NYU TANDON SCHOOL OF ENGINEERING

Lecture 2

Introduction to the RISC-V ISA

Instructor: **Azeez Bhavnagarwala**

ECE 6913, Fall 2025

Computer Systems Architecture

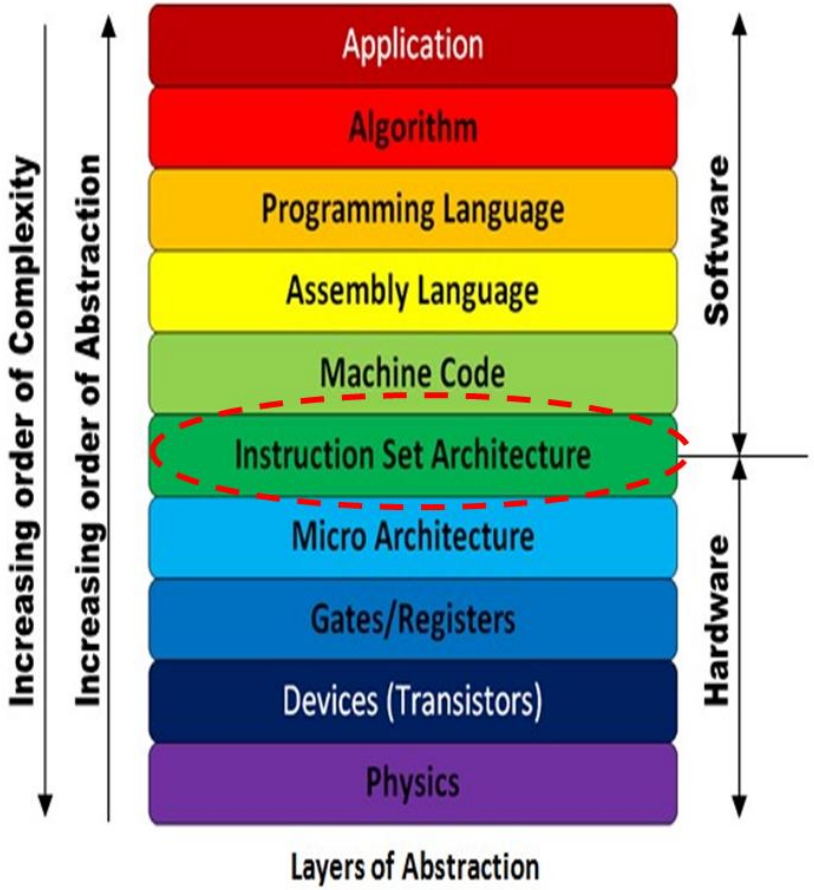
NYU Tandon School of Engineering

Computer Instructions

- Words of a Computer's language are called '**Instructions**'
  - Add 2 numbers, load data from memory, store data in memory, multiply or divide numbers, shift binary strings, other logic/arithmetic operations..
- Vocabulary is an '**Instruction Set**'
- Different computers have **different** Instruction sets – with much in common
  - ARMv7 in **most** mobile devices
  - Intel x86 instructions in **most** PCs and Datacenter Server CPUs
- Market evolution dynamics – now shaped by cost, energy efficiency and AI applications are **demanding different architectures**
- Easily the **most exciting** and **opportunity riddled** period in the History of Computers

What is an Instruction Set Architecture?

- ISA serves as the **interface** between software and hardware layers of abstraction
- Software that has been written for an ISA **can run on different implementations** of the same ISA
  - For x86 CPUs used in PCs, Servers etc. sold by Intel or AMD
- Enables **binary compatibility** of Applications between different CPUs using the same ISA to be easily achieved
- An ISA defines everything a machine language programmer needs to know in order to program a computer.



What purpose does the ISA serve?

- An Instruction Set Architecture (ISA) defines, describes, and specifies how a particular computer processor **executes in hardware the intent of the programmer.**
- The ISA describes each **machine-level instruction** and specifies exactly what each instruction does and **how it is encoded into bits**
- An ISA specification tells **Hardware engineers** what digital circuits to design to implement a given instruction.
- **Software engineers** write code (operating systems, compilers, etc.) based on a given ISA specification
- ISA attributes of **simplicity, regularity** enable higher *instruction execution rates*, higher *energy efficiencies* for given workloads enabling product differentiation across a wide spectrum of applications

Legacy ISAs

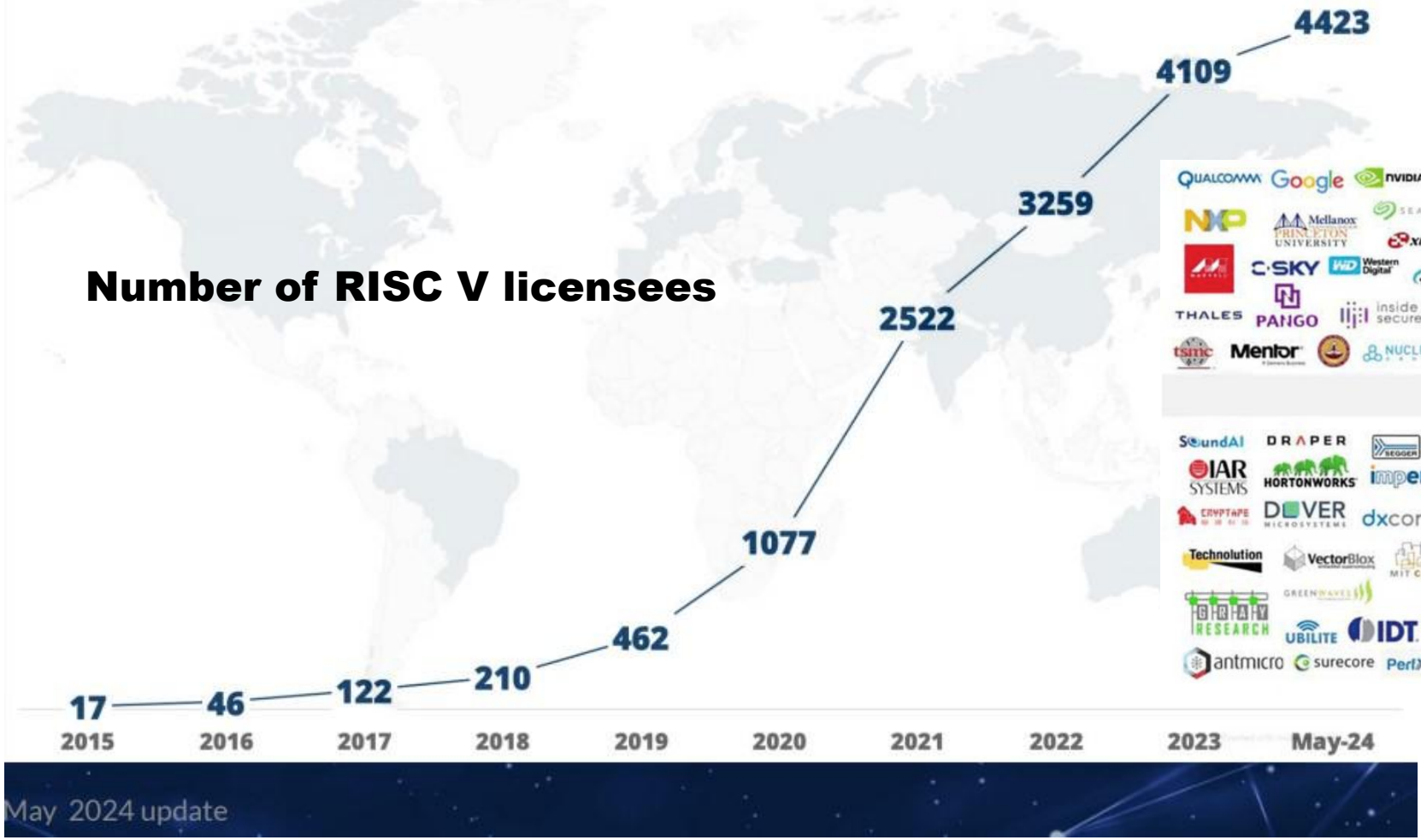
- There are a number of ISAs in widespread use, for example:
  - x86-64 (AMD, Intel)
  - ARM (ARM Holdings)
  - SPARC (Sun/Oracle)
- Each of these ISAs are **proprietary**
- Some of the details of the ISA are **not made public** at all.
- These widely used ISAs have **been around for years** and their designs carry baggage as a result, e.g., for backward compatibility.
- Since these legacy designs were first created, industry has learned (from mistakes and from emerging market opportunities) more about the **design of an ISA**.
- Changes in hardware technology **also have an impact** on which design choices are now optimal.

