Lecture 3

RISC-V Instruction Format

Recap

- Instruction set architecture
  - RISC vs. CISC
  - RISC-V/MIPS/ARM/x86
- Instructions:
  - Arithmetic instruction: add, sub, …
  - Data transfer instruction: lw, sw, lh, sh, …
  - Logical instruction: and, or, …
  - Conditional branch beq, bne, …
- Basic concepts:
  - Operands: register vs. memory vs. immediate
  - Numeric representation: signed, unsigned, sign extension

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- Conditional branch
  - *beq rs1, rs2, L1*

    if (rs1 == rs2) branch to instruction labeled L1;
  - *bne rs1, rs2, L1*
    - if (rs1 != rs2) branch to instruction labeled L1;
- Unconditional branch
  - *beq x0, x0, L1*
    - unconditional jump to instruction labeled L1

Labels in Assembly

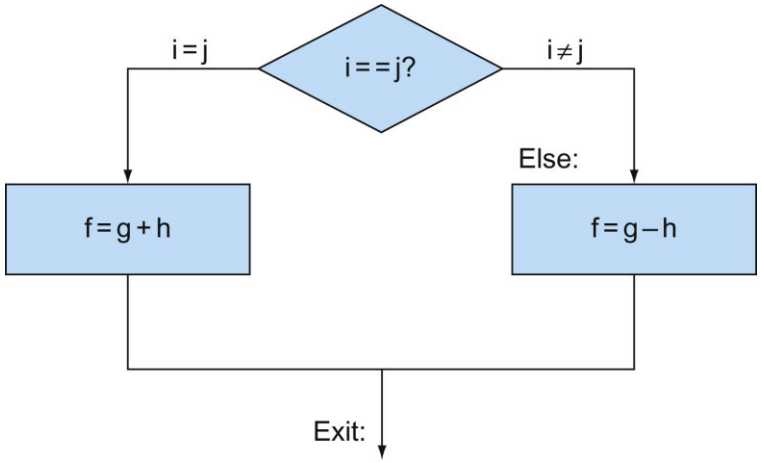
- We commonly see "labels" in the code
  - foo: add x2, x1, x0
- The assembler converts these into positions in the code
  - At what address in the code is that label …
- Labels give control flow instructions, such as jumps and branches, a place to go ...
  - e.g. bne x0, x2, foo
- The assembler in outputting the code does the necessary calculation so the jump or branch will go to the right place

Compiling If Statements

- C code

```
if (i==j) f = g+h;
else f = g-h;
```

  - i and j are in x22 and x23,
  - f,g and h are in x19, x20 and x2



- Compiled RISC-V code:

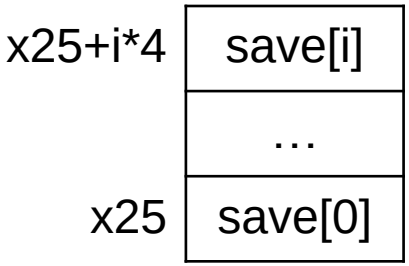
```
      bne x22, x23, Else  # go to Else if i ≠ j
      add x19, x20, x21   # f=g+h, skipped if i ≠ j
      beq x0, x0, Exit    # unconditional go to Exit
Else: sub x19, x20, x21   # f=g-h, skipped if i = j
Exit:
```

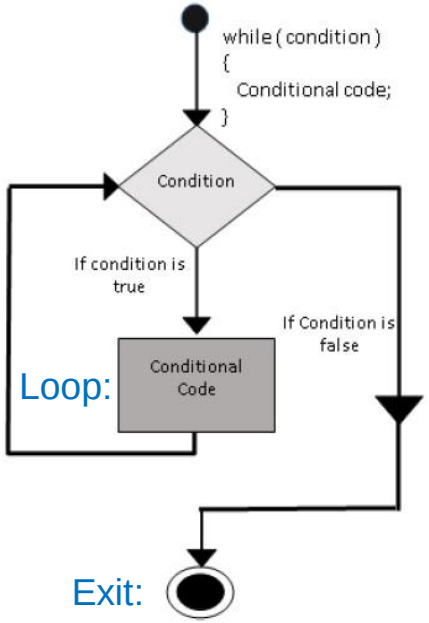
Compiling Loop Statements

- C code:

```
while (save[i] == k)
    i += 1;
```

  - i in x22, k in x24, address of save in x25





- Compiled MIPS code:

```
Loop: sll   x10, x22, 2     # Temp reg x10 = i * 4
      add   x10, x10, x25   # x10 = address of save[i]
      lw    x9, 0(x10)      # Temp reg x9 = save[i]
      bne   x9, x24, Exit   # go to Exit if save[i]≠k
      addi   x22, x22, 1    # i = i + 1
      j     Loop            # go to Loop
Exit:
```

More Conditional Operations

- Signed comparison
  - `blt rs1, rs2, L1`
    - if (rs1 < rs2) branch to instruction labeled L1
  - `bge rs1, rs2, L1`
    - if (rs1 >= rs2) branch to instruction labeled L1
  - Example, C to RISC-V
    - `if (a > b) a += 1;`
    - a in x22, b in x23

```
      bge  x23, x22, Exit   # signed comparison
      addi x22, x22, 1
Exit:
```

- Unsigned comparison
  - `bltu, bgeu`

What if we need more instructions?

