose instructions (also in binary form) stored as a program in memory"
- **Computer organization** involves using number systems, combinational building blocks, and sequential building blocks to implement a computer
- Is our Lab 2 calculator a computer?



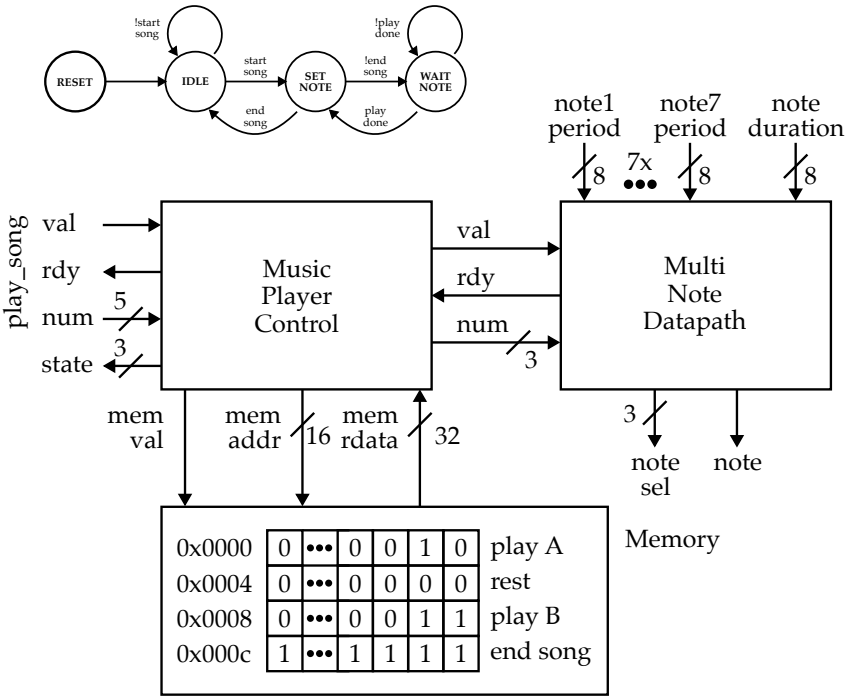
- A computer is "an electronic device for processing and storing data (typically in binary form), according to general-purpose instructions (also in binary form) stored as a program in memory"

- Is a Lab 3 multi-note player a computer?



- A computer is "an electronic device for processing and storing data (typically in binary form), according to general-purpose instructions (also in binary form) stored as a program in memory"

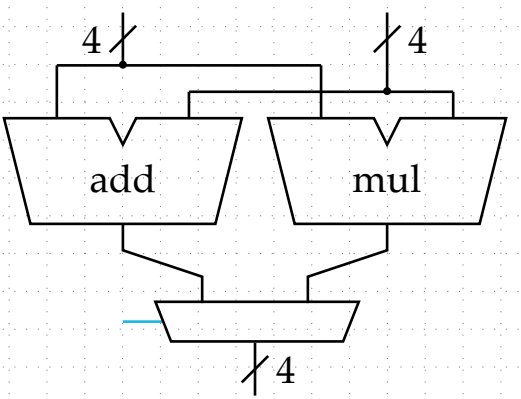
- Is a Lab 3 music player a computer?



- A computer is "an electronic device for processing and storing data (typically in binary form), according to general-purpose instructions (also in binary form) stored as a program in memory"

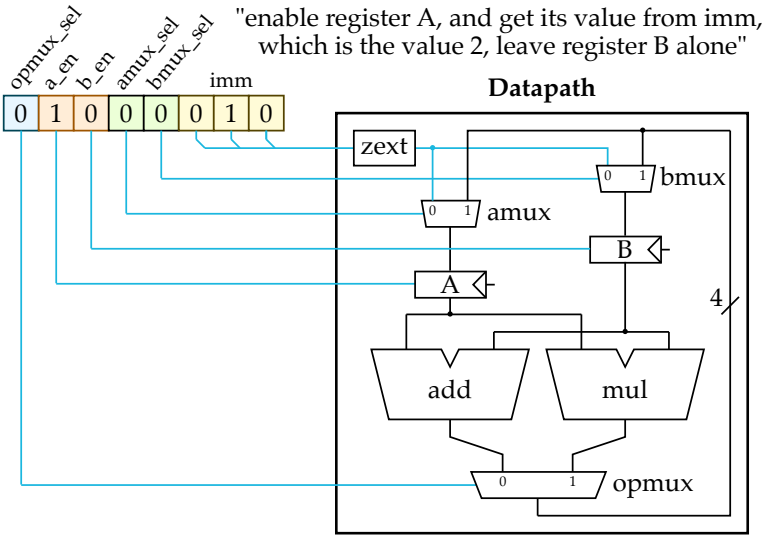
1.1. Pico-Processor

- Let's build a pico-processor [a]
  - Calculator: processing, general-purpose instructions
  - Music Player: storing, program in memory



[a] Not a micro-processor (10^{-6}), not a nano-processor (10^{-9}), but a pico-processor (10^{-12})

- An instruction is a binary value which tells a computer how to perform a specific operation



- A program is a sequence of instructions

opmux_sel	a_en	b_en	amux_sel	bmux_sel	imm		
0	1	0	0	0	0	1	0
0	0	1	0	0	0	1	1
0	1	0	1	1	0	0	0
1	0	1	1	1	0	0	0

"set register A to 2"

"set register B to 3"

