RISC-V ISA

RISC-V ISA

- base *integer* ISA
  - must be present in any implementation
  - restricted to a minimal set of instructions sufficient to provide a reasonable target for compilers, assemblers, linkers, and OSs
- plus optional extensions to the base ISA
  - standard extensions: generally useful and should not conflict with other standard extensions
  - non-standard extension: may be highly specialized, or may conflict with other standard or non-standard extensions

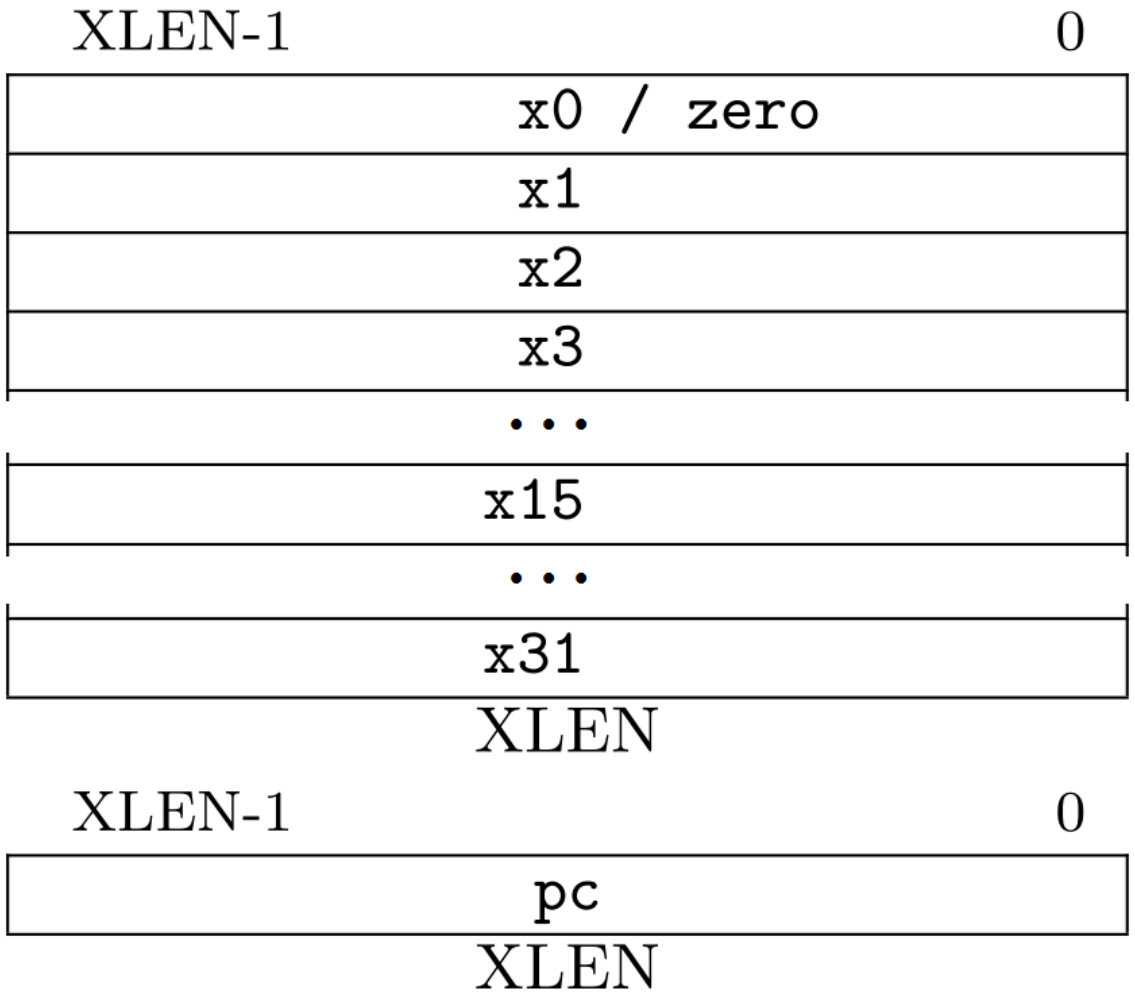
Base integer ISA

- is very similar to that of the early RISC processors
- characterized by the width (XLEN) of the integer registers and the *corresponding size of the user address space*
- variants:
  - RV32I – 32-bit user-level address space
    - RV32E - subset variant to support small microcontrollers
  - RV64I – 64-bit user-level address space
  - RV128I - 128-bit user-level address space
    - straightforward extrapolation of the existing RV32I and RV64I designs
    - It is not clear when a flat address space larger than 64 bits will be required