RISC V Instruction Set

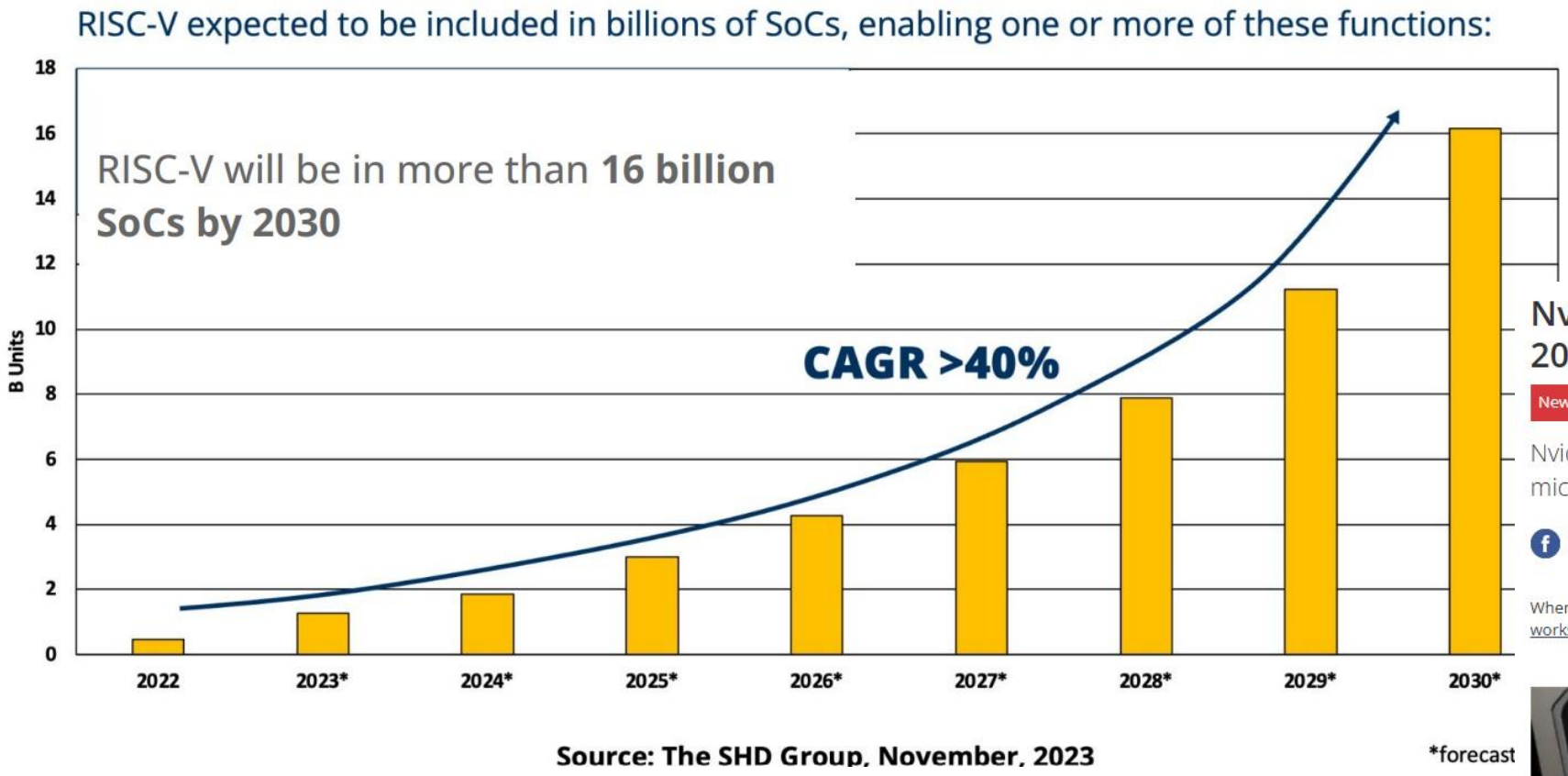
- A pure Reduced Instruction Set (RISC) architecture - execute **one instruction per clock** cycle
- An **open-source** ISA - many different companies provide hardware implementations of the RISC-V ISA creating an ecosystem in which multiple vendors can compete – **without the ISA licensing fees ($Ms)** that limit entry of potential hardware vendors.
- An ISA can serve **multiple applications/markets, multiple design constraints** – from a cheap, reliable 16 bit microcontroller for a dishwasher to a high performance, 64 bit multi-core processor for a server
- The **RISC V specification** is more **a 'menu'** - from which a particular implementation will choose some items, but not others – for e.g., the width (32/64/128b) or number (16/32) of registers, length of instructions (32b/16b[optional]), fl. point support, multiply/divide hardware support etc.

RISC V adoption industry-wide

Number of RISC V licensees



RISCV Volume of Cores Shipped Annually



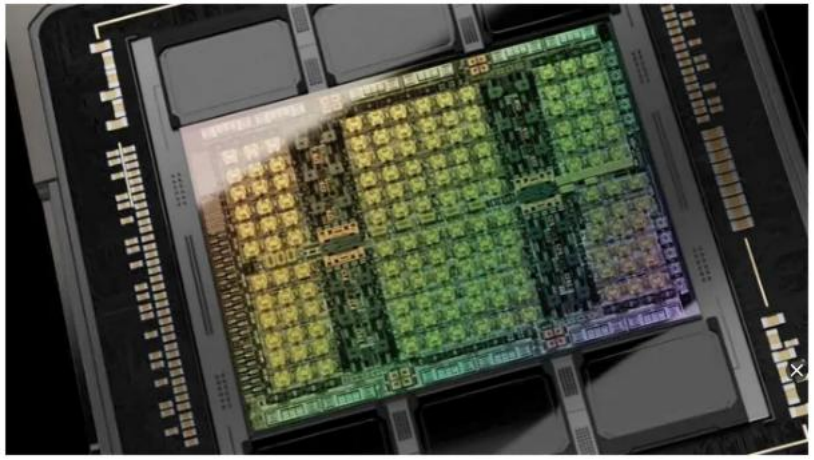
Nvidia to ship a billion of RISC-V cores in 2024

News By Anton Shilov published October 24, 2024

Nvidia quietly adopts RISC-V and replaces proprietary microcontrollers.

Comments (8)

When you purchase through links on our site, we may earn an affiliate commission. Here's how it works.



RISC V Naming Conventions

- The RISC-V specification is not a single ISA. Instead, it is a collection of ISA options.
- A naming convention is used in which a particular ISA variation is given a coded name telling which ISA options are present and supported.
- A particular hardware (chip) can be described or summarized with such a coded name, indicating which RISC-V features are implemented by the chip

RISC V Naming Conventions

- Consider the following RISC-V name:
  - RV32IMAFD
- The “RV” stands for “RISC-V” - all coded names begin with “RV”.
- The “32” indicates that registers are 32 bits wide. Other options are 64 bits and 128 bits:
  - RV32 32-bit machines
  - RV64 64-bit machines
  - RV128 128-bit machines

RISC V Naming Conventions

- The remaining letters have these meanings:
  - I – Basic integer arithmetic is supported
  - M – Multiply and divide are supported in hardware
  - A – The instructions implementing atomic synchronization are supported
  - F – Single precision (32 bit) 2loating point is supported
  - D – Double precision (64 bit) 2loating point is supported
- Each of these is considered to be an “extension” of the base ISA, except for the “I” (basic integer instructions), which is always required.
- The letter “G” is used as an abbreviation for “IMAFD”:
- RV32G = RV32IMAFD

Basic Terminology

- Prefix notation

Prefix		Example	Value		
Ki	kibi	KiByte	2^{10}	1,024	$\sim 10^3$
Mi	mebi	MiByte	2^{20}	1,048,576	$\sim 10^6$
Gi	gibi	GiByte	2^{30}	1,073,741,824	$\sim 10^9$
Ti	tebi	TiByte	2^{40}	1,099,511,627,776	$\sim 10^{12}$
Pi	pebi	PiByte	2^{50}	1,125,899,906,842,624	$\sim 10^{15}$
Ei	exbi	EiByte	2^{60}	1,152,921,504,606,846,976	$\sim 10^{18}$

	number of bytes	number of bits	example value (in hex)
byte	1	8	A4
halfword	2	16	C4F9
word	4	32	AB12CD34
doubleword	8	64	01234567 89ABCDEF
quadword	16	128	4B6D073A 9A145E40 35D0F241 DE849F03

Basic Terminology

- Notation to represent a range of bits:

Example	Meaning
[7:0]	Bits 0 through 7; e.g., all bits in a byte
[31:0]	Bits 0 through 31; e.g., all bits in a word
[31:28]	Bits 28 through 31; e.g., the upper 4 bits in a word
[1:0]	Bits 0 through 1; e.g., the least significant 2 bits

address (in hex)	data (in hex)
00000000	89
00000001	AB
00000002	CD
00000003	EF
00000004	01
00000005	23
00000006	45
00000007	67
...	...
FFFFFFFC	E0
FFFFFFFD	E1
FFFFFFFE	E2
FFFFFFFF	E3

- Basic Organization of data
- "Low" memory refers to smaller memory addresses, which will be shown higher on the page than "high" memory addresses, as in the example on the right lower corner of this slide:
- Hex Table:

Binary	Hex	Binary	Hex
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Arithmetic Instructions

- Every computer must perform arithmetic
- RISCV requires exactly 3 operands – enables simplicity in the hardware
- Variable number of operands (older CISC ISAs) leads to substantially more complex hardware
- Operands for Arithmetic instructions are restricted – must only be from a limited number of special locations: Registers
- Primitives used in hardware design also visible to the programmer

Arithmetic Operations

- Add and subtract, three operands
  - **Two** sources and **one** destination

```
add a, b, c   # a gets b + c
add a, a, d   # b+c+d now in a
add a, a, e   # a gets b+c+d+e
```

- All arithmetic operations have this form
- It took 3 instructions to sum 4 variables
- Design Principle 1: *Simplicity favors regularity*
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

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

- Translation from c to RISCV performed by compiler
- Compiled MIPS code:

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

- Comments always terminate on the same line in Assembly

Register Operands

- Arithmetic instructions use **register operands**
- RISCV has a **32 or 64 or 128 × 32-bit register file**
  - Use for *frequently accessed data*
  - Numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - x5-x7, x28-x31 for *temporary values*
  - x9, x18-x27 for *saved variables*
- Design Principle 2: Smaller is faster
  - c.f. main memory: millions of locations