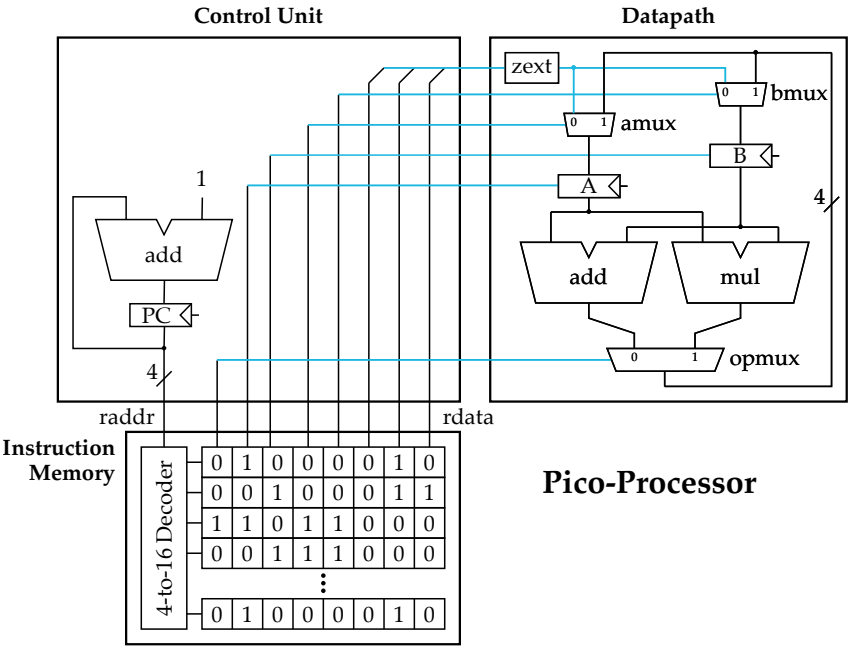
"set register A to A+B"

"set register B to A×B"

Machine Instructions make up a **Machine Program** (Program Binary)

Assembly Instructions make up an **Assembly Program**

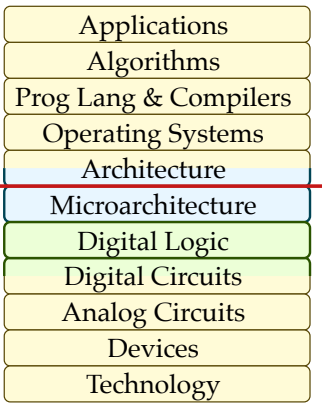
- A program stored in memory is just a sequence of machine instructions (binary values) stored in a memory array



- A computer is "an electronic device for processing and storing data (typically in binary form), according to general-purpose instructions (also in binary form) stored as a program in memory"

The pico-processor is an (extremely simple) computer!

1.2. Pico-Processor Instruction-Set Architecture



An instruction set architecture is the contract between software and hardware

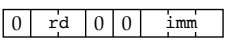
1. How is the data represented?
2. Where can the data be stored?
3. How can data be accessed?
4. What operations can be done on data?
5. How are instructions encoded?

WRIMM

`wrimm rd, imm`

R[rd] ← zext(imm)

PC ← PC + 1



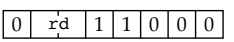
for register A, rd = 10
for register B, rd = 01

ADD

`add rd`

R[rd] ← R[rA] + R[rB]

PC ← PC + 1



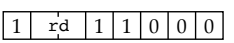
for register A, rd = 10
for register B, rd = 01

MUL

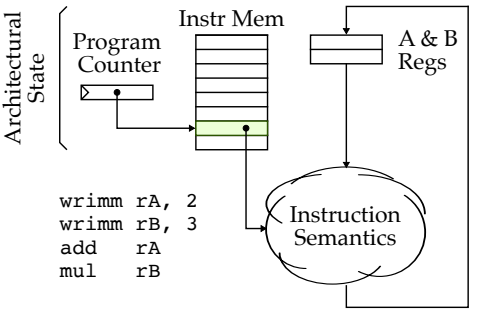
`mul rd`

R[rd] ← R[rA] × R[rB]

PC ← PC + 1



for register A, rd = 10
for register B, rd = 01

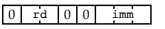


WRIMM

`wrimm rd, imm`

R[rd] ← zext(imm)

PC ← PC + 1



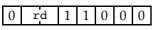
for register A, rd = 10
for register B, rd = 01

ADD

`add rd`

R[rd] ← R[rA] + R[rB]

PC ← PC + 1



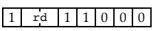
for register A, rd = 10
for register B, rd = 01

MUL

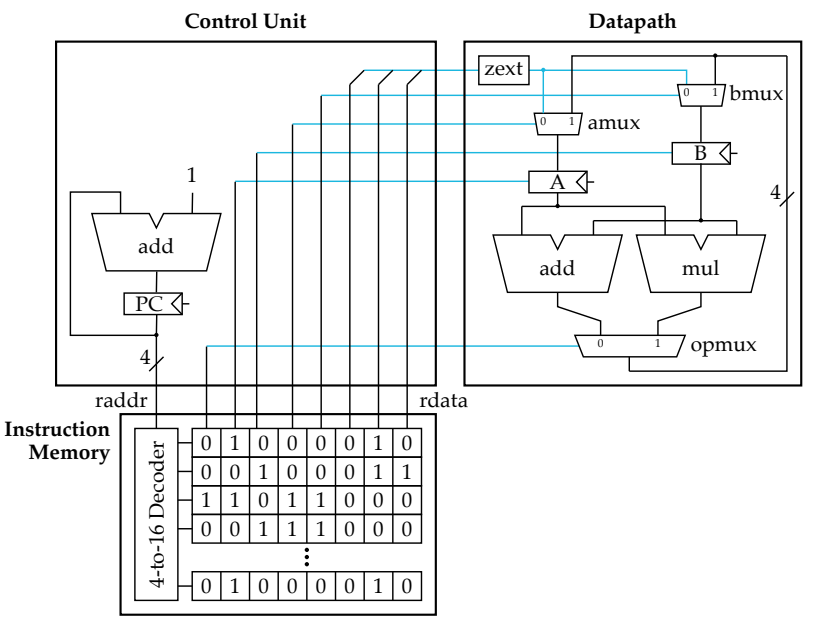
`mul rd`

R[rd] ← R[rA] × R[rB]

PC ← PC + 1



for register A, rd = 10
for register B, rd = 01



Assembly Program

```
□□□ wrimm rA, 2
□□□ wrimm rB, 3
□□□ add   rA
□□□ mul   rB
```

Memory

15
⋮
6
5
4
3
2
1
0

Program Counter

Registers

rB
rA

Assembly Program

```
□□□ wrimm rA, 2
□□□ wrimm rB, 0
□□□ add   rB
□□□ mul   rA
□□□ mul   rA
```

Memory

15
⋮
6
5
4
3
2
1
0

Program Counter

Registers

rB
rA

1.3. From Pico-Processor to Realistic ISAs

1. How is the data represented?
   - Representations for characters, integers, floating-point
   - Integer formats can be signed or unsigned
   - Floating-point formats can be single- or double-precision
   - Byte addresses can ordered within a word as either little- or big-endian