Standard extensions

- “M” Standard Extension for Integer Multiplication and Division;
- “F”/”D”/”Q” Standard Extensions for Single/Double/Quad-Precision Floating-Point
- “L” Standard Extension for *Decimal Floating-Point*
- “C” Standard Extension for Compressed Instructions
- other (complete or planned)

Registers (1)



* RV32E reduces the integer register count to 16 general-purpose registers, (x0–x15)

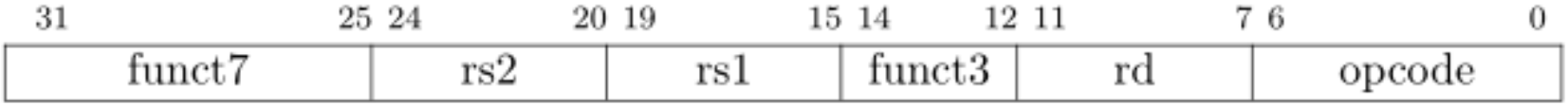
Registers (2)

- XLEN = 32 (RV32I, RV32E) / 64 (RV64I)
- signed integers, 2's complement, or
- unsigned integers
- x0 is hardwired to the constant 0
- pc holds the byte address of the current instruction

Address space

- XLEN-bit address space
- byte-addressed (8-bit bytes)
- little-endian

Instructions (R-type)



- opcode (major opcode) – group of instructions
- funct3, funct7 – type of operation
- rs1, rs2 – source registers
- rd – destination register

rd = rs1 *op* rs2

Basic instructions (R-type)

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	111	rd	0110011	AND
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA

```
add x10, x11, x12   # x10 = x11 + x12
and x13, x13, x14   # x13 &= x14
```

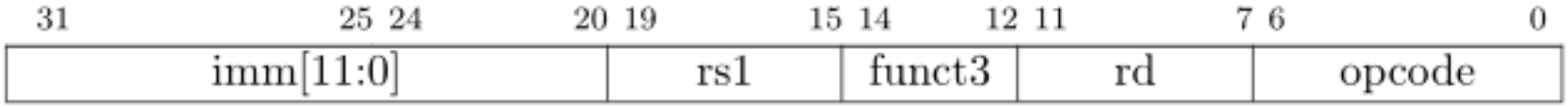
Logical operations

- bitwise AND/OR/XOR
- NOT can be implemented via XOR
  - NOT a = a XOR (-1)

Shift operations

- SLL - logical shift left
- SRL/SRA – logical/arithmetical shift right
- by the shift amount held in the lower bits of register rs2 (5 in RV32I / 6 in RV64I)
  - rs1 << 32 (32=1$\underline{00000}_2$) does nothing in RV32I;
  - rs1 << -1 (-1=1…1$\underline{11111}_2$) means rs1 << 31 in RV32I;
- barrel shifters are often utilized

Instructions (I-type)



- opcode (major opcode) – group of instructions
- funct3 – type of operation
- rs1 – register operand
- imm – immediate operand
- rd – destination register

rd = rs1 *op* *signext*(imm)

Basic instructions (I-type)

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	111	rd	0010011	ANDI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	100	rd	0010011	XORI
0000000	shamt		rs1	001	rd	0010011	SLLI
0000000	shamt		rs1	101	rd	0010011	SRLI
0100000	shamt		rs1	101	rd	0010011	SRAI

* the shift amount is encoded in the lower bits of the I-immediate field

```
slli x6, x5, 2  # x6 = x5 << 2 (= x5 * 4)
andi x7, x5, 0x1F  # x7 = x5 & 0x1F (mask lsb)
```

Load instructions (1)