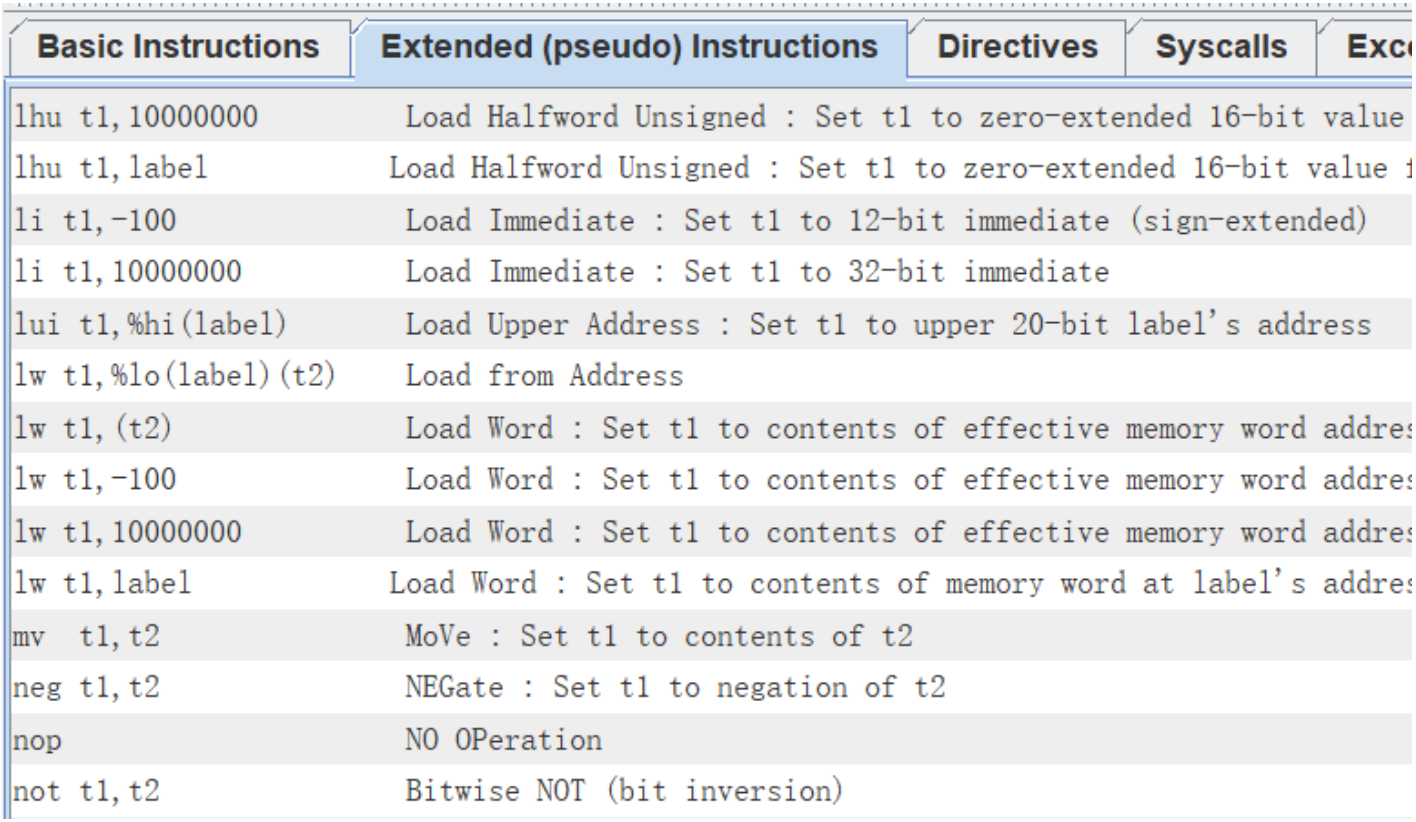
- RISC-V doesn't have "branch if greater than" or "branch if less than or equal"
- Instead you can reverse the arguments, as:
  - A > B is equivalent to B < A
  - A <= B is equivalent to B >= A
- The assembler defines pseudo-instructions for your convenience:

  `bgt x2, x3, foo (pseudo)` will become

  `blt x3, x2, foo (basic)`

Pseudo-instructions

- For more pseudo-instructions, refer to RARS Help (see in lab).

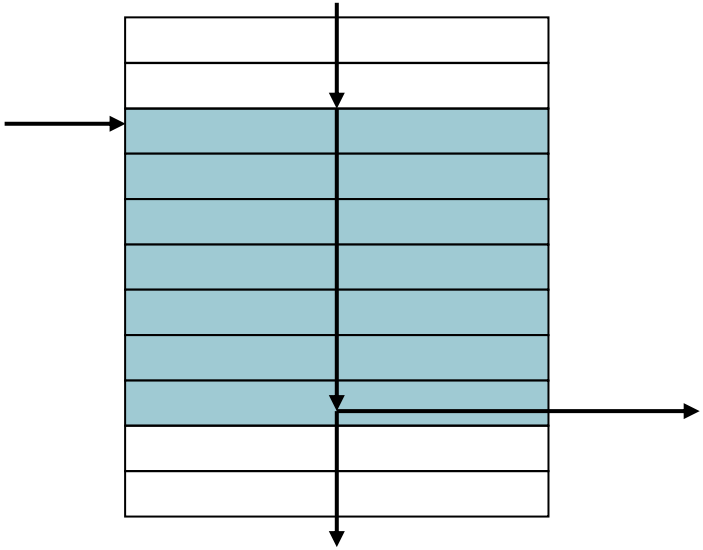


Basic Instructions | Extended (pseudo) Instructions | Directives | Syscalls | Exc

lhu t1,10000000	Load Halfword Unsigned : Set t1 to zero-extended 16-bit value
lhu t1,label	Load Halfword Unsigned : Set t1 to zero-extended 16-bit value f
li t1,-100	Load Immediate : Set t1 to 12-bit immediate (sign-extended)
li t1,10000000	Load Immediate : Set t1 to 32-bit immediate
lui t1,%hi(label)	Load Upper Address : Set t1 to upper 20-bit label's address
lw t1,%lo(label)(t2)	Load from Address
lw t1,(t2)	Load Word : Set t1 to contents of effective memory word addres
lw t1,-100	Load Word : Set t1 to contents of effective memory word addres
lw t1,10000000	Load Word : Set t1 to contents of effective memory word addres
lw t1,label	Load Word : Set t1 to contents of memory word at label's addres
mv t1,t2	MoVe : Set t1 to contents of t2
neg t1,t2	NEGate : Set t1 to negation of t2
nop	NO OPeration
not t1,t2	Bitwise NOT (bit inversion)

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



  - A compiler identifies basic blocks for optimization
  - An advanced processor can accelerate execution of basic blocks

C to RISC-V Example

```
int A[20];
int sum = 0;
for (int i=0; i<20; i++)
        sum += A[i];
```

Loop has 7 instructions

```
    # Assume x8 holds pointer to A
    # Assign x10=sum, x11=i
1   add x10, x0, x0                 # sum=0
2   add x11, x0, x0                 # i=0
3   addi x12,x0,20                  # x12=20
4   loop:
    bge x11, x12, exit
5   slli x13, x11, 2        # i * 4
6   add x13, x13, x8        # A + i
7   lw x13, 0(x13)                  # *(A +
8   i)
9   add x10, x10, x13       # increment sum
10  addi x11, x11, 1        # i++
11  beq x0, x0, loop        # iterate
    exit:
```

C to RISC-V Example Optimized

```
int A[20];
int sum = 0;
for (int i=0; i<20; i++)
        sum += A[i];
```

Loop now has 6 instructions

```
    # Assume x8 holds base address of A
    # Assign x10=sum, x11=i*4
1   add x10, x0, x0         # sum=0
2   add x11, x0, x0         # i=0
3   addi x12,x0,80          # x12=20*4
4   loop:
    bge x11, x12, exit
5   add x13, x11, x8        # A + i
6   lw x13, 0(x13)          # *(A + i)
7   add x10, x10, x13       # increment sum
8   addi x11, x11, 4        # i++
9   beq x0, x0, loop        # iterate
10  exit:
```

C to RISC-V Example Optimum

```
int A[20];
int sum = 0;
for (int i=0; i<20; i++)
        sum += A[i];
```

Loop now has 4 instructions

- Directly increment ptr into A array
- And only 1 branch/jump rather than two
  - Because first time through is always true so can move check to the end
  - The compiler will often do this automatically for optimization

```
    # Assume x8 holds base address of A
    # Assign x10=sum
    # Assume x11 holds ptr to next A
1   add x10, x0, x0         # sum=0
2   add x11, x0, x8         # Copy of A
3   addi x12, x8, 80        # x12=80 + A
4   loop:
    lw x13, 0(x11)
5   add x10, x10, x13
6   addi x11, x11, 4
7   blt x11, x12, loop
```

Summary of Design Principles

1: Simplicity favors regularity

- Keep all instructions the same size.

