Chapter 2.
Instruction Set Architecture
(RISC-V)

Introductions to RSIC-V Instruction Set Architecture

What is RSIC-V Architecture?

- **RISC-V is an open standard ISA based on RISC principles**
  - A completely *open ISA* that is freely available to academia and industry (license-free, royalty-free)
  - Originated in 2010 by researchers at UC Berkeley
    - Krste Asanović, David Patterson and students
  - RISC-V is simpler and more elegant than 80x86 ISA
  - Recently rapid adoption by many vendors (Samsung, WesternDigital…)
  - Suitable for all levels of computing system, from micro-controllers to supercomputers
    - 32-bit, 64-bit, and 128-bit variants
- **5th RISC ISA design developed at UC Berkeley**
  - Now managed by the non-profit RISC-V foundation (https://riscv.org)

RISC-V®

RISC-V: The Free and Open RISC Instruction Set Architecture

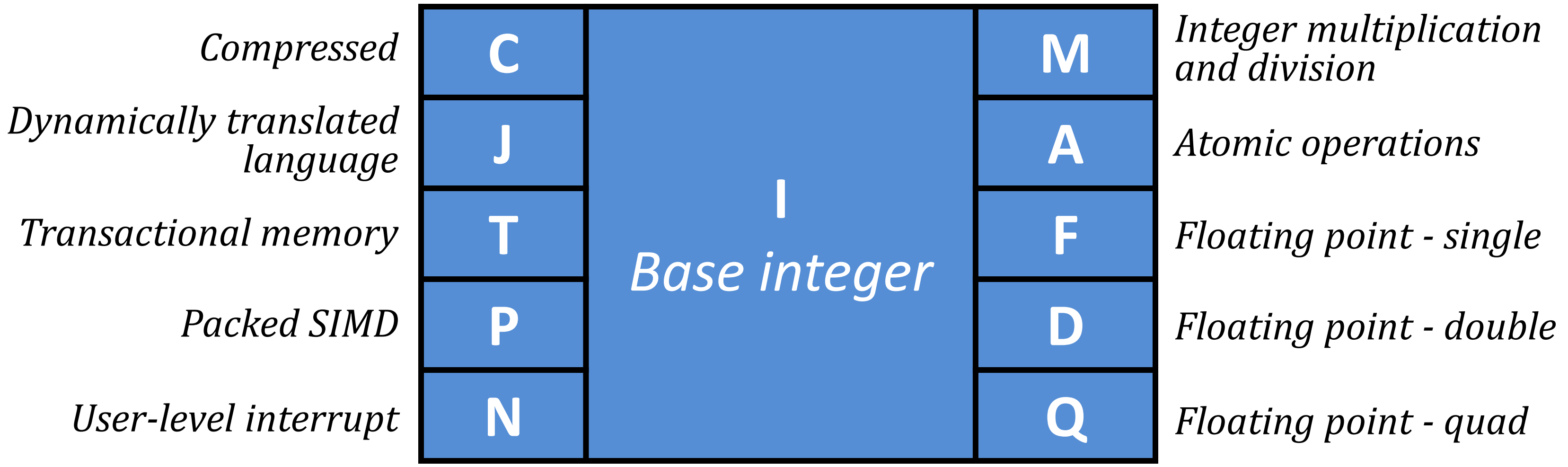
What is RSIC-V Architecture?

- **Where are operands stored?**
  - *32 General Purpose Registers*
  - *Register size is 64-bit*
- **How many explicit operands are there?**
  - (0,3) Register – Register (a.k.a load/store)
  - Can not operate with memory directly
- **How is the operand location specified (How to access memory)?**
  - ***Memory addressing : 0 ~ 2^{64}-1 (Typically, 0 ~ 2^{48}-1)***
  - ***Data is transferred between memory and register by 64-bit unit***
  - ***The size of instruction is 32-bit (why??? – recall IA-64)***
  - ***4 representative addressing mode: Register, Immediate, Base, PC-relative***
- **What type & size of operands are supported?**
  - byte, int, float, double, string, vector. . .
- **What operations are supported?**
  - add, sub, mul, move, compare . . .

What is RSIC-V Architecture?

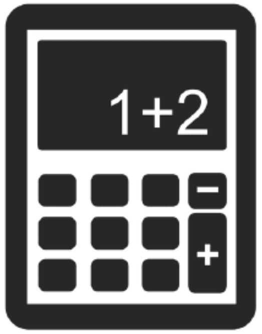
- *RISC-V is designed as modular*; **instead of taking simple but monolithic, the modular design enables a more flexible implementation that suit specific applications**
  - RISC-V is a federation of ISA extensions — from the baseline RV{32|64|128}I, to an arbitrary combination of a handful of extensions