Registers Vs Memory

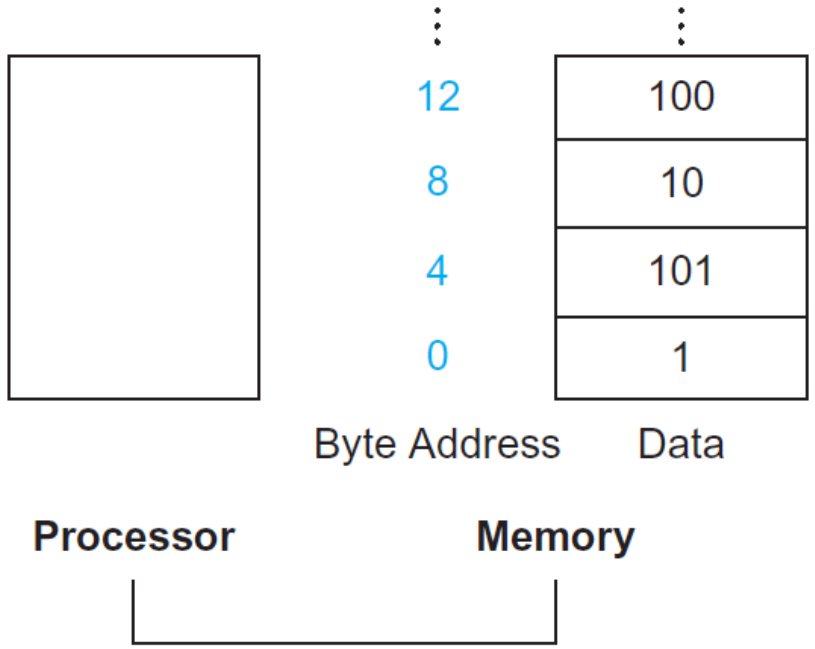
- Compilers associate variables with Registers – so they can be accessed and operated on within a cycle.
- Array data structures are stored in Memory and can only be accessed to read or write during a cycle
- Registers are *faster* to access *than memory* – they are much smaller (shorter wires, smaller device capacitances)
- Operating on memory data requires loads and stores
  - *More cycles* to be executed for a Read/Write
- Compiler must *use registers* for variables as much as possible – complete execution in 1 cycle!
  - Only spill to memory for less frequently used variables
  - Register optimization is important! There are only 32 of them!

RISC V Registers

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

Accessing Memory

- Instruction must supply the Memory Address
- Memory is a large, single-dimensional Array
- Addresses act as index starting at 0
- For e.g., Address of the third data element is ‘8’
- Value of Memory [8] is ‘10’



- **Note: ISAs usually use ‘Byte addressing’ with each Word representing 4 bytes.**
- **So each word address in memory increments by 4**

Memory Operands

- Main memory used for composite data
  - Integer, Floating Point, Character, Boolean, Arrays, Lists etc.
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Double words (64b) must start addresses that are multiples of 8 (RISCV, Intel do not have this restriction)
- RISCV is Little Endian
  - Little Endian: least-significant byte at least address

Which Byte does an Address identify?

- 8 Bytes contribute to defining a 64b address, but which byte?
- Use the address of the leftmost or “big end” byte as the doubleword address [Big Endian] – Intel… or
- The rightmost or “little end” byte [Little Endian] – RISCV, ARM
- The order matters only if you access the identical data both as a doubleword and as eight individual bytes

Memory Operand Example 1

- C code: (assume 32 bit registers)

```
g = h + A[8];
```

- g in x1, h in x2, base address of A in x3
- Compiled RISCV code:
- Index 8 requires offset of 32
- 4 bytes per word

```
lw   x8, 32(x3)     # load word
add  x1, x2, x8
```

Offset in bytes

Base Register

Memory Operand Example 2

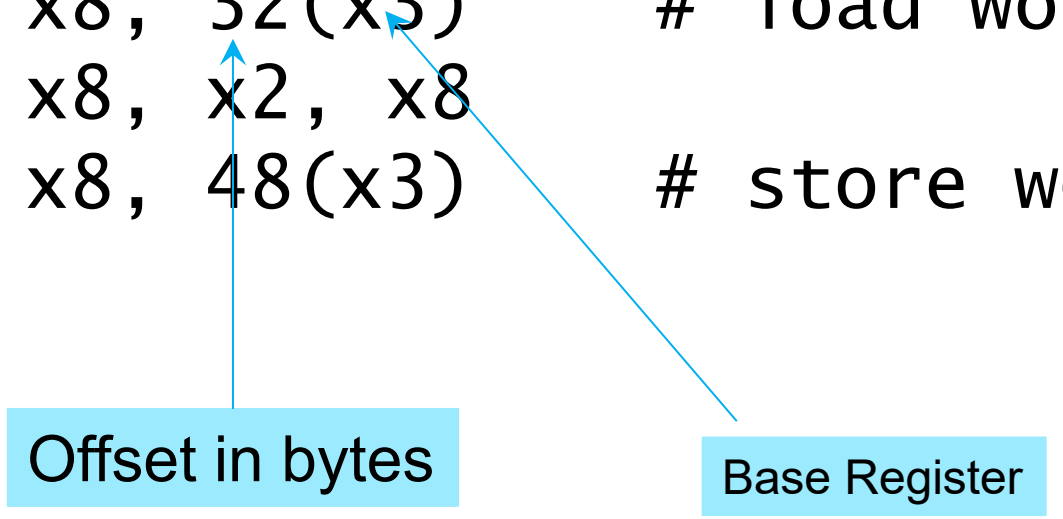
- C code:

```
A[12] = h + A[8];
```

  - h in x2, base address of A in x3

- Compiled RISCV code:
  - Index 8 requires offset of 32

```
lw  x8, 32(x3)    # load word
add x8, x2, x8
sw  x8, 48(x3)    # store word
```



Immediate Operands

- Constant data specified in an instruction

```
addi x22, x22, 4
```

  - add the number ‘4’ to register x22 and place result in register x22

- No subtract immediate instruction
  - Just use a negative constant

```
addi x22, x22, -1
```

- Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Examples

- 1. For the following C statement, what is the corresponding RISCV assembly code?

```
f = g + (h - 5); #h in x1, g in x2, f in x3
addi x22, x1, -5
add x3, x2, x22
```

- 2. For the following RISCV assembly instructions, what is a corresponding C statement?

```
add f, g, h
add f, i, f
f = g + h + i;
```

The Constant ‘0’

- RISCV register x0 is the constant 0
  - Cannot be overwritten
  - Each bit wired to GND in hardware
- Useful for common operations
  - E.g., move between registers

```
add x2, x1, x0
```

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ \ldots\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 64 bits: $-9{,}223{,}372{,}036{,}854{,}775{,}808$
  to $9{,}223{,}372{,}036{,}854{,}775{,}807$

Negation using 2's complement

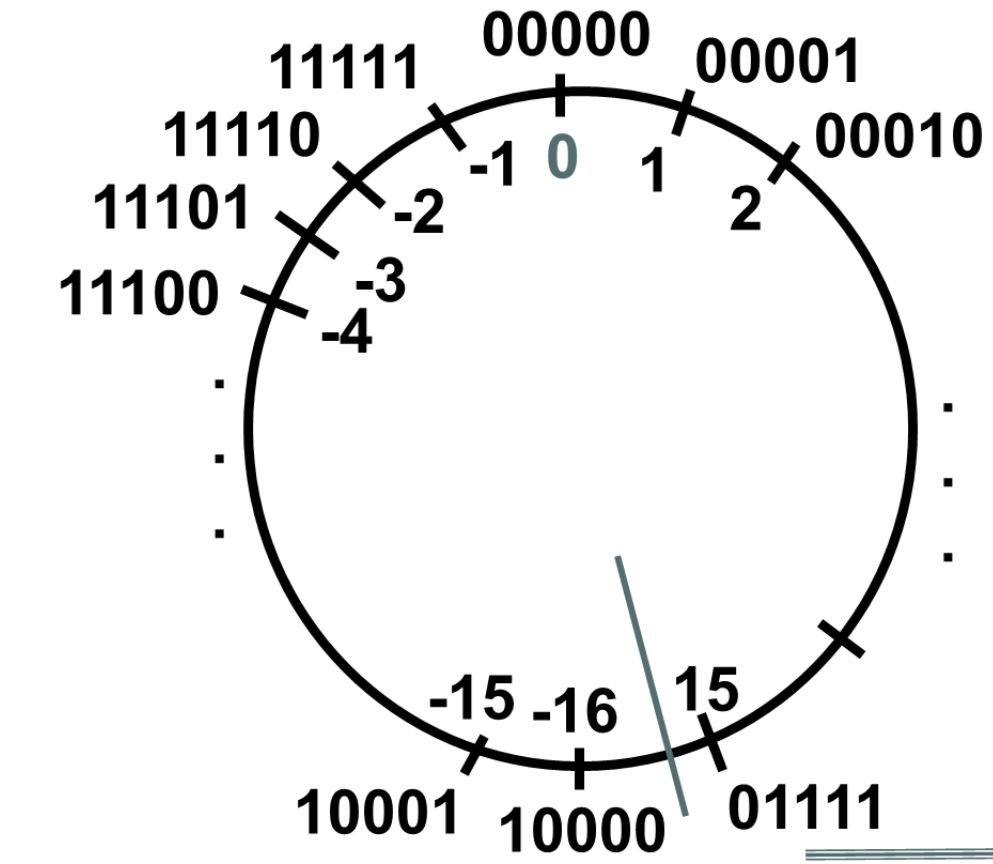
- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111\ldots111_2 = -1$$