2. Where can the data be stored?
   - Registers: general-purpose, floating-point, control
   - Memory: different addresses spaces for heap, stack, I/O

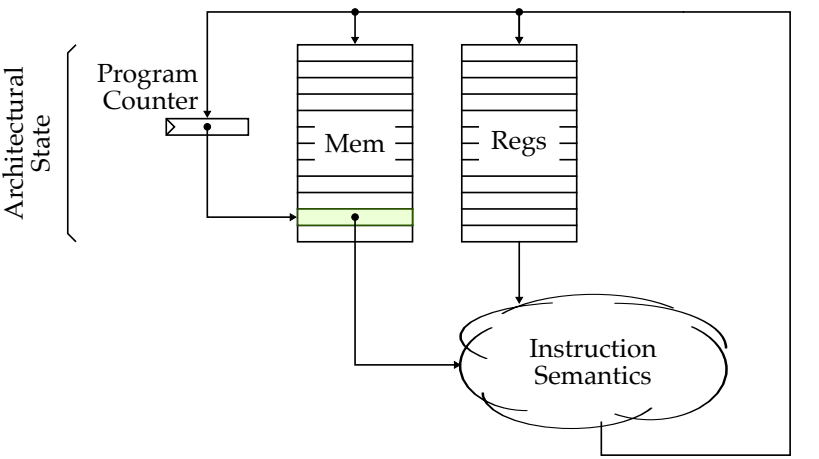
3. How can data be accessed?
   - Register: operand stored in registers
   - Immediate: operand is an immediate in the instruction
   - Direct: address of operand in memory is stored in instruction
   - Register Indirect: address of operand in memory is stored in register
   - Displacement: register indirect, addr is added to immediate
   - Autoincrement/decrement: register indirect, addr is automatically adj
   - PC-Relative: displacement is added to the program counter

4. What operations can be done on data?
   - Integer and floating-point arithmetic instructions
   - Register and memory data movement instructions
   - Control transfer instructions
   - System control instructions

5. How are instructions encoded?
   - Opcode, addresses of operands and destination, next instruction
   - Variable length vs. fixed length

2. TinyRV1

- RISC-V instruction set architecture
  - Brand new free, open instruction set architecture
  - Significant excitement around RISC-V hardware/software ecosystem
  - Helping to energize "open-source hardware"
  - Specifically designed to encourage subsetting and extension
  - Link to official ISA manual on course webpage
- TinyRV1 instruction set architecture
  - Subset we use in this course
  - Small enough for teaching
- TinyRV1 functional processor model



1. How is the data represented?

2. Where can the data be stored?

3. How can data be accessed?

4. What operations can be done on data?

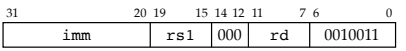
5. How are instructions encoded?

2.1. Arithmetic Instructions

ADDI

```
addi rd, rs1, imm
```

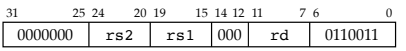
R[rd] ← R[rs1] + sext(imm)
PC ← PC + 4



ADD

```
add rd, rs1, rs2
```

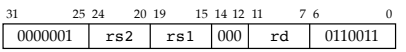
R[rd] ← R[rs1] + R[rs2]
PC ← PC + 4



MUL

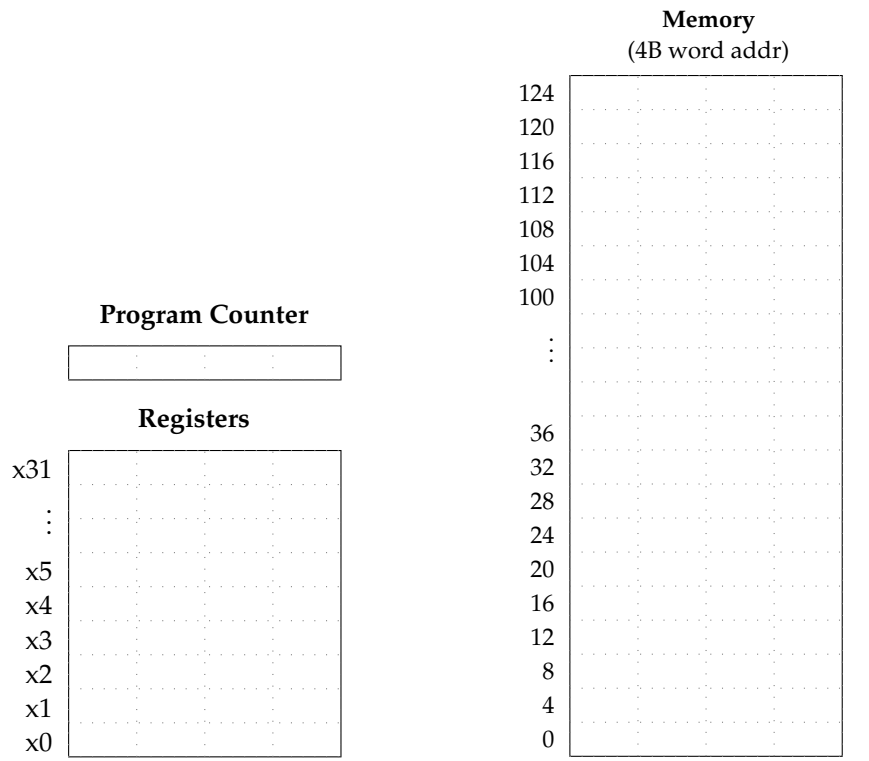
```
mul rd, rs1, rs2
```

R[rd] ← R[rs1] × R[rs2]
PC ← PC + 4



- Instruction set architecture specification includes:
  - Instruction name
  - Instruction assembly format
  - Instruction semantics
  - Instruction encoding
- Explanation of instruction semantics
  - `imm` : immediate value stored in the instruction
  - R[rX] : 4-bytes (32-bits) value stored in general-purpose register `rX`
  - sext(V) : sign extend the value `V` to 32-bits
    (fill the 32-bits with the immediate's sign bit)
- *Arithmetic Instructions*
  - Read a 4-byte (32-bit) value from general-purpose registers or immediates
  - Perform a 4-byte (32-bit) arithmetic operation
  - Write the resulting 4-byte (32-bit) value to a general-purpose register

```
addi x1, x0, 13  # R[x1] = 13
addi x2, x0, 42  # R[x2] = 42
add  x3, x1, x2  # R[x3] = R[x1] + R[x2]
addi x4, x0,  2  # R[x4] = 2
mul  x1, x2, x4  # R[x1] = R[x2] * R[x4]
```