RISC-V Instruction Set Architecture

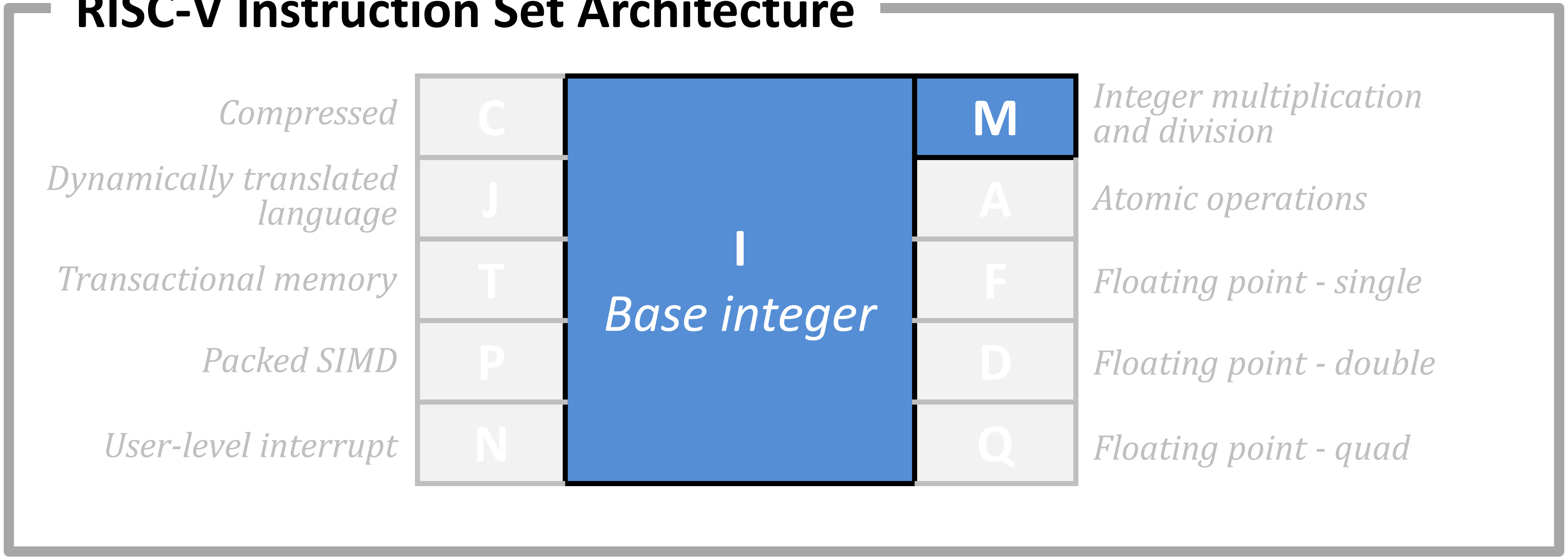


Appendix: Create Custom Processor

- **Simple integer calculator**
  - Only **I & M** modules are required



RISC-V Instruction Set Architecture



What is RSIC-V Architecture?

- **Separated into multiple specifications**
  - *User-Level ISA spec.*
  - Privileged ISA spec (supervisor-mode instructions)
  - More …
- **ISA support is given by *"RV + register-width + extensions"***
  - Base integer ISAs: RV<u>**32**</u>I (Only 40 instructions defined), RV<u>**64**</u>I, RV<u>**128**</u>I
    - **RV32I:** 32-bit RISC-V with support for the I(integer) instruction set
    - **RV32** has *32-bit registers* and *set of 32-bit instruction*. **RV64** has *64-bit registers* and also *a set of 32-bits per instruction*
    - RISC-V is a federation of ISA extensions — from the baseline RV{32|64|128}I, to an *arbitrary combination* of a handful of extensions

      : G = base (I) + above extensions (MAFD), (e.g., RV64IMAFDC == RV64GC)
    - **RV32E**: Special reduced version of RV32I with *16 registers for resource-constrained embedded systems* (others have 32 general purpose registers)
    - **RV32C:** Offers shorter 16-bit instruction (can be intermixed with 32-bit instructions)

Register @RISC-V

- **RSIC-V has a total of 33 user-visible registers**
  - Limited number of special places to hold values, built *directly into the hardware*
    - Very fast (~ 1ns)
    - Operations can only be performed on these registers
  - ***32 general purpose (integer) registers***
    - The register `x0` is hardwired to the constant "0" → ***why?***
    - `x1` to hold return address on a call
    - 64-bit wide (cf. 32-bit wide in RV32I)
  - ***1 special register, program counter (PC)***
    - Holds the address of the next instruction to be executed
    - In Intel architecture (e.g., x86-64, IA32), ***instruction pointer (IP)*** or instruction counter (IR)