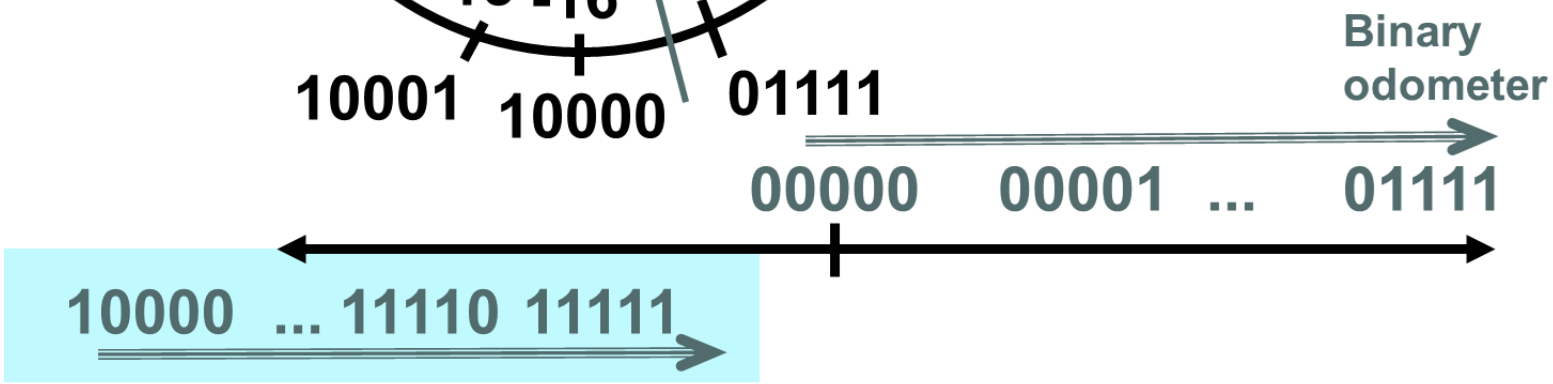
$$\bar{x} + 1 = -x$$

- Example: negate +2
  - +2 = 0000 0000 … 0010_{two}
  - –2 = 1111 1111 … 1101_{two} + 1
    = 1111 1111 … 1110_{two}

2's Complement Number "line": N = 5



- 2^{N-1} non-negatives
- 2^{N-1} negatives
- one zero
- how many positives?



Sign Extension

- Want to represent the same number using more bits than before?
  - Easy for positive #s (add leading 0's)
  - More complicated for negative #s
  - Sign and magnitude: add 0's *after* the sign bit
  - One's complement: copy MSB
  - Two's complement: copy MSB
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

In RISC-V instruction set
lb: sign-extend loaded byte
lbu: zero-extend loaded byte

Addresses

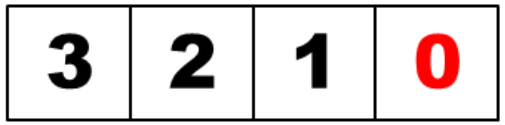
- The size of an address in bytes depends on the architecture 32-bit or 64-bit
- 32 bit architecture has 2^{32} possible addresses
- If a machine is byte-addressed, then each of its addresses points to a unique byte
- Word-addresses point to a word
- On a byte-addressed machine, how can we order the bytes of an integer in memory?
  - It depends on it's 'Endianness'

Little/Big Endian

- a **32 bit string** of binary characters – **a 'word'**
- 4 bits represent a hex character
- **a 'byte'** with 8 bits can hold **2 hex characters**
- *A byte address will point to 2 hex characters*
- a 32 bit machine will hold *8 such hex characters in a memory word address*
- abcdef12 corresponds to **8 hex characters (32 bits) as identified in red below. Pairs of these hex characters (in red) correspond to bytes:**
- ***0x* ab cd ef 12_{16} = 10101011 11001101 11101111 00010010_2**
- Virtually *all ISAs are byte addressed* providing access to chunks of data with granularity

Little/Big Endian

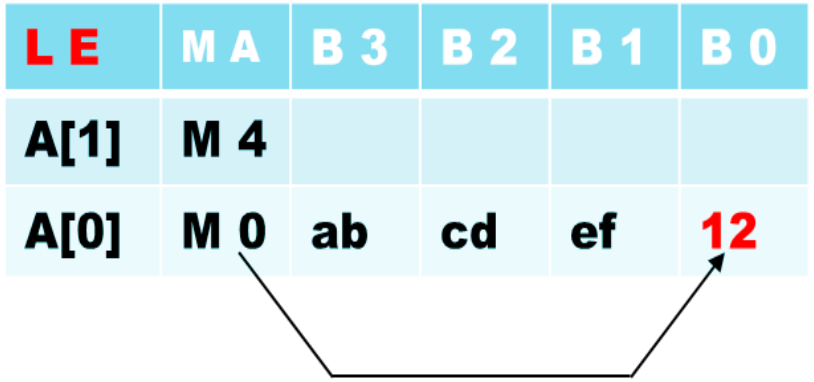
- Two different conventions for ordering the bytes within the larger object:
- ***Little Endian byte order*** puts the byte whose address is ‘x…000’ at the ***least significant position of the word***. The bytes are numbered:



- ***Big Endian byte order*** puts the byte with the same address as above: ‘x…000’ at the ***most significant position in the word***. The bytes are numbered:

Little Endian

- Byte ordering is a problem when exchanging data between computers with a different ordering
- RISC-V like ARM is ‘little endian’
- ***In little endian:***
- A byte address in a 32-bit machine points to least significant pair of hex char



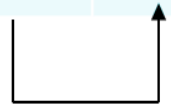
- ‘12’ in the above string goes to the lowest byte
- ‘ef’ goes to the next significant byte
- ‘cd’ to the next
- ‘ab’ to the most significant byte in a 32 bit or 4-byte string

Big Endian

- Byte ordering is a problem when exchanging data between computers with a different ordering
- RISC-V like ARM is ‘little endian’
- ***In big endian:***
- A byte address in a 32-bit machine points to most significant pair of hex char

- ‘12’ in the above string goes to the highest byte
- ‘ef’ goes to the next less significant byte
- ‘cd’ to the next
- ‘ab’ to the least significant byte in a 32 bit or 4-byte string

B E	M A	B 3	B 2	B 1	B 0
A[1]	M 4				
A[0]	M 0	12	ef	cd	ab



Endianness Footnotes

- Endianness only applies to values that occupy multiple bytes
- Endianness refers to storage in memory not number representation
- Example: char c = 97
- c == 0b01100001 in both big and little endian

RISC V Instructions

- **32 bits (4 bytes, 1 word) in length** and must be stored at **wordaligned** memory locations
- Instruction Encoding
- User-Level Instructions
- Arithmetic Instructions (ADD, SUB, …)
- Logical Instructions (AND, OR, XOR, …)
- Shifting Instructions (SLL, SRL, SRA, …)
- Miscellaneous Instructions
- Branch and Jump Instructions
- Load and Store Instructions
- Integer Multiply and Divide
- Compressed Instructions

Instruction Formats – register ops

- Since there are **32 registers**, a field with a **width of 5 bits** is used to encode each register operand within instructions
- The 3 registers in an instruction are symbolically called
  - RegD – The destination
  - Reg1 – The first operand
  - Reg2 – The second operand
- In addition, a number of instructions contain **immediate data.** The immediate data value is always sign-extended to yield a 32-bit value

6 RISC V Instruction Formats

- **R-Format**: instructions using *3 register inputs* – `add, xor, mul` — arithmetic/logical ops
- **I-Format**: instructions with *immediates*, `loads, addi, lw, jalr, slli`
- **S-Format**: *store* instructions: `sw, sb`
- **SB-Format**: *branch* instructions: `beq, bge`
- **U-Format**: instructions with *upper immediates* – `lui, auipc` — upper immediate is 20-bits
- **UJ-Format**: *jump* instructions: `jal`

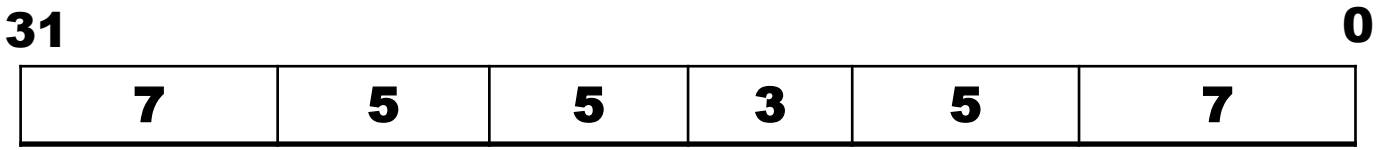
6 Instruction Formats

	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

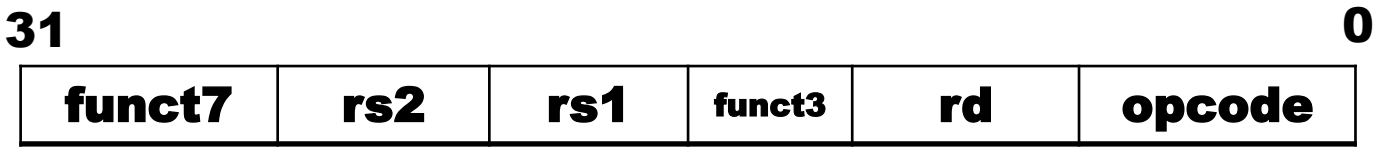
R-Format Instructions 1/4

- Define “fields” of the following number of bits each:

7 + 5 + 5 + 3 + 5 + 7 = 32

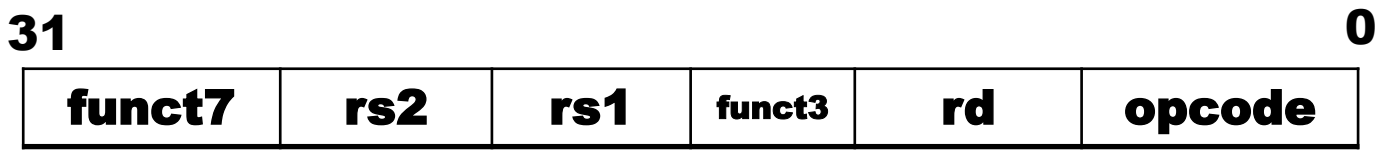


- Each field has a name:



- Each field is viewed as its own unsigned int
  - 5-bit fields can represent any number 0-31, while 7-bit fields can represent any number 0-127, etc.