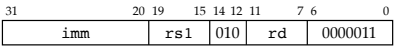
2.2. Memory Instructions

LW

`lw rd, imm(rs1)`

R[`rd`] ← M[R[`rs1`] + sext(`imm`)]

PC ← PC + 4

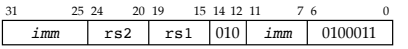


SW

`sw rs2, imm(rs1)`

M[R[`rs1`] + sext(`imm`)] ← R[`rs2`]

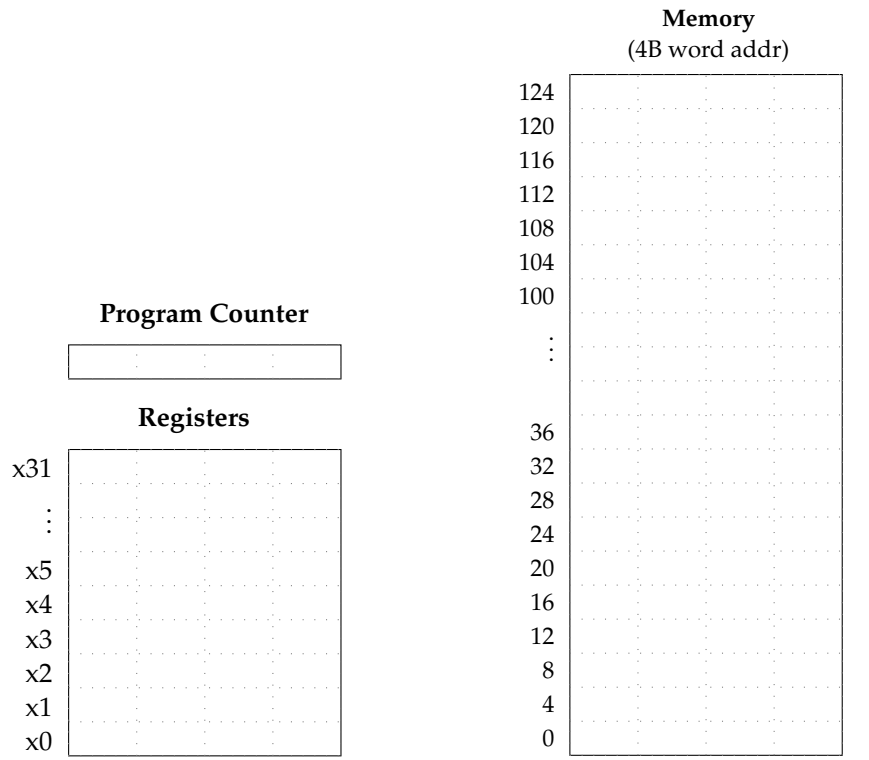
PC ← PC + 4



`imm` ↔ `{ inst[31:25], inst[11:7] }`

- Explanation of instruction semantics
  - M[`A`] : 4-byte (32-bit) memory value at byte address `A`
- *Load Instructions*
  - Generate address using values from a general-purpose reg and immediate
  - Read a 4-byte (32-bit) value from the corresponding memory address
  - Write this 4-byte (32-bit) value to a general-purpose register
- *Store Instructions*
  - Generate address using values from a general-purpose reg and immediate
  - Read a 4-byte (32-bit) value from a general-purpose register
  - Write this 4-byte (32-bit) value to the corresponding memory address
- Data memory addresses
  - Memory addresses will always be *byte* addresses (i.e., 4-byte word aligned), thus bottom two bits of the address must always be zero!
  - Unaligned memory accesses (i.e., bottom two bits of the address are non-zero) results in undefined behavior
  - *Virtual address space* is the entire range of addressable memory locations (2^{32} or 4 billion bytes = 1 billion 4-byte words)
  - *Physical address space* is the range of memory locations that can actually be used for storage (2^{9} or 512 bytes = 128 4-byte words)

```
addi x1, x0, 100  # R[x1] = 100
addi x2, x0, 13   # R[x2] = 13
sw   x2, 0(x1)    # M[ R[x1] ] = R[x2]
lw   x3, 0(x1)    # R[x3] = M[ R[x1] ]
```

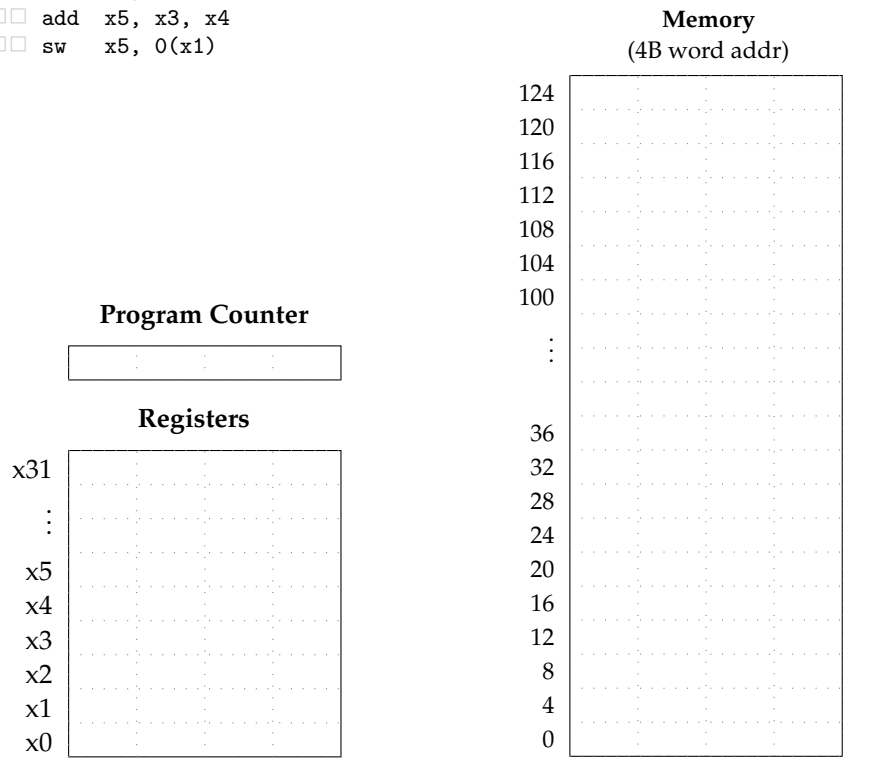


```
addi x1, x0, 100

addi x2, x0, 13
sw   x2, 0(x1)

addi x2, x0, 42
sw   x2, 4(x1)

lw   x3, 0(x1)
lw   x4, 4(x1)
add  x5, x3, x4
sw   x5, 0(x1)
```