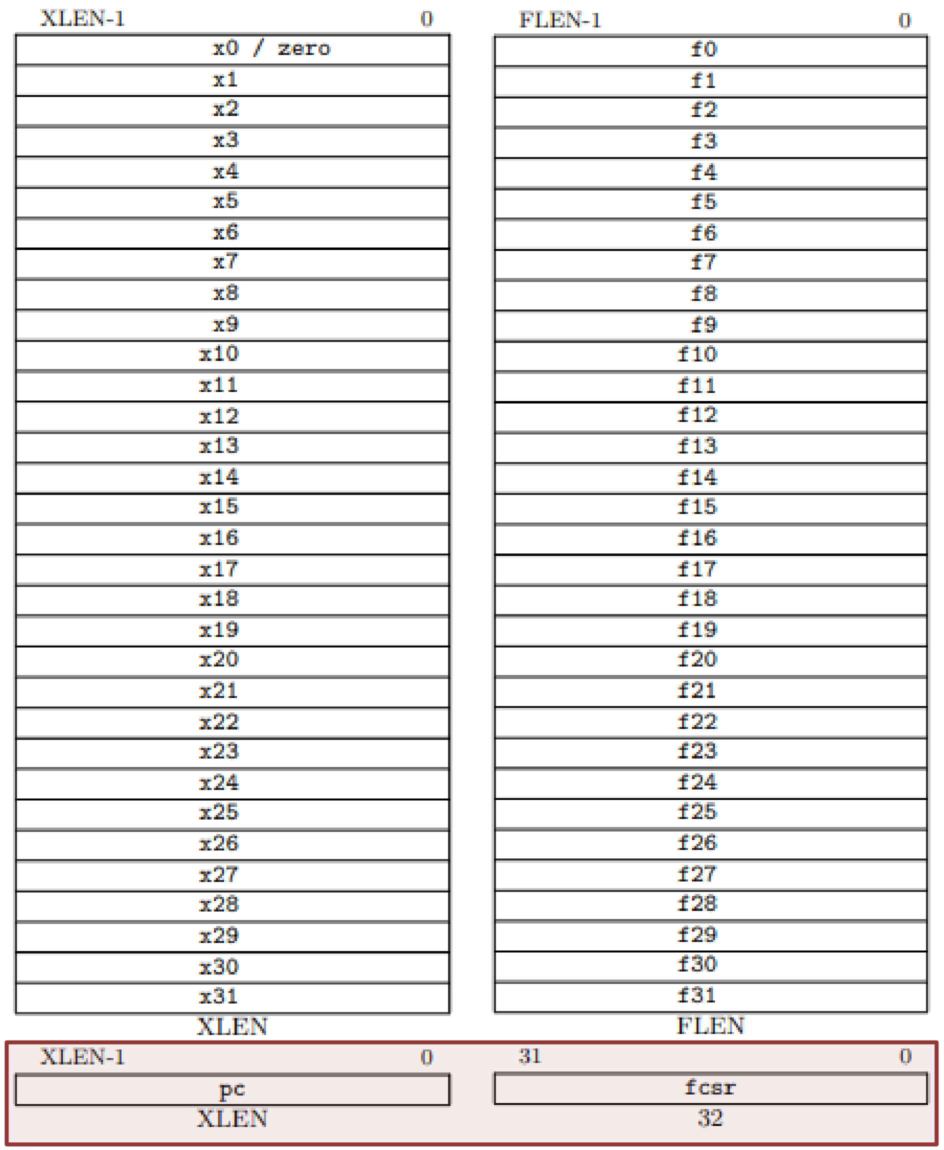
Symbolic name in assembly

Register	ABI Name	Description	Saver
`x0`	zero	hardwired zero	-
`x1`	ra	return address	Caller
`x2`	sp	stack pointer	Callee
`x3`	gp	global pointer	-
`x4`	tp	thread pointer	-
`x5-7`	t0-2	temporary registers	Caller
`x8`	s0 / fp	saved register / frame pointer	Callee
`x9`	s1	saved register	Callee
`x10-11`	a0-1	function arguments / return values	Caller
`x12-17`	a2-7	function arguments	Caller
`x18-27`	s2-11	saved registers	Callee
`x28-31`	t3-6	temporary registers	Caller
`PC`		program counter	

Q) How to make "0" in Intel architecture?

Appendix: Register @RISC-V

- RSIC-V also has a total of 33 floating-point (FP) registers (`f0–f31`)
  - *Each can contain a single- or double-precision FP value (32-bit or 64-bit IEEE FP)*
  - FP status register (fsr), used for FP rounding mode & exception reporting



Appendix: Constant Zero

- **Register `x0`**
  - Special register ***hard-wired to have value 0***
  - *Cannot be overwritten (modified)*