R-Format Instructions 2/4

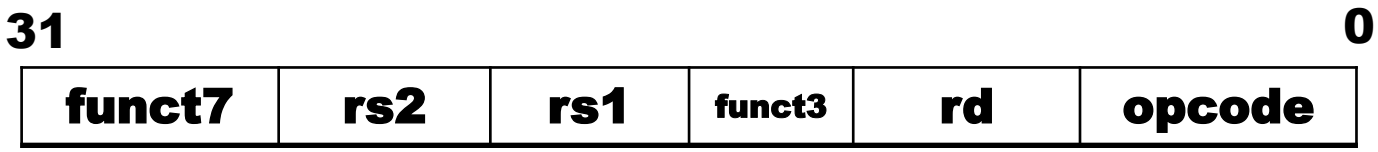


- opcode (7): partially specifies operation
  - e.g. R-types have opcode = `0b0110011`,
  - SB (branch) types have opcode = `0b1100011`

- funct7+funct3 (10): combined with opcode, these two fields describe what operation to perform

- How many R-format instructions can we encode?
  - With `opcode` fixed at `0b0110011`, just `funct` varies:
  - $2^7 \times 2^3 = 2^{10} = 1024$

R-Format Instructions 3/4



- rs1 (5): 1st operand (“source register 1”)
- rs2 (5): 2nd operand (second source register)
- rd (5): “destination register” — receives the result of computation
- Recall: RISCV has 32 registers
  - A 5 bit field can represent exactly 2^5 = 32 things (interpret as the register numbers x0-x31)

R-Format Instructions 4/4

Operands:

- RegD,Reg1,Reg2

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Example:

- **ADD x9,x21,x9** *# x9 = x21+x9*

funct7	rs2	rs1	funct3	rd	opcode
0000000	01001	10101	000	01001	0110011

Encoding (all R-type):

XXXX XXX2 2222 1111 1XXX DDDD DXXX XXXX

DDDDD = RegD
11111 = Reg1
22222 = Reg2
VVVVV = Immediate value
XXXXX = Op-code / function code

The opcode is the rightmost 7 bits. We decode the rest of the instruction by looking at the field values. The *funct7 and funct3 fields are both zero, indicating the instruction is add*. The decimal values for the register operands are 9 for the rs2 field, 21 for rs1, and 9 for rd. These numbers represent registers x9, x21, and x9. Now we can reveal the assembly instruction: `add x9,x21,x9`

Reading from Green Sheet

```
add x5 x7 x6
```

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
sd	S	0100011	011		23/3
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20
sll	R	0110011	001	0000000	33/1/00
slt	R	0110011	010	0000000	33/2/00

31 0

funct7	rs2	rs1	funct3	rd	opcode
0	**7**	**6**	**0**	**5**	**0x33**
0000 000	0 0111	0011 0	000	0010 1	011 0011
0000 0000 0111 0011 0000 0010 1011 0011					

hex representation: `0x 0073 02B3`

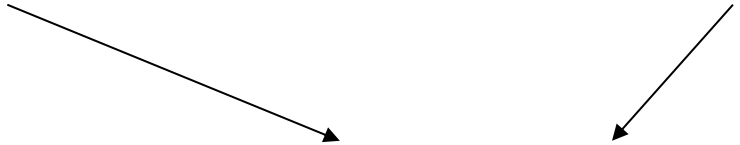
decimal representation: `7,537,331`

Called a Machine Language Instruction:

0000 000 0 0111 0011 0000 0010 1011 0011

All R-Format Instructions

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



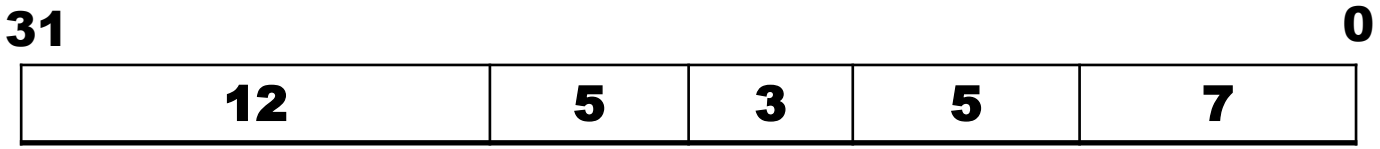
Different encoding in funct7 + funct3 selects different operations

I-Format Instructions 1/4

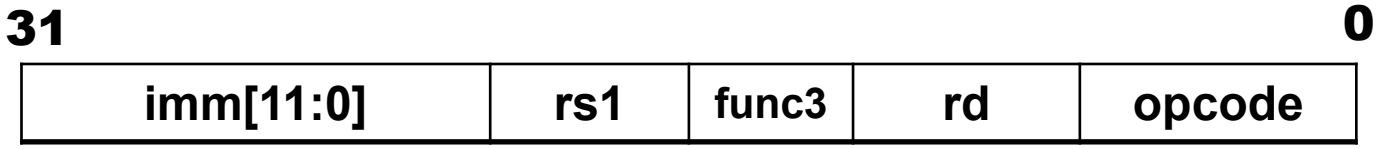
- What about instructions with immediates?
  - *5-bit field too small for most immediates*
- Ideally, RISCV would have only one instruction format (for simplicity)
  - *Unfortunately here we need to compromise*
- Define new instruction format that is mostly consistent with R-Format
  - *First notice that, if instruction has immediate, then it uses at most 2 registers (1 src, 1 dst)*

I-Format Instructions 2/4

- Define “fields” of the following number of bits each: 12 + 5 + 3 + 5 + 7= 32

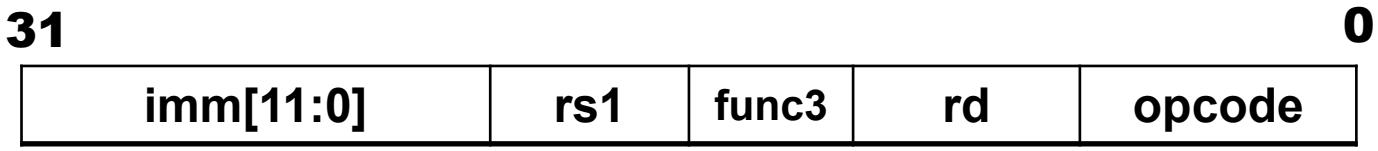


- Each field has a name:



- Key Concept: Only imm field is different from R-format: rs2 and funct7 replaced by 12-bit signed immediate, imm[11:0]

I-Format Instructions 3/4



- **opcode** (7): uniquely specifies the instruction
- **rs1** (5): specifies a **s**ource **r**egister operand
- **rd** (5): specifies **d**estination **r**egister that receives result of computation
- **immediate** (12): 12 bit number
  - All computations done in words, so 12-bit immediate must be extended to 32 bits
  - Always sign-extended to 32-bits before use in an arithmetic operation
  - Can represent 2^{12} different immediates – imm[11:0] can hold values in range [-2^{11} , $+2^{11}$)

I-Format Instructions 4/4

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

immediate	rs1	funct3	rd	opcode
1	9	0	9	19

Operands:

- RegD,Reg1,Immed-12

Example:

- `ADDI x9,x9,1  # x9 = x9+1`

Encoding:

VVVV VVVV VVVV 1111 1XXX DDDD DXXX XXXX

The 12-bit immediate is interpreted as a two's complement value, so it can represent integers from -2^{11} to $2^{11}-1$. When the I-type format is used for load instructions, the immediate represents a byte offset, so the load doubleword instruction can refer to any *doubleword* within a region of $\pm 2^{11}$ or 2048 bytes ($\pm 2^{8}$ or 256 doublewords) of the base address in the base register rd

DDDDD = RegD
11111 = Reg1
22222 = Reg2
VVVVV = Immediate value
XXXXX = Op-code / function code

Reading from Green Sheet

```
addi x15 x1 -50
```

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
fence.i	I	0001111	001		0F/1
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00

31 **0**

imm[11:0]	rs1	func3	rd	opcode
-50	**1**	**0**	**15**	**0x13**
1111 1100 1110	0000 1	000	0111 1	001 0011
1111 1100 1110 0000 1000 0111 1001 0011				

hex representation: 0x FCE0 8793

decimal representation: 4,242,573,203

Called a Machine Language Instruction:

1111 1100 1110 0000 1000 0111 1001 0011

All RISC V I-Type Arithmetic Instructions

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)

Question

If the number of registers were halved, which statement is true?