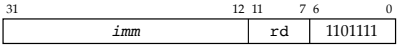
2.3. Control Instructions

JAL

`jal rd, addr`

R[rd] ← PC + 4

PC ← addr



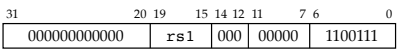
imm ↔ addr – PC

```
imm ↔ { inst[31], inst[19:12],
        inst[20], inst[30:21], 0 }
```

JR

`jr rs1`

PC ← R[rs1]

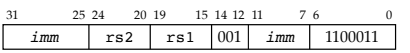


BNE

`bne rs1, rs2, addr`

if (R[rs1] != R[rs2]) PC ← addr

else PC ← PC + 4

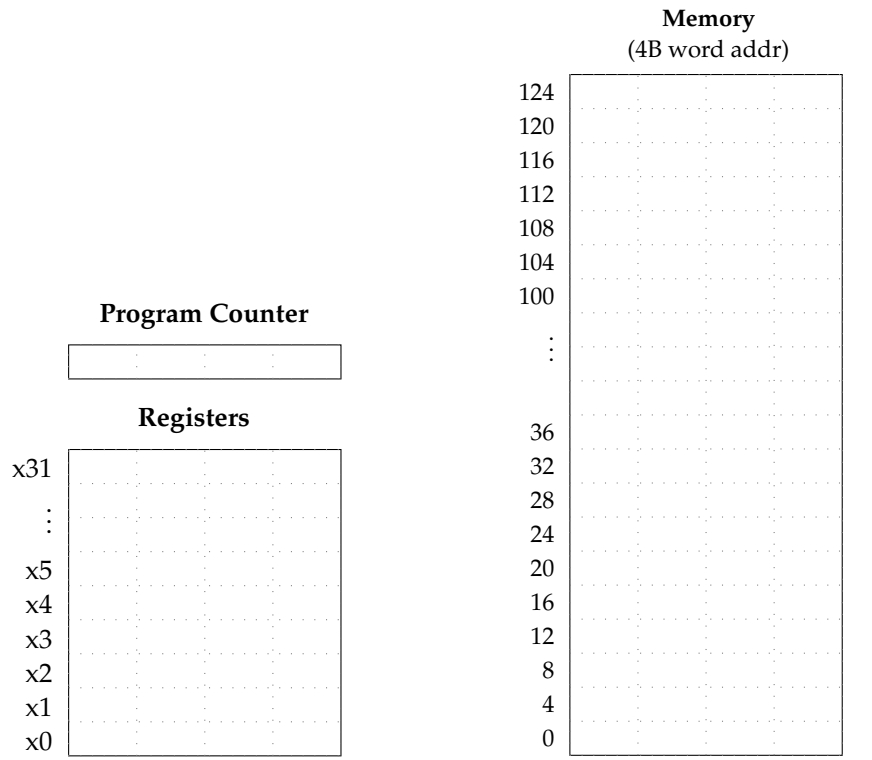


imm ↔ addr – PC

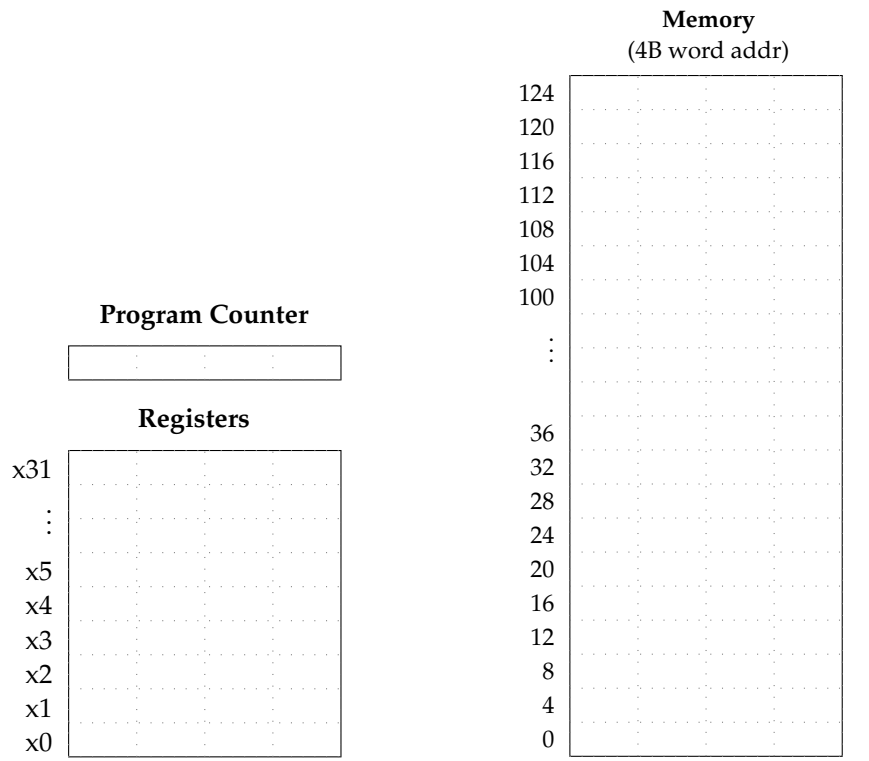
```
imm ↔ { inst[31], inst[7],
        inst[30:25], inst[11:8], 0 }
```