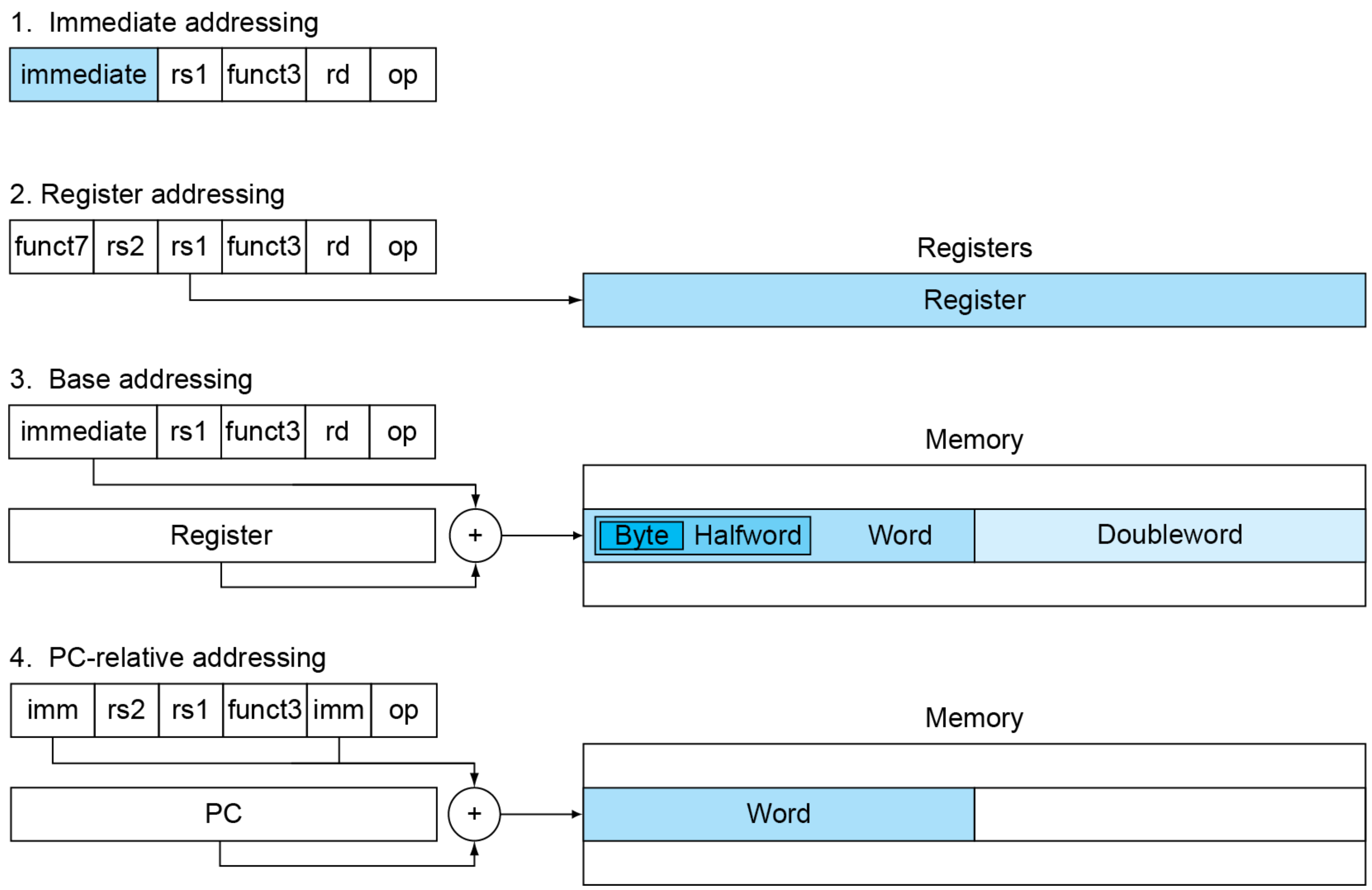
  - Useful for common operations
    - E.g., move data between registers

```
add x22, x21, x0        # x22 ← x21
```

    - E.g., negate the value

```
sub x22, x0, x22        # x22 = -x22
```

Addressing Mode @RISC-V – *Will Be Studied Later*



Operation @RISC-V

- ***Arithmetic or logical (ALU)*** **instructions on register data**
- ***Data transfer*** **instructions between memory and register**
  - Load data from memory into register
  - Store register data into memory
- ***Control (or Branch)*** **instructions**
  - Unconditional jump
  - Conditional branch
  - Procedure call and return

We will focus on these 3-types operation in RISC-V ISA !!!

Appendix: Signed and Unsigned Numbers

- **Numbers in base 2 (binary numbers)**
  - ex) 123 base 10 = 1111011 base 2 $\longrightarrow$ *How to convert?*
- **Value of *i*th digit d: d x Base^i**
  - $1101_2 = (1x2^3)+(1x2^2)+(0x2^1)+(1x2^0) = 8 + 4 + 0 + 1 = 13_{10}$
- **1101_2 in a** `doubleword`

63		55		47		39	32
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0

31		23		15		8	0
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	1 1 0 1

- LSB (least significant bit) the right most bit, bit position 0
- MSB (most significant bit) the left most bit, bit position 31

Appendix: Signed and Unsigned Numbers

- **RISC-V word can represent 2^{64} bit patterns**

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two} = 0_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = 2_{ten}$

. . . . . .

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} = 18,446,774,073,709,551,613_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = 18,446,744,073,709,551,614_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} = 18,446,744,073,709,551,615_{ten}$

- **Decimal values calculated using:**

$$(x_{63} \times 2^{63}) + (x_{62} \times 2^{62}) + (x_{61} \times 2^{61}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

→ representation of the unsigned numbers

- **Representation limit**
  - Computer can express the numbers only within the given bit range
  - If arithmetic operation results in the larger number than representable, ***overflow*** happens.