1. There must be more R-type instructions
2. There must be less I-type instructions
3. Shift amounts would change to 0-63
4. I-type instructions could have 2 more immediate bits

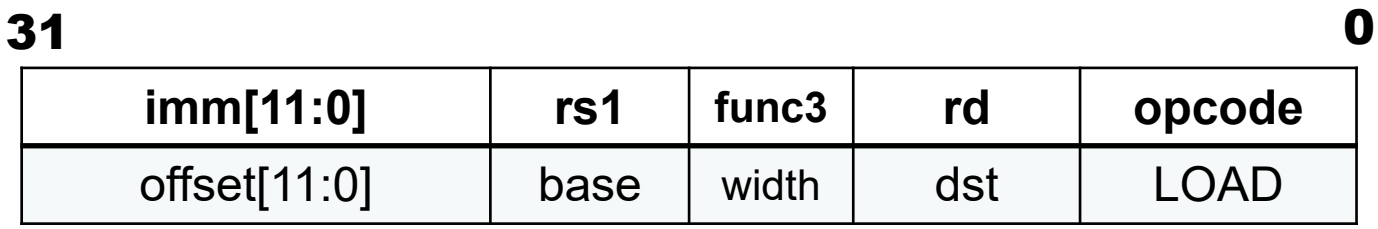
One fewer bit required to decode a register. So, rs1 and rd fields would need to be only 4 bits wide and not 5 bits

So, the imm field could have 2 more bits

31 0

imm[11:0]	rs1	func3	rd	opcode
12	5	3	5	7
14	4	3	4	7

Load Instructions are also I-Format



- 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
- Value loaded from memory is stored in `rd`

I-Format Load example

- `lw x14, 8(x2)`

31 0

imm[11:0]	rs1	func3	rd	opcode
offset[11:0]	base	width	dst	LOAD
0000 0000 1000	0001 0	010	0111 0	000 0011
imm = +8	rs1=x2	lw	rd=x14	LOAD

All I-Format Load Instructions

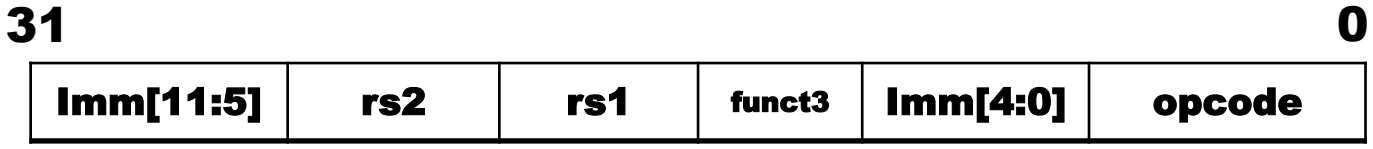
imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

funct3 field encodes size and signedness of load data

- LBU is “load unsigned byte”
- LH is “load halfword”, which loads 16 bits (2 bytes) and sign-extends to fill destination 32-bit register
- LHU is “load unsigned halfword”, which zero-extends 16 bits to fill destination 32-bit register
- There is no LWU in RV32, because there is no sign/zero extension needed when copying 32 bits from a memory location into a 32-bit register

S-Format Instructions

- Store needs to read two registers, `rs1` for base memory address, and `rs2` for data to be stored, as well as need immediate offset!
- Can't have both `rs2` and immediate in same place as other instructions!
- Note: stores don't write a value to the register file, no `rd`!
- RISC-V design decision is **move low 5 bits of immediate** to where `rd` field was in other instructions – keep `rs1/rs2` fields in same place
- register names more critical than immediate bits in hardware design



S-Format Load example

- `sw x14, 8(x2)`

31 0

Imm[11:5]	rs2	rs1	funct3	Imm[4:0]	opcode
00000000	01110	00010	010	01000	0100011
off[11:5] = 0	rs2=x14	rs1=x2	SW	01000	0100011

0000000	01000

Combined 12 bit offset = 8

All RV32 Store Instructions

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

Branching Instructions

- `beq, bne, bge, blt`
  - Need to specify an address to go to
  - Also take two registers to compare
  - Doesn't write into a register (similar to stores)
- How to encode label, i.e., where to branch to?
- Branches typically used for loops (`if-else, while, for`)
  - Loops are generally small (< 50 instructions)
- Instructions stored in a localized area of memory (Code/Text)
  - Largest branch distance limited by size of code
  - Address of current instruction stored in the program counter (PC)

Branch Reach

- PC-Relative Addressing: Use the `immediate` field as a two's complement offset to PC
  - Branches generally change the PC by a small amount
  - Can specify ± 2^{11} ***addresses*** from the PC
- Why not use byte address offset from PC as the `immediate`?
- RISCV uses 32-bit addresses, and memory is **byte-addressed**
- Instructions are "**word-aligned**": Address is always a multiple of 4 (in bytes)
- PC ALWAYS points to an instruction
  - PC is ***typed*** as a pointer to a word
  - can do C-like pointer arithmetic
- Let `immediate` specify **#words** instead of **#bytes**
  - Instead of specifying ± 2^{11} **bytes** from the PC, we will now specify ± 2^{11} **words** = **± 2^{13} byte addresses around PC**

Branch Calculation

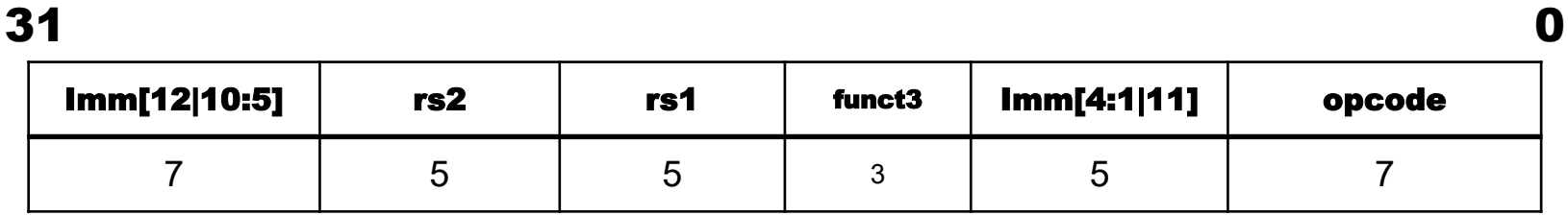
- If we don't take the branch:
  - PC = PC+4 = next instruction
- If we do take the branch:
  - PC = PC + (immediate*4)
- Observation:
  - `immediate` is number of instructions to move (remember, specifies words) either forward (+) or backwards (–)

RISC V Feature, n×16-bit instructions

- Extensions to RISC-V base ISA support 16-bit **compressed instructions** and also variable-length instructions that are multiples of 16-bits in length
- 16-bit = half-word
- To enable this, RISC-V scales the branch offset to be **half-words** even when there are no 16-bit instructions
- Reduces branch reach by half and means that ½ of possible targets will be errors on RISC-V processors that only support 32-bit instructions (as used in this class)
- RISC-V conditional branches can only reach $\pm 2^{10} \times$ 32-bit instructions either side of PC

RISC-V Format for Branches

- SB-format is mostly same as S-Format, with two register sources (`rs1`/`rs2`) and a 12-bit `immediate`
- But now `immediate` represents values -2^{12} to $+2^{12}-1$ in 2-byte increments
- The 12 `immediate` bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)



RISC-V Branch Example 1/2

- **RISCV Code:**

```
Loop: beq    x19,x10,End
      add  x18,x18,x10
      addi     x19,x19,-1
      j    Loop
      End:     <target instr>
```

Start counting from instruction AFTER the branch

1

2

3

4 instructions

So, **Branch offset = 4 instructions** = 16 bytes

- (Branch with offset of 0, branches to itself)

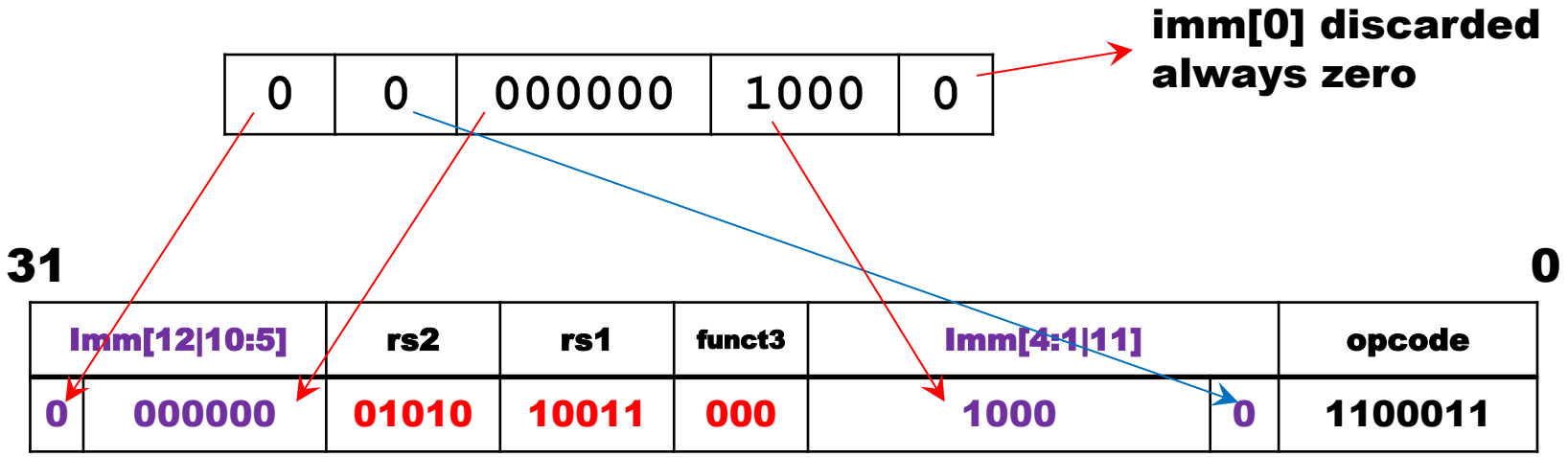
31 **0**

Imm[12|10:5]	rs2	rs1	funct3	Imm[4:1|11]	opcode
7	5	5	3	5	7
	01010	**10011**	**000**		**1100011**
0000000	**rs2=10**	**rs1=19**	**000**	**10000**	**1100011**