31 27	26 25 24 20	19 15	14 12	11 7	6 0	
imm[11:0]		rs1	funct3	rd	opcode	I-type
RV32I Base Instruction Set						
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
RV64I Base Instruction Set (in addition to RV32I)						
imm[11:0]		rs1	110	rd	0000011	LWU
imm[11:0]		rs1	011	rd	0000011	LD

rd ← Mem[rs1 + *signext*(imm)]

```
lw x10, -8(x5)    # x10 ← Mem[ x5 - 8 + 0..3 ]
```

Load instructions (2)

- LB[U]/LH[U]/LW[U]/LD loads a 8/16/32/64-bit value from memory (byte/half-word/word/ double-word)
- Lx sign-extends the value before storing in rd
- LxU zero-extends the value before storing in rd

Store instructions (1)

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type

RV32I Base Instruction Set

imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

RV64I Base Instruction Set (in addition to RV32I)

imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD

Mem[rs1 + *signext*(imm)] ← rs2

```
sw x10, 0(x6)    # Mem[ x6 + 0..3 ] ← x10
```

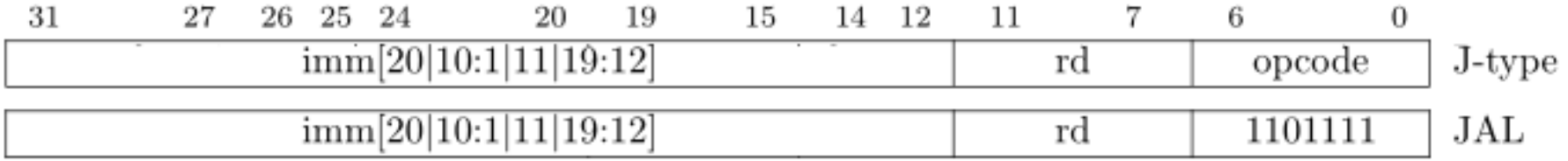
Store instructions (2)

- SB/SH/SW/SD stores 8/16/32/64-bit values from the low bits of register rs2 to memory

Misaligned accesses

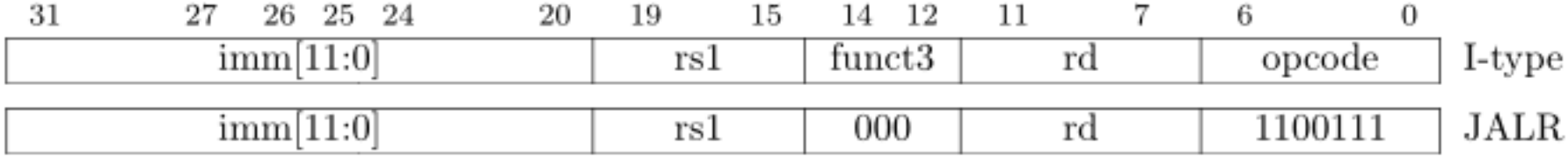
- misaligned loads and stores
  - are supported
  - might run extremely slowly
  - (*) not guaranteed to execute atomically
- instructions are 32-bit (4-byte) aligned in the base ISA
  - 16-bit (2-byte) aligned when instruction extensions with 16-bit lengths added
  - instruction address misaligned exceptions are generated

Jump instructions (1)



rd = pc + 4

pc = pc + *signext*(imm)



rd = pc + 4

pc = rs1 + *signext*(imm)

Jump instructions (2)

```
    …
    jal x1, func   # x1 = pc + 4,
                   # goto func
    …

func:
    …
    jalr x0, x1, 0   # return
                     # (forget pc)
```

Jump instructions (3)

- JAL: J-immediate encodes a signed offset in multiples of 2 bytes
- JALR: add the 12-bit signed I-immediate to the register rs1 then set the least-significant bit of the result to zero (i.e. ignore the lowest bit of the sum)
  - the result is again divisible by 2
  - avoids one more immediate format in hardware
  - simplifies the hardware slightly
  - potentially a slight loss of error checking (bad)
- instruction address misaligned exception is generated if the target address is not properly aligned

Branch instructions (1)