Appendix: Signed and Unsigned Numbers

- **Expressing the sign of a number**
  - One bit to indicate positive or negative sign
  - Where should the sign be? How does it affect the arithmetic operation?
- **Naïve solution**
  - Simply use the MSB as a sign → ***"sign and magnitude"*** representation
  - Pros: Easy to understand (or calculate)
  - Cons: Subtraction is difficult → It needs additional design(circuit) to support "subtraction"

$0000_2 = 0_{10}$	$1000_2 = -0_{10}$
$0001_2 = 1_{10}$	$1001_2 = -1_{10}$
$0010_2 = 2_{10}$	$1010_2 = -2_{10}$
$0011_2 = 3_{10}$	$1011_2 = -3_{10}$
$0100_2 = 4_{10}$	$1100_2 = -4_{10}$
$0101_2 = 5_{10}$	$1101_2 = -5_{10}$
$0110_2 = 6_{10}$	$1110_2 = -6_{10}$
$0111_2 = 7_{10}$	$1111_2 = -7_{10}$

$$\begin{aligned}11 - 4 &= 00001011 + 10000100\\ &= 10001111 = -15\ ???\end{aligned}$$

$$6 - 3 = 0110 + 1011 = ???$$

Appendix: Signed and Unsigned Numbers

- **1's complement**
  - Invert positive number bits to use as a negative number

$0000_2 = 0_{10}$	$1111_2 = -0_{10}$
$0001_2 = 1_{10}$	$1110_2 = -1_{10}$
$0010_2 = 2_{10}$	$1101_2 = -2_{10}$
$0011_2 = 3_{10}$	$1100_2 = -3_{10}$
$0100_2 = 4_{10}$	$1011_2 = -4_{10}$
$0101_2 = 5_{10}$	$1010_2 = -5_{10}$
$0110_2 = 6_{10}$	$1001_2 = -6_{10}$
$0111_2 = 7_{10}$	$1000_2 = -7_{10}$

6 – 3 = 0110 + 1100 = 0010 = 2 ???

6 – 3 = 0110 + 1100 **+ 1** = 0011 = 3

carry

- **2's complement**

$0000_2 = 0_{10}$	$1111_2 = -1_{10}$
$0001_2 = 1_{10}$	$1110_2 = -2_{10}$
$0010_2 = 2_{10}$	**1101**$_2 = -3_{10}$
0011$_2 = 3_{10}$	$1100_2 = -4_{10}$
$0100_2 = 4_{10}$	$1011_2 = -5_{10}$
$0101_2 = 5_{10}$	$1010_2 = -6_{10}$
$0110_2 = 6_{10}$	$1001_2 = -7_{10}$
$0111_2 = 7_{10}$	$1000_2 = -8_{10}$

6 **– 3** = 0110 + **1101** = **0011** = 3

Appendix: Signed and Unsigned Numbers

- **2's Complement**
  - All negative numbers have 1 in MSB
    - To find out if number is positive or negative, H/W needs to check only one bit
  - Computing the decimal value

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

ex) 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111100

$= (-1x2^{63})+(1x2^{62})+(1x2^{61})+\ldots+(1x2^{2})+(0x2^{1})+(0x2^{0})$

= -9,223,372,036,854,775,808 + 9,223,372,036,854,775,804 = -4

- **Negating 2's complement binary number**
  - Negate all bits and add 1

$0000_2 = 0_{10}$	$1111_2 = -1_{10}$
$0001_2 = 1_{10}$	$1110_2 = -2_{10}$
$0010_2 = 2_{10}$	**1101_2** $= -3_{10}$
0011_2 $= 3_{10}$	$1100_2 = -4_{10}$

2→-2: 0010 → 1101 → add 1 → 1110

-2→2: 1110 → 0001 → add 1 → 0010

Appendix: Signed and Unsigned Numbers

- **Sign extension**
  - Converting a binary number in *n* bits to a binary number in *m* bits, where m>n
  - Take the MSB and replicate it to fill up all the bits
  - ex) 0000 0010 = 2
    - → 0000 0000 0000 0010 = 2
  - ex) 1111 1110 = -2
    - → 1111 1111 1111 1110 = -2

- **Signed load performs sign extension to correctly load value**
  - loading 16 bit number into 32 bit register
  - instructions: `lb` (load byte), `lh` (load halfword), `lw` (load word)
  - If you don’t want the sign extension, use `lbu` (load byte unsigned)