RISC-V Branch Example 2/2

beq x19,x10,offset = 16 bytes

13-bit immediate, imm[12:0], with value 16



All RISC-V Branch Instructions

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

Questions on PC-addressing

- 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 (why?)
- What do we do if destination is > 2^{10} instructions away from branch?
  - Other instructions save us:

```
beq x10,x0,far          bne x10,x0,next
# next instr    →       j far  ⟶ Use of large (20b) immediates
                next:# next instr
```

S-type & SB-type instructions

Operands:

- Reg1,Reg2,Immed-12

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Imm[12|10:5]		rs2	rs1	funct3	Imm[4:1|11]		opcode
0	11 1101	0 1011	0 1010	001	0000	1	110 0011

Example:

```
bne x10, x11, 2000 // if x10 != x11, go to location 2000₁₀ = 01 11 1101 0000
```

Encoding:

```
VVVV VVV2 2222 1111 1XXX VVVV VXXX XXXX
```

```
DDDDD = RegD
11111 = Reg1
22222 = Reg2
VVVVV = Immediate value
XXXXX = Op-code / function code
```

The three RISC-V instruction formats reviewed so far are R, I, and S. The R-type format has two source register operand and one destination register operand. The I-type format replaces one source register operand and the func7 field with a 12-bit immediate field. The S-type format has two source operands and a 12-bit immediate field, but no destination register field. The S-type immediate field is split into two parts, with bits 11—5 in the leftmost field and bits 4—0 in the second-rightmost field

Dealing With Large Immediates

- How do we deal with 32-bit immediates?
  - Our I-type instructions only give us 12 bits
- Solution: Need a different instruction format for dealing with the rest of the 20 bits.
- This instruction should provide:
  - destination register to put the 20 bits into
  - immediate field of 20 bits
  - an instruction opcode

U-Format for “Upper Immediate” instructions

- 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
  - `AUIPC` – Add Upper Immediate to PC

31 0

imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI/AUIPC

`lui` to create long immediates

- `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`).

```
lui x10, 0x87654          # x10 = 0x87654000
addi x10, x10, 0x321      # x10 = 0x87654321
```

Corner Case

- • How to set `0xDEADBEEF`?

```
lui x10, 0xDEADB # x10 = 0xDEADB000
addi x10, x10,0xEEF # x10 = 0xDEADAEEF
```

- `addi` 12-bit immediate is always sign-extended!
  - if top bit of the 12-bit immediate is a 1, it will subtract 1 from upper 20 bits

Solution

- How to set 0xDEADBEEF?

```
lui x10, 0xDEADC # x10 = 0xDEADC000
addi x10, x10,0xEEF # x10 = 0xDEADBEEF
```

- Pre-increment value placed in upper 20 bits, if sign bit will be set on immediate in lower 12 bits.
- Assembler pseudo-op handles all of this:

```
lui x10, 0xDEADBEEF  # Creates two instructions
```

AUIPC

- Adds upper immediate value to PC and places result in destination register
- Used for PC-relative addressing

```
AUIPC x4,0x12345 # x4 = PC + (0x12345<<12)
```

- Sets **rd** (x4 in above example) to the sum of the current PC and a 32-bit value with the low 12 bits as 0 and the high 20 bits coming from the U-type 20b immediate

U-Format Instructions

Operands:

- RegD, Immed-20

Imm[31:12]	rd	opcode
0000 0000 0011 1101 0000	10011	0110111

Example:

- **lui x19, 976** // 976_{10} = 0000 0000 0011 1101 0000_2

Encoding:

VVVV VVVV VVVV VVVV VVVV DDDD DXXX XXXX

The RISC-V instruction set includes the instruction *Load upper immediate* (`lui`) to load a 20-bit constant into bits 12 through 31 of a register. The leftmost 32 bits are filled with copies of bit 31, and the rightmost 12 bits are filled with zeros. This instruction allows, for example, a 32-bit constant to be created with two instructions. `lui` uses a new instruction format, `U-type`, as the other formats cannot accommodate such a large constant

DDDDD = RegD
11111 = Reg1
22222 = Reg2
VVVVV = Immediate value
XXXXX = Op-code / function code

Jump Vs Branch

- Jump and Branch instructions used for transfer of control
- Branch instructions transfer conditionally – by comparing contents of 2 source registers
- Jump instructions transfer unconditionally
- In Function (subroutine) calls, Jump [`jal`] stores the return address in the (dedicated) register ‘x1’
- `jal` loads the address of the next instruction into ‘x1’ or the ‘ra’ register (slide 17) and loads the PC with the jump-to-address

Procedure Call Instructions

- **Procedure call:** jump and link

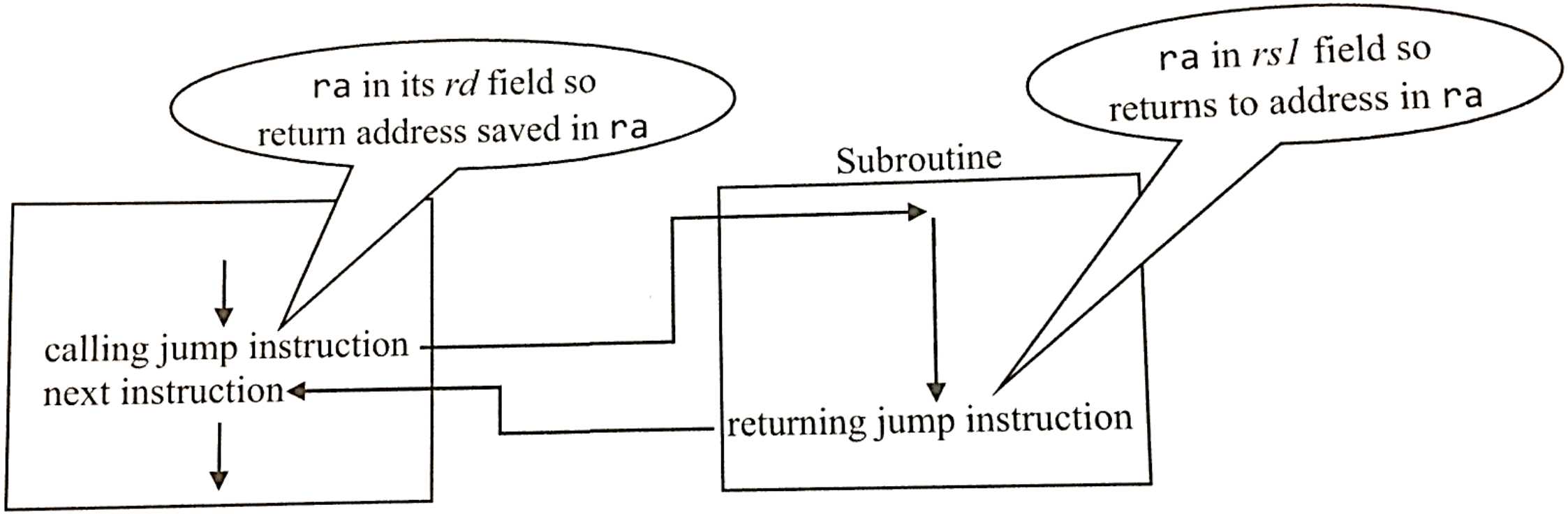
  `jal x1, ProcedureLabel`

  - Address of following instruction put in `x1`
  - Jumps to target address
- **Procedure return:** jump and link register

  `jalr x0, 0(x1)`

  - Like `jal`, but jumps to 0 + address in `x1`
  - Use `x0` as `rd` (`x0` cannot be changed)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

jal, jalr


ra in its rd field so
return address saved in ra
ra in rs1 field so
returns to address in ra
Subroutine
calling jump instruction
next instruction
returning jump instruction

UJ-Format Instructions, `jal`

- For branches, we assumed that we will not need to branch too far, so we can specify a ***change*** in the PC
- For general jumps (`jal`), we may jump to ***anywhere*** in code memory
  - Ideally, we would specify a 32-bit memory address to jump to
  - Unfortunately, we can't fit both a 7-bit `opcode` and a 32-bit address into a single 32-bit word
  - Also, when linking we must write to an `rd` register

UJ-Format Instruction `jal`

- `jal` saves PC+4 in register `rd` (or 'x1' the return address)
- Uses immediate encoding (20 bits) for destination address
  - Set PC = PC + offset (PC-relative jump)
  - Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart {Note: imm[0]=0 always}
  - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

31 **0**

imm[20|10:1|11|19:12]	rd	opcode
20	5	7
offset[31:12]	dest	`jal`

`jalr` Instruction (I-Format)

- `jalr rd, rs1, offset`
- Writes PC+4 to `rd` (return address), just as `jal`
  - Uses indirect address: Sets PC = `rs1 + offset`
  - Use '`x0`' instead of '`rd`' if you don't need to 'return'
- Uses same immediates as arithmetic & loads
  - **no** multiplication by 2 bytes