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011	BGEU

pc = pc + (rs1 *cmpop* rs2 ? *signext*(imm) : 4)

- EQ = equals, NE = not equal
- LT = less than, GE = greater than or equals to
- U = unsigned comparison

(*) Another popular alternative is to use status register (Program Status Word, FLAGS, or Condition Codes)

Branch instructions (2)

```
# do {
#       …
# }
# while( x12 < x13 );

loop:
      …
      bltu x12, x13, loop
```

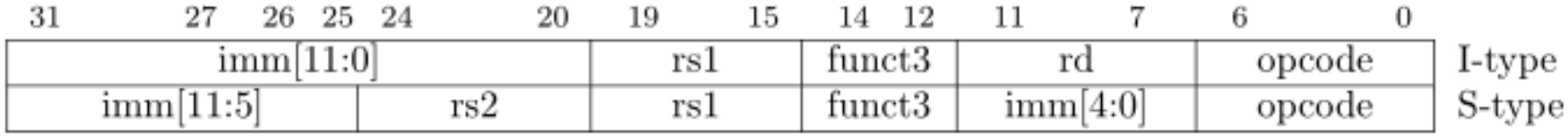
Branch instructions (3)

- use x0 for comparisons against zero
- rs1 <= rs2 ⇔ rs2 >= rs1 (BGE[U])
- rs1 > rs2 ⇔ rs2 < rs1 (BLT[U])

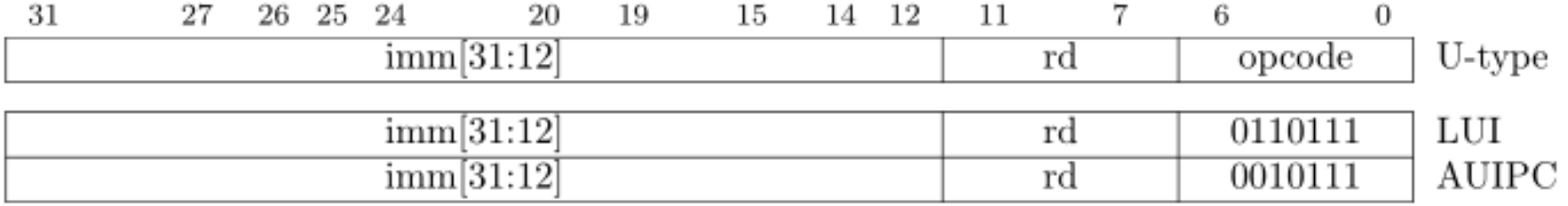
Branch instructions (4)

- B-immediate encodes a signed offset in multiples of 2 bytes
- instruction address misaligned exception is generated on a taken branch if the target address is not properly aligned

LUI and AUIPC instructions (1)



- 12-bit immediate fields
- 32 (RV32I) / 64 (RV64I) bit addresses



LUI: rd = *signext*(imm << 12)

AUIPC: rd = pc + *signext*(imm << 12)

(*) LUI and AUIPC instructions (2)

```
# x10 = 0x04c11db7

lui x10, 0x04c12
addi x10, x10, -585  # 0xfffffdb7

lui x10, %hi(0x04c11db7)
addi x10, x10, %lo(0x04c11db7)

li x10, 0x04c11db7 # pseudo-instruction
```

(*) LUI and AUIPC instructions (3)

```
# let the assembler (and the linker)
# calculate addresses
lui x10, %hi(data_label)
lw x11, %lo(data_label)(x10)

auipc x10, %pcrel_hi(data_label)
lw x11, %pcrel_lo(data_label)(x10)

lw x11, data_label # pseudo-instruction
sw x12, data_label # pseudo-instruction
```

(*) LUI and AUIPC instructions (4)

```
# let the assembler (and the linker)
# calculate addresses
auipc x10, %pcrel_hi(data_label)
addi x10, x10, %pcrel_lo(data_label)

la x10, data_label # pseudo-instruction
```

(*) LUI and AUIPC instructions (5)

```
# auipc+jalr pair
auipc x1, %pcrel_hi(target)
jalr x1, x1, %pcrel_lo(target)

call target  # pseudo-instruction

# Note: link-time optimization
```