Basics of RSIC-V Instruction Set Architecture

References

https://five-embeddev.com/

https://shakti.org.in/docs/risc-v-asm-manual.pdf

RISC-V ASSEMBLY LANGUAGE Programmer Manual Part I

https://web.cecs.pdx.edu/~harry/riscv/RISCV-Summary.pdf

https://riscv.org/technical/specifications/

https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md

https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf

RISC-V Green Card

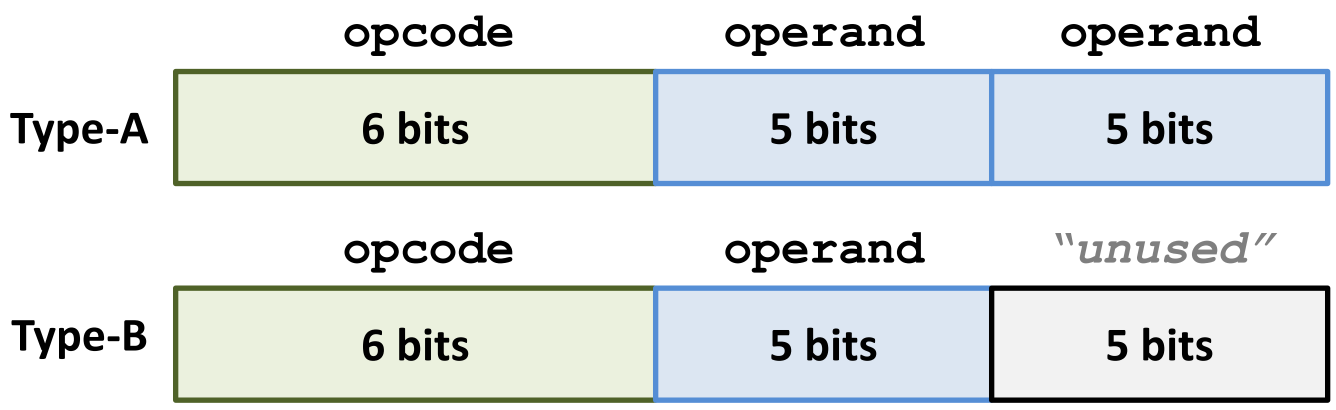
https://www.cs.sfu.ca/~ashriram/Courses/CS295/assets/notebooks/RISCV/RISCV_CARD.pdf

RISC-V Reference Card

Before RISC-V ISA …

- **RISC-V ISA is "fixed-length" instruction.**
  - Fixed length instruction presents a much more interesting challenge: "How to fit multiple sets of instruction types into same number of bits?"
  - Example:
    - 16-bit fixed length instructions, with 2 types of instructions
    - **Type-A**: 2 operands, each operand is 5-bit
    - **Type-B**: 1 operand of 5-bit
    - First solution) we make two types of instructions with the same fixed-length format → waste bits

Q) How many different instructions do we really have?

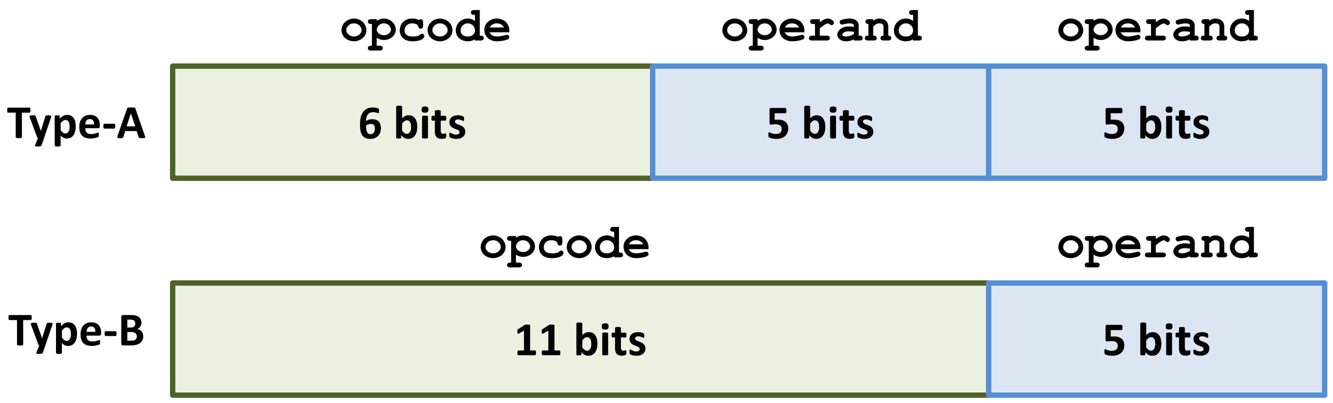


Before RISC-V ISA …

- **Use *expanding opcode* scheme**
  - Extend the opcode for **Type-B** instructions to 11 bits
  - No wasted bits and result in a larger instruction set