2: Smaller is faster

- Register vs memory
- Number of registers is small

3: Make the common case fast

- Immediate operand

4: Good design demands good compromises

- Keep formats as similar as possible

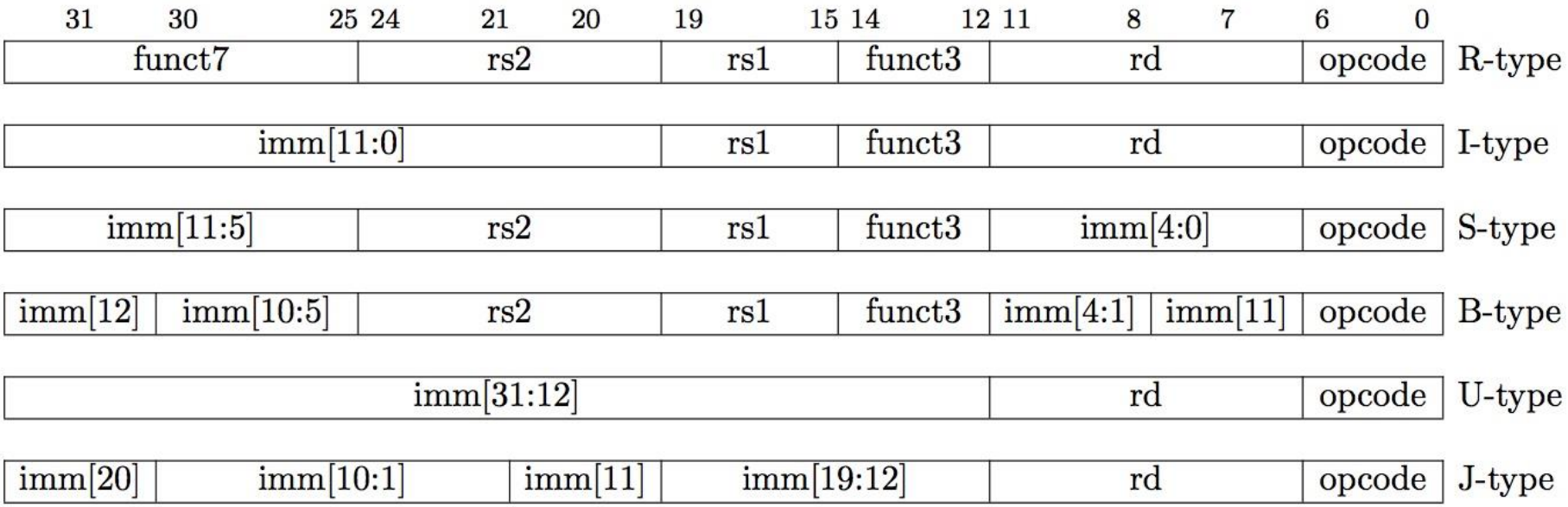
Instructions as Numbers

- Most data we work with is in words (32-bit chunks):
  - Each register holds a word
  - lw and sw both access memory one word at a time
- So how do we represent instructions?
  - Remember: Computer only represents 1s and 0s, so assembler string "`add x10,x11,x0`" is meaningless to hardware
  - RISC-V seeks simplicity: since data is in words, make instructions be fixed-size 32-bit words also
    - Same 32-bit instruction definitions used for RV32, RV64, RV128

Instructions in Binary

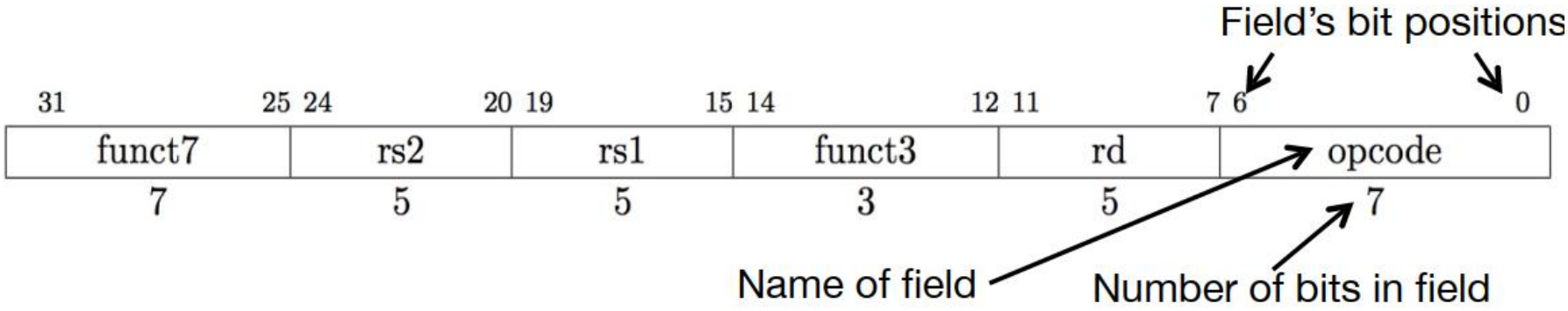
- Divide 32-bit instruction word into “fields”
- Each field tells processor something about instruction
- We could define different set of fields for each instruction, but for hardware simplicity, group possible instructions into six basic types of instruction formats:
  - **R-format** for register-register arithmetic/logical operations
  - **I-format** for register-immediate ALU operations and loads
  - **S-format** for stores
  - **B-format** for branches (SB in textbook)
  - **U-format** for 20-bit upper immediate instructions
  - **J-format** for jumps (UJ in textbook)