Other base instructions (1)

- SLT[I][U]
  - SLT[U]: rd = (rs1 < rs2 ? 1 : 0) (R-type)
  - SLTI[U]: rd = (rs1 < *signext*(imm) ? 1 : 0) (I-type)

```
slt x10, x11, x12  # x10 = ( x11 < x12 )
```

Other base instructions (2)

- ECALL is used to make a request to the supporting execution environment, which is usually an OS
- EBREAK is used by debuggers

```
li x10, 17
li x11, 0
ecall
```

(*) Other base instructions (3)

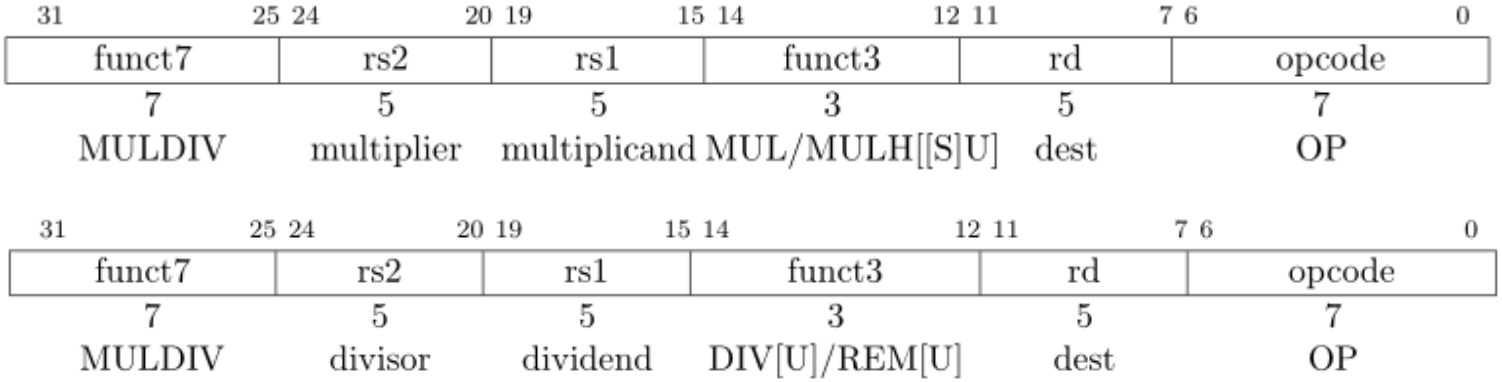
- FENCE is used to order device I/O and memory accesses
- FENCE.I is used to synchronize the instruction and data streams

(*) Other base instructions (4)

- CSRRW(I), CSRS(I), CSRC(I) atomically read-modify-write control and status registers
  - CSR = control and status registers
  - RW = atomic read and write
  - RC/RS = atomic read and clear/set bits
- CSRs
  - 3 mandatory 64-bit CSRs in RV32I and RV64I
    - cycle/wall-clock real time/instruction counters
  - No mandatory CSRs in RV32E

Integer multiplication and division

- using subroutines
- the “M” standard extension
  - additional instructions




```
mul x12, x10, x11    # x13:x12 = x10 * x11
mulh x13, x10, x11   #
```

(*) Floating point operations

- using subroutines
- the “F”/”D”/”Q” standard extension
  - 32 floating-point registers, each 32/64/128 bits wide (single/double/quad precision)
  - fcsr – floating-point control and status register
  - additional instructions
    - load/store
    - computational
    - conversion/move
    - compare
    - classify

(*) “C” Standard Extension for Compressed Instructions

- can be added to any of the base ISAs (RV32, RV64, RV128)
- reduces *static and dynamic* code size by adding short 16-bit instruction encodings for common operations
  - typically, 50%–60% of the RISC-V instructions in a program can be replaced, resulting in a 25%–30% code-size reduction
- freely intermixed with 32-bit instructions
- 8 compressed instructions formats
  - limited instruction set
  - the immediate is small
  - specific registers are used (x0, x1, x2, x8-x15, f8-f15)
  - rd and rs1 are identical (**2-operand** instructions)
- instructions are 2-byte aligned