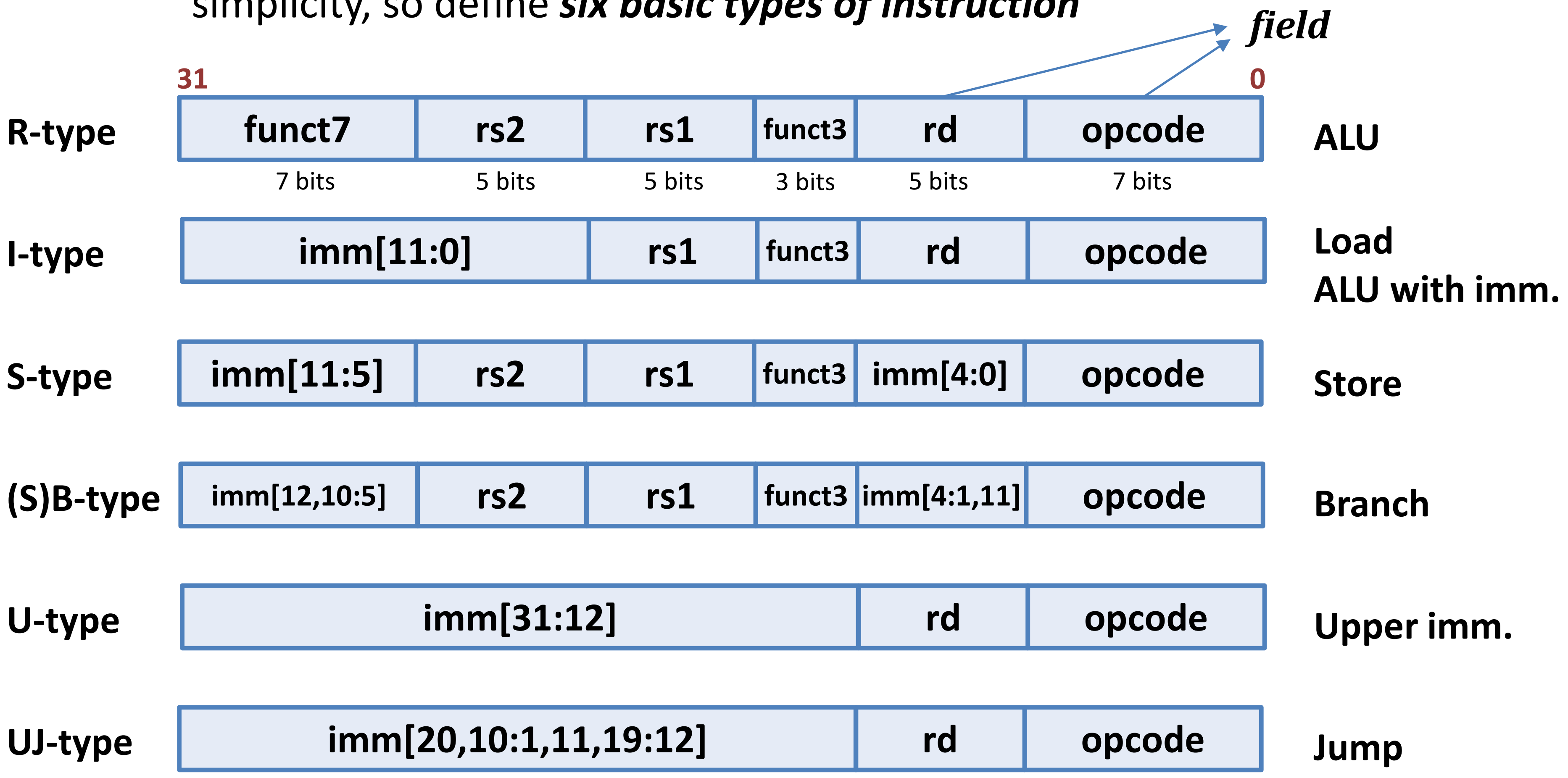
Q1) How do we distinguish between Type-A and Type-B instructions?

Q2) How many different instructions do we really have?



Instruction Format @RISC-V

- **32-bit wide instruction can be divided into multiple "fields"**
  - We could define different fields for each instruction, but RISC- V seeks simplicity, so define ***six basic types of instruction***



※ B-type instruction was called as SB-type before

(RISC-V ISA: R-Type Instruction) - Arithmetic

- **Arithmetic operations need *three operands:***
  - ***"Two sources"*** and "***One destination***"

```
add a, b, c            // a ← b + c
sub a, b, c            // a ← b - c
```

- **Arithmetic instructions use only *register operands***
  - In RISC-V, data in memory cannot be directly addressed by ALU instructions
  - Compiler must use registers for variables as much as possible. ***Why?***
    - Register is faster → Register optimization is important!!!

C code:

```
// f in x19
// g in x20
// h in x21
// i in x22
// j in x23

f = (g + h) – (i + j);
```

Compiled RISC-V code:

```
add  x5, x20, x21
add  x6, x22, x23
sub  x19, x5, x6
```

(RISC-V ISA: R-Type Instruction) - Format

- **R-type *(Register-type)* instruction format (32-bit wide)**
  - Assembly (e.g., register-register addition)

    `ADD rd, rs1, rs2`

  - Semantics
    - **`GPR[rd] ← GPR[rs1] + GPR[rs2]`**
    - **`PC`** ← **`PC`** + **4** (The length of instruction is 32-bit wide.)
  - Exception: None (<u>ignore overflow</u>)

    Will be explained later slide

  - Variations
    - Arithmetic: {`ADD`, `SUB`}
    - Compare: {signed, unsigned} X {`SLT`(Set if Less Than)}
    - Logic: {`AND`, `OR`, `XOR`}
    - Shift: {`SLL`, `SRL`, `SRA` (Left, Right-Logical, Right-Arithmetic)}

(RISC-V ISA: R-Type Instruction) - Format

- R-*type (Register-type)* instruction format (32-bit wide)

31 funct7	rs2	rs1	funct3	rd	opcode 0
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- opcode: Basic operation $\longrightarrow$ partially specifies what instruction it is
- funct3: Additional opcode
- funct7: Additional opcode $\longrightarrow$ combined with opcode, these fields describe what operation to perform
- rd: Destination register
- rs1: 1st Source register
- rs2: 2nd Source register

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011

7bit opcode
but low 2 bits =11_2

add x9, x20, x21

0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_2 = 0x015A04B3_{16}$

(RISC-V ISA: R-Type Instruction) - List

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

31 … 0

funct7	rs2	rs1	funct3	rd	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

Different encoding in **funct7 + funct3** selects different operations

opcode = 51_{ten}

Question) Why aren't opcode and funct7 and funct3 a single 17-bit field?