- Explanation of instruction semantics
  - addr : memory byte address
- *Jump and Link Instructions*
  - Write the current program counter (PC) to a general-purpose register
  - Unconditionally write the PC with a new target address
- *Jump Register Instructions*
  - Read a new target address from a general-purpose register
  - Unconditionally write the PC with the target address
- *Branch Instructions*
  - Compare the values from two general-purpose registers
  - Conditionally write PC with a new target address based on comparison
- Instruction memory addresses
  - Memory addresses will always be *byte* addresses (i.e., 4-byte word aligned), thus bottom two bits of the address must always be zero!
  - If target address is encoded as an immediate relative to the PC, then control instruction only supports a limited range
  - If target address is in a register supports entire virtual address space

```
  addi x1, x0, 0   # R[x1] = 0
  bne  x1, x0, L1  # if ( R[x1] == R[x0] ) goto L1
  addi x2, x0, 13  # R[x2] = 13
  jal  x0, L2      # goto L2
L1:
  addi x2, x0, 42  # R[x2] = 42
L2:
```



```
  addi x1, x0, 1
  bne  x1, x0, L1
  addi x2, x0, 13
  jal  x0, L2
L1:
  addi x2, x0, 42
L2:
```



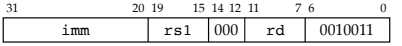
2.4. TinyRV1 ISA Specification

ADDI

`addi rd, rs1, imm`

R[rd] ← R[rs1] + sext(imm)

PC ← PC + 4

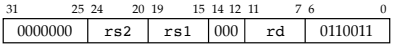


ADD

`add rd, rs1, rs2`

R[rd] ← R[rs1] + R[rs2]

PC ← PC + 4

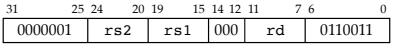


MUL

`mul rd, rs1, rs2`

R[rd] ← R[rs1] × R[rs2]

PC ← PC + 4

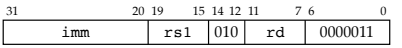


LW

`lw rd, imm(rs1)`

R[rd] ← M[R[rs1] + sext(imm)]

PC ← PC + 4

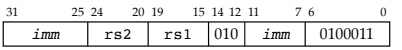


SW

`sw rs2, imm(rs1)`

M[R[rs1] + sext(imm)] ← R[rs2]

PC ← PC + 4



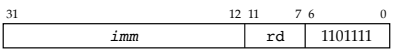
`imm ↔ { inst[31:25], inst[11:7] }`

JAL

`jal rd, addr`

R[rd] ← PC + 4

PC ← addr



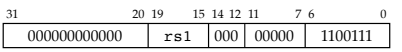
`imm ↔ addr – PC`

```
imm ↔ { inst[31], inst[19:12],
        inst[20], inst[30:21], 0 }
```

JR

`jr rs1`

PC ← R[rs1]

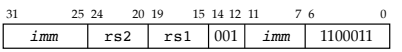


BNE

`bne rs1, rs2, addr`

if (R[rs1] != R[rs2]) PC ← addr

else PC ← PC + 4



`imm ↔ addr – PC`

```
imm ↔ { inst[31], inst[7],
inst[30:25], inst[11:8], 0 }
```

Note 1 R[x0] is always zero Note 2 PC reset value is 0x00000000

Note 3 Valid physical mem range `0x00000000-0x00001fff` w/ 8 additional memory-mapped IO addresses

Note 4 Invalid or unnaligned memory addresses result in undefined behavior

Note 5 `nop` is pseudo-instruction for `addi x0, x0, 0`

Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	`LB rd,rs1,imm`	
	Load Halfword	I	`LH rd,rs1,imm`	
	Load Word	I	`LW rd,rs1,imm`	`L{D|Q} rd,rs1,imm`
	Load Byte Unsigned	I	`LBU rd,rs1,imm`	
	Load Half Unsigned	I	`LHU rd,rs1,imm`	`L{W|D}U rd,rs1,imm`
Stores	Store Byte	S	`SB rs1,rs2,imm`	
	Store Halfword	S	`SH rs1,rs2,imm`	
	Store Word	S	`SW rs1,rs2,imm`	`S{D|Q} rs1,rs2,imm`
Shifts	Shift Left	R	`SLL rd,rs1,rs2`	`SLL{W|D} rd,rs1,rs2`
	Shift Left Immediate	I	`SLLI rd,rs1,shamt`	`SLLI{W|D} rd,rs1,shamt`
	Shift Right	R	`SRL rd,rs1,rs2`	`SRL{W|D} rd,rs1,rs2`
	Shift Right Immediate	I	`SRLI rd,rs1,shamt`	`SRLI{W|D} rd,rs1,shamt`
	Shift Right Arithmetic	R	`SRA rd,rs1,rs2`	`SRA{W|D} rd,rs1,rs2`
	Shift Right Arith Imm	I	`SRAI rd,rs1,shamt`	`SRAI{W|D} rd,rs1,shamt`
Arithmetic	ADD	R	`ADD rd,rs1,rs2`	`ADD{W|D} rd,rs1,rs2`
	ADD Immediate	I	`ADDI rd,rs1,imm`	`ADDI{W|D} rd,rs1,imm`
	SUBtract	R	`SUB rd,rs1,rs2`	`SUB{W|D} rd,rs1,rs2`
	Load Upper Imm	U	`LUI rd,imm`	
	Add Upper Imm to PC	U	`AUIPC rd,imm`	
Logical	XOR	R	`XOR rd,rs1,rs2`	
	XOR Immediate	I	`XORI rd,rs1,imm`	
	OR	R	`OR rd,rs1,rs2`	
	OR Immediate	I	`ORI rd,rs1,imm`	
	AND	R	`AND rd,rs1,rs2`	
	AND Immediate	I	`ANDI rd,rs1,imm`	
Compare	Set <	R	`SLT rd,rs1,rs2`	
	Set < Immediate	I	`SLTI rd,rs1,imm`	
	Set < Unsigned	R	`SLTU rd,rs1,rs2`	
	Set < Imm Unsigned	I	`SLTIU rd,rs1,imm`	
Branches	Branch =	SB	`BEQ rs1,rs2,imm`	
	Branch ≠	SB	`BNE rs1,rs2,imm`	
	Branch <	SB	`BLT rs1,rs2,imm`	
	Branch ≥	SB	`BGE rs1,rs2,imm`	
	Branch < Unsigned	SB	`BLTU rs1,rs2,imm`	
	Branch ≥ Unsigned	SB	`BGEU rs1,rs2,imm`	
Jump & Link	J&L	UJ	`JAL rd,imm`	
	Jump & Link Register	UJ	`JALR rd,rs1,imm`	
Synch	Synch thread	I	`FENCE`	
	Synch Instr & Data	I	`FENCE.I`	
System	System CALL	I	`SCALL`	
	System BREAK	I	`SBREAK`	
Counters	ReaD CYCLE	I	`RDCYCLE rd`	
	ReaD CYCLE upper Half	I	`RDCYCLEH rd`	
	ReaD TIME	I	`RDTIME rd`	
	ReaD TIME upper Half	I	`RDTIMEH rd`	
	ReaD INSTR RETired	I	`RDINSTRET rd`	
	ReaD INSTR upper Half	I	`RDINSTRETH rd`	