RISC-V Instruction Formats



R-Format Instructions

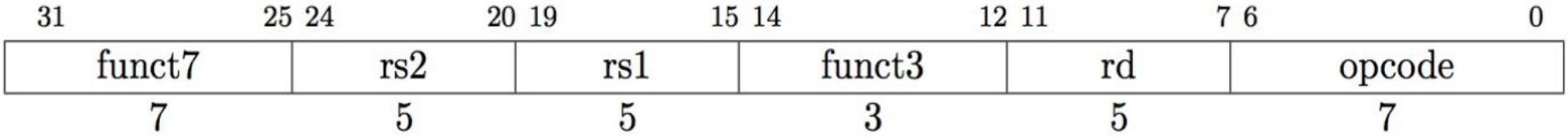
- Layout Annotation



- This example: 32-bit instruction word divided into six fields of differing numbers of bits each field: 7+5+5+3+5+7 = 32
- In this case:
  - opcode is a 7-bit field that lives in bits 0-6 of the instruction
  - rs2 is a 5-bit field that lives in bits 20-24 of the instruction

R-Format Instructions

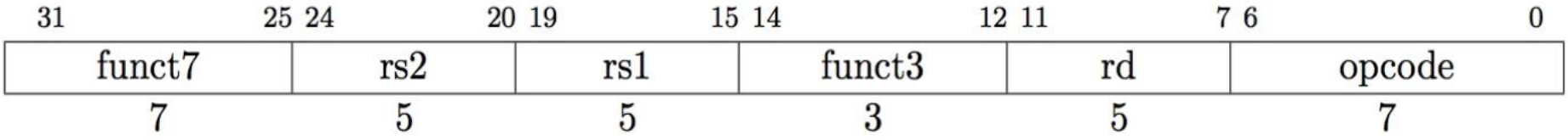
- opcode/funct fields



- opcode: partially specifies which instruction it is
  - Note: This field is contains 0110011two for all R-Format register-register arithmetic/logical instructions
- funct7+funct3: combined with opcode, these two fields describe what operation to perform
- Question: Why aren't opcode and funct7 and funct3 a single 17-bit field?
  - We'll answer this later

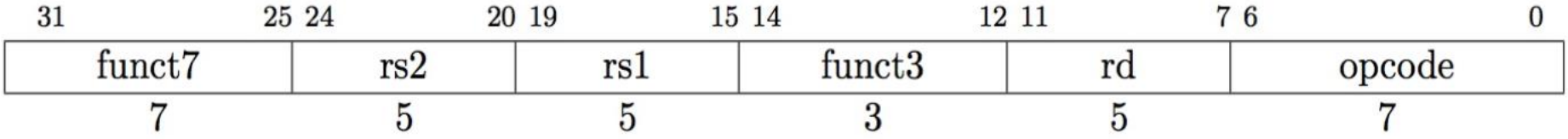
R-Format Instructions

- register specifiers



- Each register field (rs1, rs2, rd) holds a 5-bit unsigned integer [0-31] corresponding to a register number (x0-x31)
  - rs1 (Source Register #1): specifies register containing first operand
  - rs2 : specifies second register operand
  - rd (Destination Register): specifies register which will receive result of computation

R-Format Example



- Convert RISC-V Assembly to Machine Code:

```
add x18,x19,x10
```

0000000	01010	10011	000	10010	0110011
ADD	rs2=10	rs1=19	ADD	rd=18	Reg-Reg OP

- Exercise
  - `sub x10, x11, x12`
  - Machine code:
  - 0100000 01100 01011 000 01010 0110011
  - 0x40C58533

All RV32 R-format instructions

- All can be found in RISC-V reference card

funct7			funct3		opcode	
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

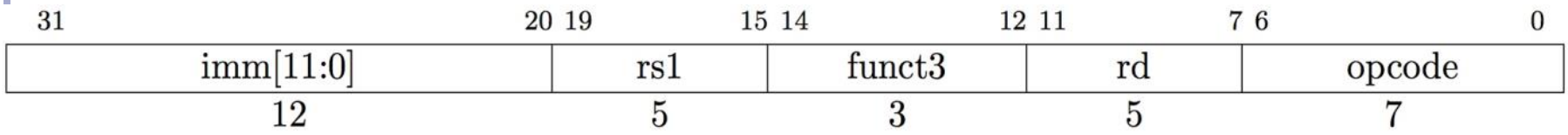
Encoding in funct7 + funct3 selects particular operation

I-Format Instructions

- What about instructions with immediates?
  - Ideally, RISC-V would have only one instruction format (for simplicity): unfortunately, we need to compromise
  - 5-bit field only represents numbers up to the value 31: would like immediates to be much larger
- Define another instruction format that is mostly consistent with R-format
  - Note: if instruction has immediate, then uses at most 2 registers (one source, one destination)

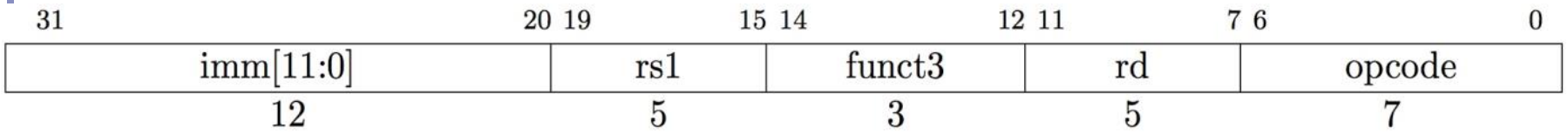
I-Format Instructions

- Layout Annotation



- Only one field is different from R-format, rs2 and funct7 replaced by 12-bit signed immediate, imm[11:0]
- Remaining field format (rs1, funct3, rd, opcode) same as before
- imm[11:0] can hold values in range [-2048_{ten} , $+2047_{ten}$]
- Immediate is always sign-extended to 32-bits before use in an arithmetic/logic operation
- We'll later see how to handle immediates > 12 bits

I-Format Instructions Example



- Convert RISC-V Assembly to Machine Code:

```
addi x15,x1,-50
```

111111001110	00001	000	01111	0010011
imm=-50	rs1=1	ADDI	rd=15	OP-Imm

```
slli x20, x8, 5
```

0000000 00101	01000	001	10100	0010011
imm = 0000000_shmnt(5)	rs1= 8	SLLI	rd=20	OP-Imm

All RV32 I-format Arithmetic/Logical Instructions

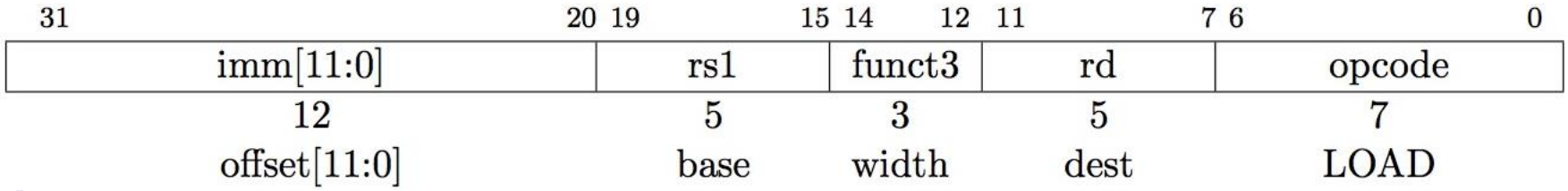
imm		funct3			opcode	
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