31 0

imm[11:0]	rs1	func3	rd	opcode
offset	base	0	dst	`jalr`

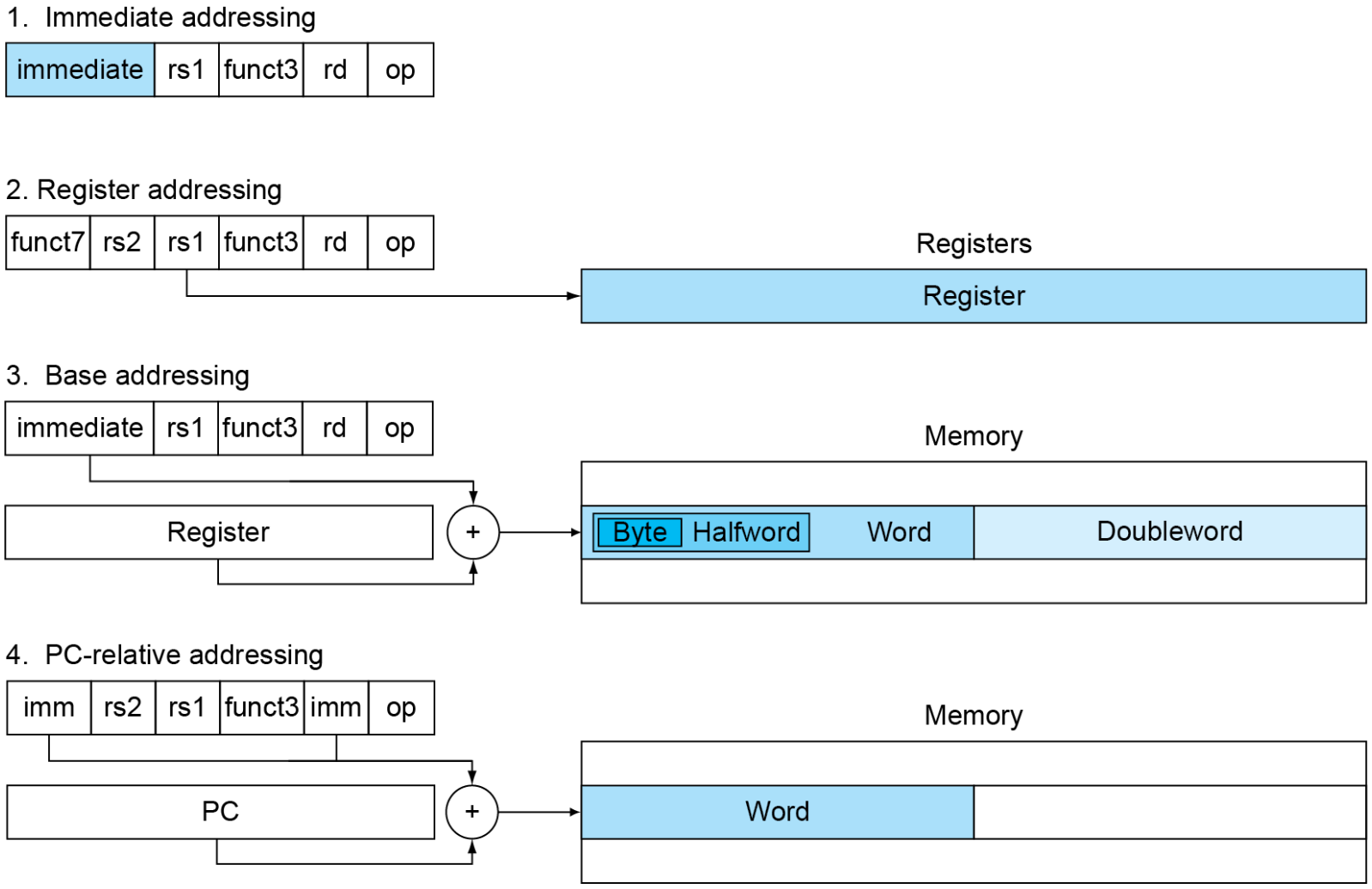
Pseudo Instruction, j

- # j pseudo-instruction
- j Label = jal x0, Label # Discard return address
- # Call function within +/- 2^{19} instructions of PC
- jal ra, FuncLabel

31 0

imm[20|10:1|11|19:12]	rd	opcode
20	5	7
offset[31:12]	dest	jal

RISC-V Addressing Summary



RISC V “C” Instructions

- RISC-V Compressed (RVC), a variable-length instruction set extension. RVC is a superset of the RISC-V ISA
- Both-16-bit and 32-bit instructions can be intermixed -there is no “mode” bit to put the processor into “compressed-instruction” mode
- RVC encodes the most frequent instructions in half the size of a RISC-V instruction
- RVC improves the performance and energy per operation of RISC-V
- RVC programs are 25% smaller than RISC-V programs, fetch 25% fewer instruction bits than RISCV programs, and incur fewer instruction cache misses

RISC V “C” Instructions

- RISC-V uses three-address instructions. The instruction:

```
ADD x8,x8,x10      # Add x10 to x8
```

is compressed to:

```
C.ADDW RegD,RegB   # RegD = RegD + RegB
```

- Compressed instructions are identified with the “C.” prefix. The opcode ends with a “W” suffix, indicating that it operates on a 32-bit word, as opposed to 64-bit or 128-bit values.
- Compressed instructions are expanded to equivalent 32-bit instructions by the FETCH and DECODE hardware, after an instruction is fetched from memory, directly before it is executed
- However, the expansion to a full-sized 32-bit instruction could be performed by the assembler. This would be useful when assembling for a machine that does not support the compressed extension

RISC V “C” Instructions

For instructions of length 32 bits, the least significant 2 bits must be 11. In addition, the next 3 bits must not be 111, since this indicates an instruction of length greater than 32-bits.

Compressed (16-bit) Instructions:

```
xxxx xxxx xxxx xxaa
```

(where aa ≠ 11)

Normal (32-bit) Instructions:

```
xxxx xxxx xxxx xxxx xxxx xxxx xxxb bb11
```

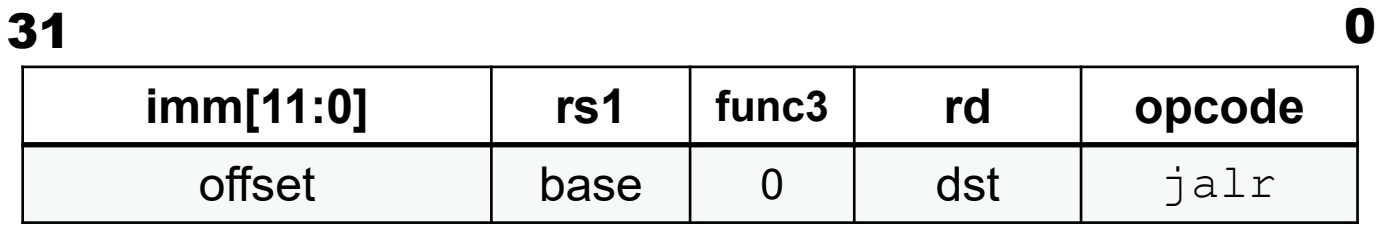
(where bbb ≠ 111)

Longer Instructions:

```
xxxx xxxx ... xxxx xxxx xxxx xxxx xxx1 1111
```

Uses of `jalr`

- `# ret and jr psuedo-instructions`
- `ret = jr ra = jalr x0, ra, 0`
- `# Call function at any 32-bit absolute address`
- `lui x1, <hi 20 bits>`
- `jalr ra, x1, <lo 12 bits>`
- `# Jump PC-relative with 32-bit offset`
- `auipc x1, <hi 20 bits>`
- `jalr x0, x1, <lo 12 bits>`



Question

When combining two C files into one executable, we can compile them independently and then merge them together.

When merging two or more binaries:

1) `Jump` instructions don't require any changes

2) `Branch` instructions don't require any changes

	1	2
(A)	F	F
(B)	F	T
(C)	T	F
(D)	T	T

Comments on Instruction Formats

Note that Reg1, Reg2, and RegD occur in the same place in all instruction formats.

This simplifies the chip circuitry, which would be more complex if, for example, RegD was sometimes in one part of the instruction and other times in a different place in the instruction.

A consequence of keeping the register fields in that same place is that immediate data values in an instruction are sometimes not always in contiguous bits. And note that the bits encoding the immediate values in the S-type and B-type instructions are not contiguous.

Order of Immediate fields

- **I-type:**
- VVVV VVVV VVVV ---- ---- ---- ---- ——
- **S-type and B-type:**
- VVVV VVV- ---- ---- ---- VVVV V--- ——
- **U-type and J-type:**
- VVVV VVVV VVVV VVVV VVVV ---- ---- ——
- While both S-type and B-type have a 12-bit immediate value, the precise order of the bits in those 2ields differs between the two formats. While you would reasonably assume the bits are in order, they are in fact scrambled up a little in the B-type instruction format. Similarly, the bits of the 20-bit immediate value field in the J-type formats are scrambled, as compared to U-type. Consult the spec for details.
- Immediate values are always sign-extended. Although the immediate value can be different sizes and may be broken into multiple fields, the sign bit is always in the same instruction bit (namely the leftmost bit, bit 31). This simplifies and speeds sign-extension circuitry

Machine Size

- The RISC-V standard describe three different machine sizes:
  - RV32 32-bit registers
  - RV64 64-bit registers
  - RV128 128-bit registers
- Any particular RISC-V hardware chip will implement only one of the above standards.
- However, each size is strictly more powerful than the smaller sizes. An RV64 chip will include all the instructions necessary to manipulate 32-bit data, so it can easily do anything an RV32 chip can. Likewise, an RV128 chip is a strict superset of the RV32 and RV64 chips

Machine Size

- **Regarding 64-bit and 128-bit extensions:** For the larger machine sizes, the basic 32-bit instructions work identically.
- Smaller data values are sign-extended to fit into larger registers.
- Thus, 32-bit values are sign-extended to 64 bits on an RV64 machine. This means you can run 32-bit code directly on a 64-bit machine with no changes!
- Of course, some instructions from a RV64 machine will not be present on a RV32 machine, so code that truly requires 64-bits won't work.
- When using a 64-bit machine to execute code written for a 32-bit machine, the upper 32 bits of registers will normally contain the sign extension, and not a zero extension.
- Likewise, 32-bit and 64-bit values are sign-extended to 128-bits on an RV128 machine.
- The 12-bit and 20-bit immediate values in instructions are sign-extended to the basic machine size. This includes the immediate values in the two U-type instructions (LUI and AUIPC).