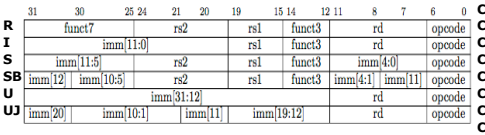
RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	`CSRRW rd,csr,rs1`
	Atomic Read & Set Bit	`CSRRS rd,csr,rs1`
	Atomic Read & Clear Bit	`CSRRC rd,csr,rs1`
	Atomic R/W Imm	`CSRRWI rd,csr,imm`
	Atomic Read & Set Bit Imm	`CSRRSI rd,csr,imm`
	Atomic Read & Clear Bit Imm	`CSRRCI rd,csr,imm`
Change Level	Env. Call	`ECALL`
	Environment Breakpoint	`EBREAK`
	Environment Return	`ERET`
Trap Redirect	to Supervisor	`MRTS`
	Redirect Trap to Hypervisor	`MRTH`
	Hypervisor Trap to Supervisor	`HRTS`
Interrupt	Wait for Interrupt	`WFI`
MMU	Supervisor FENCE	`SFENCE.VM rs1`

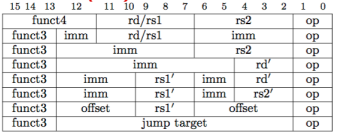
Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC	RVI equivalent
Loads	Load Word	CL	`C.LW rd',rs1',imm`	`LW rd',rs1',imm*4`
	Load Word SP	CI	`C.LWSP rd,imm`	`LW rd,sp,imm*4`
	Load Double	CL	`C.LD rd',rs1',imm`	`LD rd',rs1',imm*8`
	Load Double SP	CI	`C.LDSP rd,imm`	`LD rd,sp,imm*8`
	Load Quad	CL	`C.LQ rd',rs1',imm`	`LQ rd',rs1',imm*16`
	Load Quad SP	CI	`C.LQSP rd,imm`	`LQ rd,sp,imm*16`
Stores	Store Word	CS	`C.SW rs1',rs2',imm`	`SW rs1',rs2',imm*4`
	Store Word SP	CSS	`C.SWSP rs2,imm`	`SW rs2,sp,imm*4`
	Store Double	CS	`C.SD rs1',rs2',imm`	`SD rs1',rs2',imm*8`
	Store Double SP	CSS	`C.SDSP rs2,imm`	`SD rs2,sp,imm*8`
	Store Quad	CS	`C.SQ rs1',rs2',imm`	`SQ rs1',rs2',imm*16`
	Store Quad SP	CSS	`C.SQSP rs2,imm`	`SQ rs2,sp,imm*16`
Arithmetic	ADD	CR	`C.ADD rd,rs1`	`ADD rd,rd,rs1`
	ADD Word	CR	`C.ADDW rd,rs1`	`ADDW rd,rd,imm`
	ADD Immediate	CI	`C.ADDI rd,imm`	`ADDI rd,rd,imm`
	ADD Word Imm	CI	`C.ADDIW rd,imm`	`ADDIW rd,rd,imm`
	ADD SP Imm * 16	CI	`C.ADDI16SP x0,imm`	`ADDI sp,sp,imm*16`
	ADD SP Imm * 4	CIW	`C.ADDI4SPN rd',imm`	`ADDI rd',sp,imm*4`
	Load Immediate	CI	`C.LI rd,imm`	`ADDI rd,x0,imm`
	Load Upper Imm	CI	`C.LUI rd,imm`	`LUI rd,imm`
	MoVe	CR	`C.MV rd,rs1`	`ADD rd,rs1,x0`
	SUB	CR	`C.SUB rd,rs1`	`SUB rd,rd,rs1`
Shifts	Shift Left Imm	CI	`C.SLLI rd,imm`	`SLLI rd,rd,imm`
Branches	Branch=0	CB	`C.BEQZ rs1',imm`	`BEQ rs1',x0,imm`
	Branch≠0	CB	`C.BNEZ rs1',imm`	`BNE rs1',x0,imm`
Jump	Jump	CJ	`C.J imm`	`JAL x0,imm`
	Jump Register	CR	`C.JR rd,rs1`	`JALR x0,rs1,0`
Jump & Link	J&L	CJ	`C.JAL imm`	`JAL ra,imm`
	Jump & Link Register	CR	`C.JALR rs1`	`JALR ra,rs1,0`
System	Env. BREAK	CI	`C.EBREAK`	`EBREAK`