One of the higher-order immediate bits is used to distinguish “shift right logical” (SRLI) from “shift right arithmetic” (SRAI)

“Shift-by-immediate” instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)

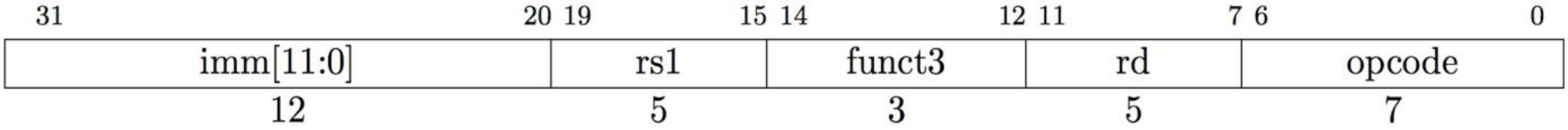
Load Instructions are also I-Type

rd = M[rs1+imm][0:31]



- The 12-bit signed immediate is added to the base address in register rs1 to form the memory address
  - This is very similar to the add-immediate operation but used to create address not to create final result
- The value loaded from memory is stored in register rd

I-Format Load Example



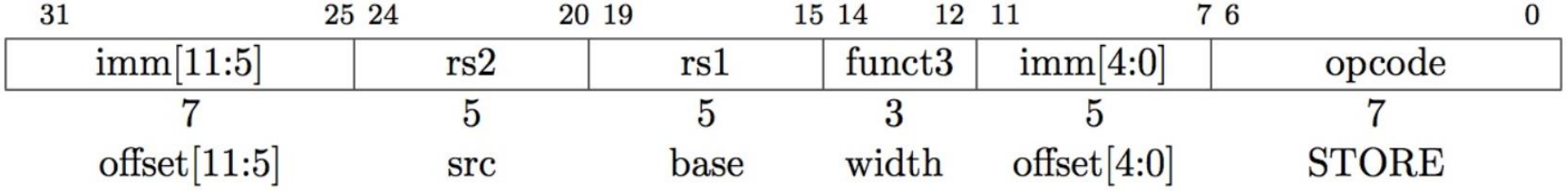
- Convert RISC-V Assembly to Machine Code:

```
lw x14, 8(x2)
```

000000001000	00010	010	01110	0000011
imm=8	rs1=2	LW	rd=14	LOAD

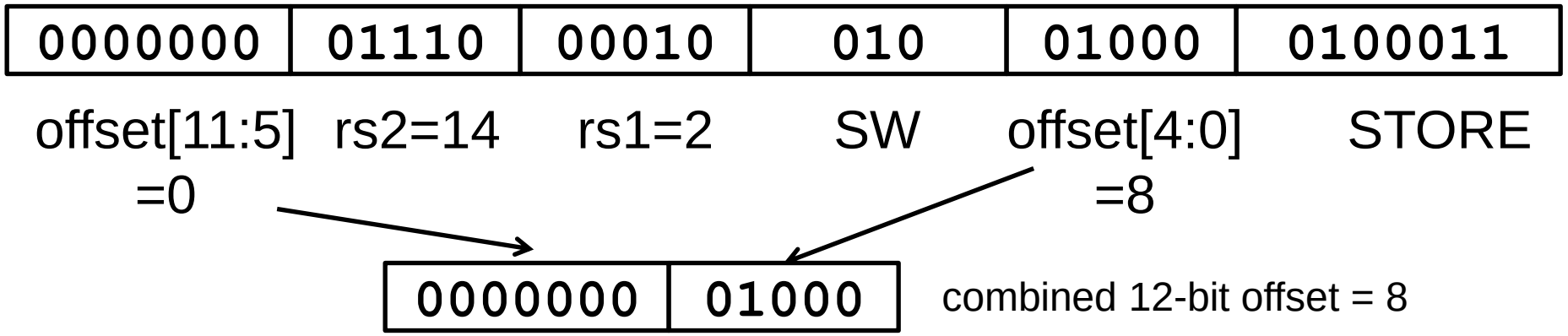
- Exercise
  - `lbu x6, 4(x5)`
  - Machine code:
  - 0000000 00100 00101 100 00110 0000011
  - 0x0042C303

S-Format Used for Stores



- Convert RISC-V Assembly to Machine Code:

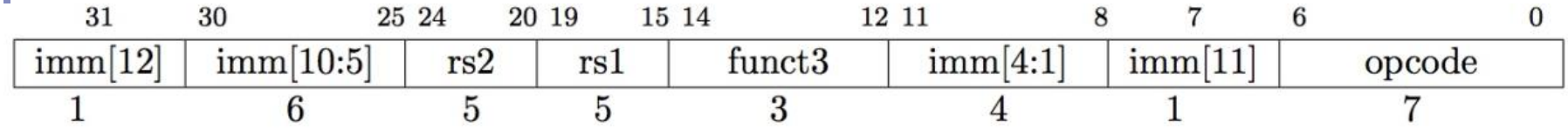
```
sw x14, 8(x2)
```



RISC-V Conditional Branches

- E.g., `BEQ x1, x2, Label`
- Branches read two registers but don't write a register (similar to stores)
- How to encode the label, i.e., where to branch to?
- We use an immediate to encode PC relative offset
  - If we **don't** take the branch:
  - PC = PC + 4 (i.e., next instruction)
  - If we **do** take the branch:
  - PC = PC + immediate

RISC-V B-Format for Branches

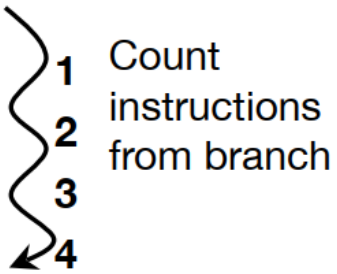


- B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate
- But now immediate represents the branch offset in units of half-words. To convert to units of Bytes, left-shift by 1.
- The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)
  - Thus the imm[12:1] in the total encoding, compared with imm[11:0] in the I-type encodings

Branch Example

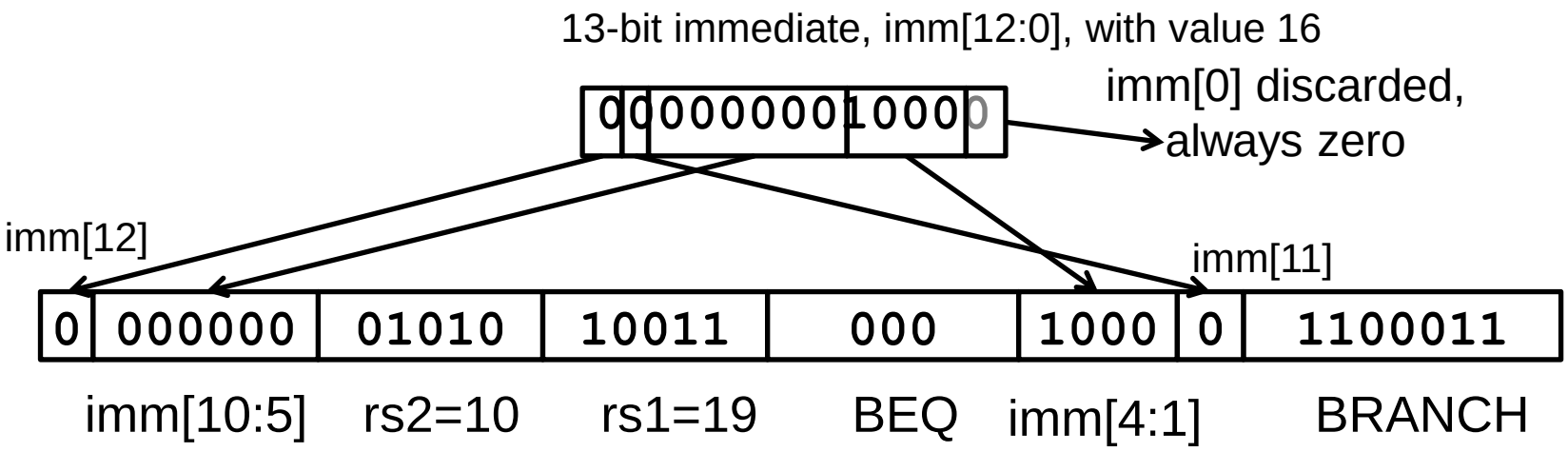
- RISC-V Assambly:

```
Loop: beq x19,x10,End
      add x18,x18,x
      addi x19,x19,
      j Loop
End:
```



- Branch offset = 4×32-bit instructions = 16 bytes = 8x2

beq x19,x10, offset = 16 bytes



PC-Relative Addressing

- PC-Relative Addressing: Use the immediate field as a two's complement offset relative to PC
  - Branches generally change the PC by a small amount
  - With the 12-bit immediate, could specify $\pm 2^{11}$ byte **address** offset from the PC
- Why not use byte address offset from PC as the immediate?
  - RISC-V uses 32-bit addresses, and memory is byte-addressed
  - 32-bit Instructions are "word-aligned": Address is always a multiple of 4 (in bytes)
  - PC ALWAYS points to an instruction
- However, extensions to RISC-V base ISA support 16-bit compressed instructions and also variable-length instructions that are multiples of 2-Bytes in length

RISC-V Feature, n×16-bit instructions

- To enable this, RISC-V always scales the branch immediate by 2 bytes - even when there are no 16-bit instructions
- This means for us
  - the low bit of the stored immediate value will always be 0)
  - The immediate is left-shifted by 1 before adding to PC
- RISC-V conditional branches can only reach $\pm 2^{10} \times 32$bit instructions either side of PC
- Thus we have:
- PC-relative addressing
  - **Target address = PC + immediate × 2**

Branching Far Away

- Does the value in branch immediate field change if we move the code?
  - If moving individual lines of code, then yes
  - If moving all of code, then no (because PC-relative offsets)
- What do we do if destination is > 2^{10} instructions away from branch?
  - replace conditional jump to unconditional jump

```
beq x10,x0,far
# next instr
```

```
        bne x10,x0,next
        j far
next: # next instr
```

32-bit Constants

- Most constants are small, 12-bit immediate is sufficient
- For the occasional 32-bit constant
  - LUI writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits.
  - Together with an ADDI to set low 12 bits, can create any 32-bit value in a register using two instructions (LUI/ADDI)
- Example, set 0x87654321

```
LUI x10, 0x87654       # x10 = 0x87654000
ADDI x10, x10, 0x321 # x10 = 0x87654321
```

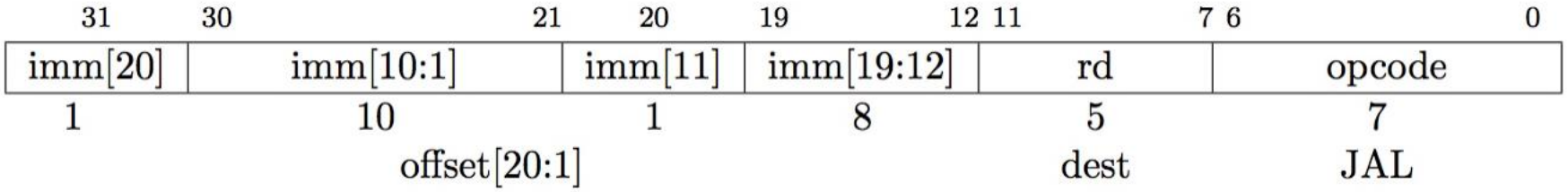
- set 0x0xDEADBEEF (ADDI 12-bit immediate is always sign-extended, if top bit is set, will subtract -1 from upper 20 bits)

```
LUI x10, 0xDEADC # x10 = 0xDEADB000
ADDI x10, x10, 0xEEF # x10 = 0xDEADBEEF
```

U-Format for “Upper Immediate”

- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, rd
- Used for two instructions
  - LUI – Load Upper Immediate, rd = imm << 12
  - AUIPC – Add Upper Immediate to PC, rd = PC + (imm << 12)