32-bit Instruction Formats



16-bit (RVC) Instruction Formats



RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

Free & Open RISC-V Reference Card (riscv.org) ②

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)		+RV{64,128}	
Multiply	MULtiply	R	MUL	rd,rs1,rs2	MUL{W|D}	rd,rs1,rs2
	MULtiply upper Half	R	MULH	rd,rs1,rs2		
	MULtiply Half Sign/Uns	R	MULHSU	rd,rs1,rs2		
	MULtiply upper Half Uns	R	MULHU	rd,rs1,rs2		
Divide	DIVide	R	DIV	rd,rs1,rs2	DIV{W|D}	rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2		
Remainder	REMainder	R	REM	rd,rs1,rs2	REM{W|D}	rd,rs1,rs2
	REMainder Unsigned	R	REMU	rd,rs1,rs2	REMU{W|D}	rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)		+RV{64,128}	
Load	Load Reserved	R	LR.W	rd,rs1	LR.{D|Q}	rd,rs1
Store	Store Conditional	R	SC.W	rd,rs1,rs2	SC.{D|Q}	rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W	rd,rs1,rs2	AMOSWAP.{D|Q}	rd,rs1,rs2
Add	ADD	R	AMOADD.W	rd,rs1,rs2	AMOADD.{D|Q}	rd,rs1,rs2
Logical	XOR	R	AMOXOR.W	rd,rs1,rs2	AMOXOR.{D|Q}	rd,rs1,rs2
	AND	R	AMOAND.W	rd,rs1,rs2	AMOAND.{D|Q}	rd,rs1,rs2
	OR	R	AMOOR.W	rd,rs1,rs2	AMOOR.{D|Q}	rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W	rd,rs1,rs2	AMOMIN.{D|Q}	rd,rs1,rs2
	MAXimum	R	AMOMAX.W	rd,rs1,rs2	AMOMAX.{D|Q}	rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W	rd,rs1,rs2	AMOMINU.{D|Q}	rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W	rd,rs1,rs2	AMOMAXU.{D|Q}	rd,rs1,rs2

Three Optional Floating-Point Instruction Extensions: RVF, RVD, & RVQ

Category	Name	Fmt	RV32{F|D|Q} (HP/SP,DP,QP FI Pt)		+RV{64,128}	
Move	Move from Integer	R	FMV.{H|S}.X	rd,rs1	FMV.{D|Q}.X	rd,rs1
	Move to Integer	R	FMV.X.{H|S}	rd,rs1	FMV.X.{D|Q}	rd,rs1
Convert	Convert from Int	R	FCVT.{H|S|D|Q}.W	rd,rs1	FCVT.{H|S|D|Q}.{L|T}	rd,rs1
	Convert from Int Unsigned	R	FCVT.{H|S|D|Q}.WU	rd,rs1	FCVT.{H|S|D|Q}.{L|T}U	rd,rs1
	Convert to Int	R	FCVT.W.{H|S|D|Q}	rd,rs1	FCVT.{L|T}.{H|S|D|Q}	rd,rs1
	Convert to Int Unsigned	R	FCVT.WU.{H|S|D|Q}	rd,rs1	FCVT.{L|T}U.{H|S|D|Q}	rd,rs1
Load	Load	I	FL{W,D,Q}	rd,rs1,imm		
Store	Store	S	FS{W,D,Q}	rs1,rs2,imm		
Arithmetic	ADD	R	FADD.{S|D|Q}	rd,rs1,rs2		
	SUBtract	R	FSUB.{S|D|Q}	rd,rs1,rs2		
	MULtiply	R	FMUL.{S|D|Q}	rd,rs1,rs2		
	DIVide	R	FDIV.{S|D|Q}	rd,rs1,rs2		
	SQuare RooT	R	FSQRT.{S|D|Q}	rd,rs1		
Mul-Add	Multiply-ADD	R	FMADD.{S|D|Q}	rd,rs1,rs2,rs3		
	Multiply-SUBtract	R	FMSUB.{S|D|Q}	rd,rs1,rs2,rs3		
	Negative Multiply-SUBtract	R	FNMSUB.{S|D|Q}	rd,rs1,rs2,rs3		
	Negative Multiply-ADD	R	FNMADD.{S|D|Q}	rd,rs1,rs2,rs3		
Sign Inject	SiGN source	R	FSGNJ.{S|D|Q}	rd,rs1,rs2		
	Negative SiGN source	R	FSGNJN.{S|D|Q}	rd,rs1,rs2		
	Xor SiGN source	R	FSGNJX.{S|D|Q}	rd,rs1,rs2		
Min/Max	MINimum	R	FMIN.{S|D|Q}	rd,rs1,rs2		
	MAXimum	R	FMAX.{S|D|Q}	rd,rs1,rs2		
Compare	Compare Float =	R	FEQ.{S|D|Q}	rd,rs1,rs2		
	Compare Float <	R	FLT.{S|D|Q}	rd,rs1,rs2		
	Compare Float ≤	R	FLE.{S|D|Q}	rd,rs1,rs2		
Categorization	Classify Type	R	FCLASS.{S|D|Q}	rd,rs1		
Configuration	Read Status	R	FRCSR	rd		
	Read Rounding Mode	R	FRRM	rd		
	Read Flags	R	FRFLAGS	rd		
	Swap Status Reg	R	FSCSR	rd,rs1		
	Swap Rounding Mode	R	FSRM	rd,rs1		
	Swap Flags	R	FSFLAGS	rd,rs1		
	Swap Rounding Mode Imm	I	FSRMI	rd,imm		
	Swap Flags Imm	I	FSFLAGSI	rd,imm		

RISC-V Calling Convention

Register	ABI Name	Saver	Description
x0	zero	---	Hard-wired zero
x1	ra	Caller	Return address
x2	sp	Callee	Stack pointer
x3	gp	---	Global pointer
x4	tp	---	Thread pointer
x5-7	t0-2	Caller	Temporaries
x8	s0/fp	Callee	Saved register/frame pointer
x9	s1	Callee	Saved register
x10-11	a0-1	Caller	Function arguments/return values
x12-17	a2-7	Caller	Function arguments
x18-27	s2-11	Callee	Saved registers
x28-31	t3-t6	Caller	Temporaries
f0-7	ft0-7	Caller	FP temporaries
f8-9	fs0-1	Callee	FP saved registers
f10-11	fa0-1	Caller	FP arguments/return values
f12-17	fa2-7	Caller	FP arguments
f18-27	fs2-11	Callee	FP saved registers
f28-31	ft8-11	Caller	FP temporaries

RISC-V calling convention and five optional extensions: 10 multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 25 floating-point instructions each for single-, double-, and quadruple-precision (RV32F, RV32D, RV32Q). The latter add registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. Each larger address adds some instructions: 4 for RVM, 11 for RVA, and 6 each for RVF/D/Q. Using regex notation, `{}` means set, so `L{D|Q}` is both `LD` and `LQ`. See risc.org. (8/21/15 revision)