J-Format for Jump Instructions

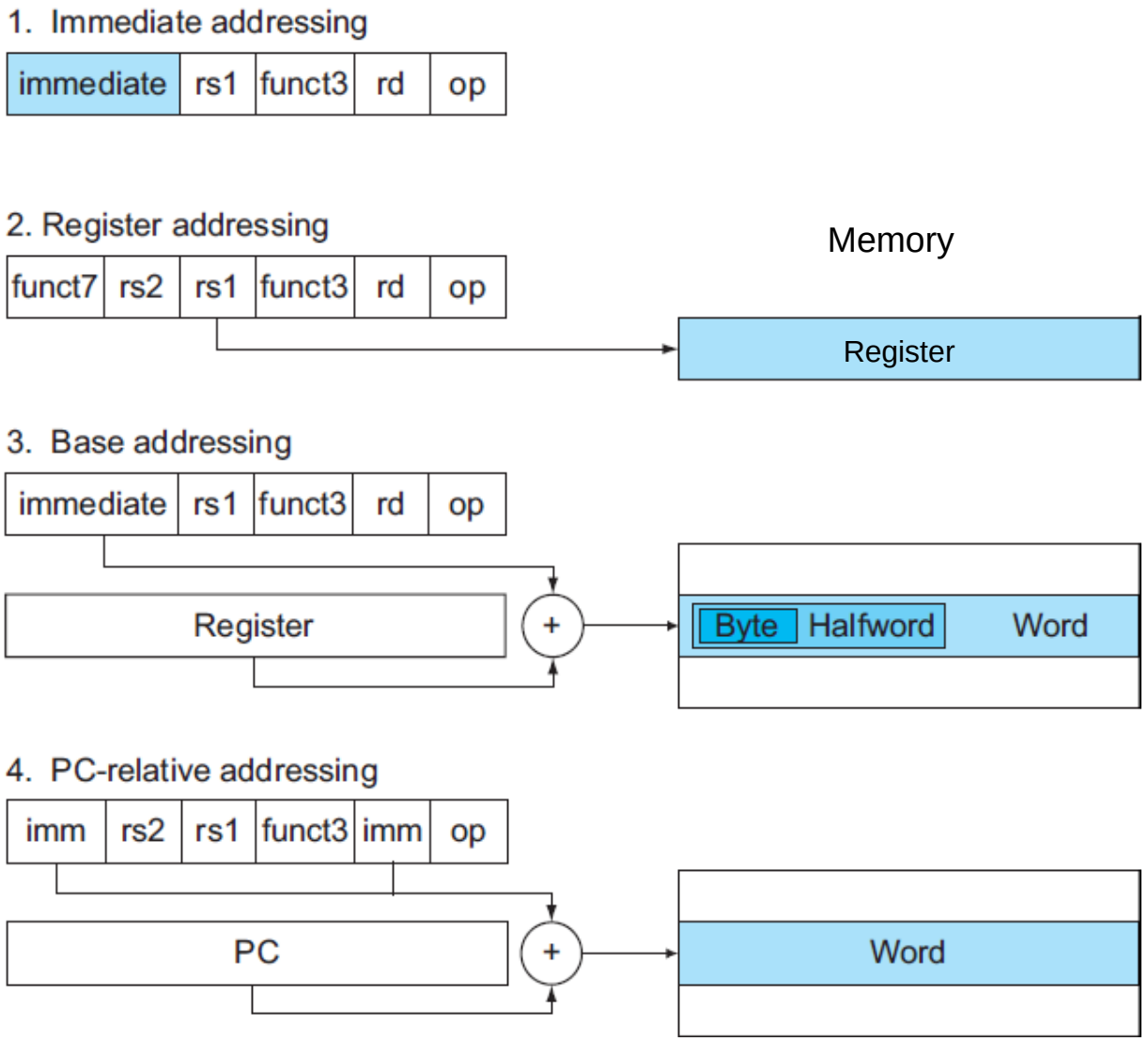


- JAL saves PC+4 in register rd (the return address)
  - Assembler "j" jump is pseudo-instruction, uses JAL but sets rd=x0 to discard return address
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
  - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

Long Jumps

- For long jumps, we use jalr
- eg, to 32-bit absolute address
  - lui: load address[31:12] to temp register
  - jalr: add address[11:0] and jump to target
- jalr is I-Format Instruction
- Example: `JALR rd, rs, immediate`
  - Writes PC+4 to rd (return address)
  - Sets PC = rs + immediate
  - Uses same immediates as arithmetic and loads
  - **no** multiplication by 2 bytes

RISC-V Addressing Summary



①

RISC-V Reference Data

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
add,addw	R	ADD (Word)	R[rd] = R[rs1] + R[rs2]	1)
addi,addiw	I	ADD Immediate (Word)	R[rd] = R[rs1] + imm	1)
and	R	AND	R[rd] = R[rs1] & R[rs2]	
andi	I	AND Immediate	R[rd] = R[rs1] & imm	
auipc	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	
beq	SB	Branch EQual	if(R[rs1]==R[rs2]) PC=PC+{imm,1b'0}	
bge	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2]) PC=PC+{imm,1b'0}	
bgeu	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2]) PC=PC+{imm,1b'0}	2)
blt	SB	Branch Less Than	if(R[rs1]<R[rs2]) PC=PC+{imm,1b'0}	
bltu	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2]) PC=PC+{imm,1b'0}	2)
bne	SB	Branch Not Equal	if(R[rs1]!=R[rs2]) PC=PC+{imm,1b'0}	
csrrc	I	Cont./Stat.RegRead&Clear	R[rd] = CSR;CSR = CSR & ~R[rs1]	
csrrci	I	Cont./Stat.RegRead&Clear Imm	R[rd] = CSR;CSR = CSR & ~imm	
csrrs	I	Cont./Stat.RegRead&Set	R[rd] = CSR; CSR = CSR | R[rs1]	
csrrsi	I	Cont./Stat.RegRead&Set Imm	R[rd] = CSR; CSR = CSR | imm	
csrrw	I	Cont./Stat.RegRead&Write	R[rd] = CSR; CSR = R[rs1]	
csrrwi	I	Cont./Stat.Reg Read&Write Imm	R[rd] = CSR; CSR = imm	
ebreak	I	Environment BREAK	Transfer control to debugger	
ecall	I	Environment CALL	Transfer control to operating system	
fence	I	Synch thread	Synchronizes threads	
fence.i	I	Synch Instr & Data	Synchronizes writes to instruction stream	
jal	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}	
jalr	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	3)
lb	I	Load Byte	R[rd] = {56'bM[]7],M[R[rs1]+imm][7:0]}	4)
lbu	I	Load Byte Unsigned	R[rd] = {56'b0,M[R[rs1]+imm][7:0]}	
ld	I	Load Doubleword	R[rd] = M[R[rs1]+imm][63:0]	
lh	I	Load Halfword	R[rd] = {48'bM[]15],M[R[rs1]+imm][15:0]}	4)
lhu	I	Load Halfword Unsigned	R[rd] = {48'b0,M[R[rs1]+imm][15:0]}	
lui	U	Load Upper Immediate	R[rd] = {32'bimm<31>, imm, 12'b0}	
lw	I	Load Word	R[rd] = {32'bM[]31],M[R[rs1]+imm][31:0]}	4)
lwu	I	Load Word Unsigned	R[rd] = {32'b0,M[R[rs1]+imm][31:0]}	
or	R	OR	R[rd] = R[rs1] | R[rs2]	
ori	I	OR Immediate	R[rd] = R[rs1] | imm	
sb	S	Store Byte	M[R[rs1]+imm][7:0] = R[rs2][7:0]	
sd	S	Store Doubleword	M[R[rs1]+imm][63:0] = R[rs2][63:0]	
sh	S	Store Halfword	M[R[rs1]+imm][15:0] = R[rs2][15:0]	
sll,sllw	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
slli,slliw	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	2)
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	2)
sra,sraw	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
srai,sraiw	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
srl,srlw	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
srli,srliw	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)
sub,subw	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	1)
sw	S	Store Word	M[R[rs1]+imm][31:0] = R[rs2][31:0]	
xor	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
xori	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

Notes: 1) The Word version only operates on the rightmost 32 bits of a 64-bit registers
2) Operation assumes unsigned integers (instead of 2's complement)
3) The least significant bit of the branch address in jalr is set to 0
4) (s)mask Load instructions extend the sign bit of data to fill the 64-bit register
5) Replicates the sign bit to fill in the leftmost bits of the result during right shift
6) Multiply with one operand signed and one unsigned
7) The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register
8) Classify writes a 10-bit mask to show which properties are true (e.g., -inf, -8, +0, +inf, denorm, ...)
The immediate field is sign-extended in RISC-V

②

ARITHMETIC CORE INSTRUCTION SET

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
mul,mulw	R	MULtiply (Word)	R[rd] = (R[rs1] * R[rs2])[63:0]	1)
mulh	R	MULtiply upper Half	R[rd] = (R[rs1] * R[rs2])[127:64]	
mulhsu	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])[127:64]	6)
mulhu	R	MULtiply upper Half Unsigned	R[rd] = (R[rs1] * R[rs2])[127:64]	2)
div,divw	R	DIVide (Word)	R[rd] = (R[rs1] / R[rs2])	1)
divu	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
rem,remw	R	REMainder (Word)	R[rd] = (R[rs1] % R[rs2])	1)
remu,remuw	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])	1,2)

RV64F and RV64D Floating-Point Extensions

flw,fld	I	Load (Word)	F[rd] = M[R[rs1]+imm]	1)
fsw,fsd	S	Store (Word)	M[R[rs1]+imm] = F[rd]	1)
fadd.s,fadd.d	R	ADD	F[rd] = F[rs1] + F[rs2]	7)
fsub.s,fsub.d	R	SUBtract	F[rd] = F[rs1] – F[rs2]	7)
fmul.s,fmul.d	R	MULtiply	F[rd] = F[rs1] * F[rs2]	7)
fdiv.s,fdiv.d	R	DIVide	F[rd] = F[rs1] / F[rs2]	7)
fsqrt.s,fsqrt.d	R	SQuare RooT	F[rd] = sqrt(F[rs1])	7)
fmadd.s,fmadd.d	R	Multiply-ADD	F[rd] = F[rs1] * F[rs2] + F[rs3]	7)
fmsub.s,fmsub.d	R	Multiply-SUBtract	F[rd] = F[rs1] * F[rs2] – F[rs3]	7)
fnmsub.s,fnmsub.d	R	Negative Multiply-SUBtract	F[rd] = –(F[rs1] * F[rs2] – F[rs3])	7)
fnmadd.s,fnmadd.d	R	Negative Multiply-ADD	F[rd] = –(F[rs1] * F[rs2] + F[rs3])	7)
fsgnj.s,fsgnj.d	R	SiGN source	F[rd] = {F[rs2][-63>,F[rs1][-62:0>}	7)
fsgnjn.s,fsgnjn.d	R	Negative SiGN source	F[rd] = {(~F[rs2][-63>,F[rs1][-62:0>}	7)
fsgnjx.s,fsgnjx.d	R	Xor SiGN source	F[rd] = {F[rs2][-63>^F[rs1][-63>, F[rs1][-62:0>}	7)
fmin.s,fmin.d	R	MINimum	F[rd] = (F[rs1] < F[rs2]) ? F[rs1] : F[rs2]	7)
fmax.s,fmax.d	R	MAXimum	F[rd] = (F[rs1] > F[rs2]) ? F[rs1] : F[rs2]	7)
feq.s,feq.d	R	Compare Float EQual	R[rd] = (F[rs1] == F[rs2]) ? 1 : 0	7)
flt.s,flt.d	R	Compare Float Less Than	R[rd] = (F[rs1] < F[rs2]) ? 1 : 0	7)
fle.s,fle.d	R	Compare Float Less than or =	R[rd] = (F[rs1] <= F[rs2]) ? 1 : 0	7)
fclass.s,fclass.d	R	Classify Type	R[rd] = class(F[rs1])	7,8)
fmv.s.x,fmv.d.x	R	Move from Integer	F[rd] = R[rs1]	7)
fmv.x.s,fmv.x.d	R	Move to Integer	R[rd] = F[rs1]	7)
fcvt.s.d	R	Convert from DP to SP	F[rd] = single(F[rs1])	
fcvt.d.s	R	Convert from SP to DP	F[rd] = double(F[rs1])	
fcvt.s.w,fcvt.d.w	R	Convert from 32b Integer	F[rd] = float(R[rs1][31:0])	7)
fcvt.s.l,fcvt.d.l	R	Convert from 64b Integer	F[rd] = float(R[rs1][63:0])	7)
fcvt.s.wu,fcvt.d.wu	R	Convert from 32b Int Unsigned	F[rd] = float(R[rs1][31:0])	2,7)
fcvt.s.lu,fcvt.d.lu	R	Convert from 64b Int Unsigned	F[rd] = float(R[rs1][63:0])	2,7)
fcvt.w.s,fcvt.w.d	R	Convert to 32b Integer	R[rd][31:0] = integer(F[rs1])	7)
fcvt.l.s,fcvt.l.d	R	Convert to 64b Integer	R[rd][63:0] = integer(F[rs1])	7)
fcvt.wu.s,fcvt.wu.d	R	Convert to 32b Int Unsigned	R[rd][31:0] = integer(F[rs1])	2,7)
fcvt.lu.s,fcvt.lu.d	R	Convert to 64b Int Unsigned	R[rd][63:0] = integer(F[rs1])	2,7)

CORE INSTRUCTION FORMATS

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

PSEUDO INSTRUCTIONS

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs]!=0) PC=PC+{imm,1b'0}	bne
fabs.s,fabs.d	Absolute Value	F[rd] = (F[rs]<0) ? -F[rs] : F[rs]	fsgnx
fmv.s,fmv.d	FP Move	F[rd] = F[rs]	fsgnj
fneg.s,fneg.d	FP negate	F[rd] = -F[rs]	fsgnjn
j	Jump	PC = {imm,1b'0}	jal
jr	Jump register	PC = R[rs]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs]	addi
neg	Negate	R[rd] = -R[rs]	sub
nop	No-operation	R[0] = R[0]	addi
not	Not	R[rd] = ~R[rs]	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	R[rd] = (R[rs]== 0) ? 1 : 0	sltiu
snez	Set ≠ zero	R[rd] = (R[rs]!= 0) ? 1 : 0	sltu