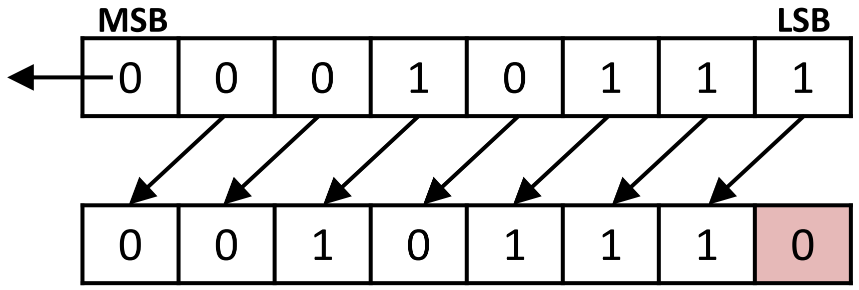
(RISC-V ISA: R-Type Instruction) - Shift

- **SLL (SLR): Shift Logical Left (or Right)**
  - `SLL` (`SRL`) performs left shift (logically) on the value in register (`rs1`) by the *shift amount* held in the register (`rs2`) and stores in (`rd`) register.
  - `SLL:` *The low-order bit (the right-most bit) is replaced by a zero* bit and *the high-order bit (the left-most bit) is discarded.*
  - `SRL:` *The high-order bit (the left-most bit) is replaced by a zero bit and the low-order bit (the Right-most bit) is discarded.*
  - Assembly

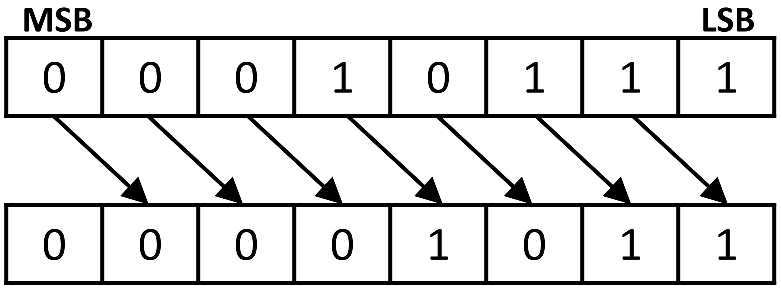
    `Sll (or SRL) rd, rs1, rs2`

Determine the shift amount
→ What is the max amount of shift?
→ Which bits in `rs2` are used for determining the "shift amount"?

The result of **`SLL`**



The result of **`SRL`**

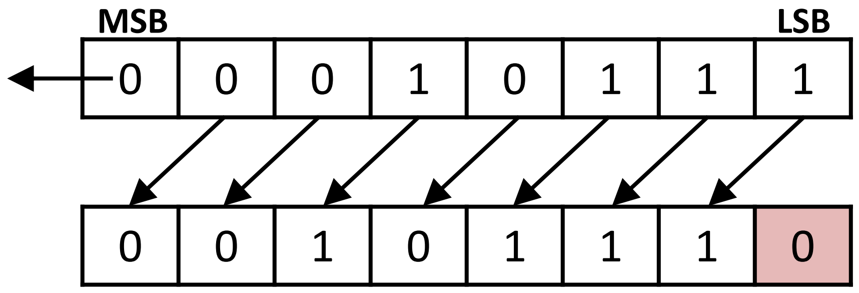


RISC-V ISA: R-Type Instruction - Shift

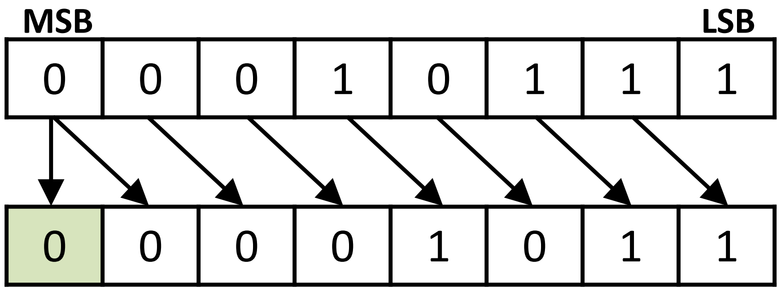
- **SRA: Shift Right Arithmetic**
  - `SRA` performs right shift on the value in register (`rs1`) by the *shift amount* held in the register (`rs2`) and stores in (`rd`) register.
  - The vacated bits at the most significant end are filled with "***sign bit***", known as ***"sign extending".***
  - Assembly

```
SRA rd, rs1, rs2
```

The result of **SLA**



The result of **SRA**



Copy of original sign bit = preserve sign bit

※ Why SLA is not supported?

Appendix: What Is The Meaning Of Bit Shift

- **Left shift "n" : multiply 2^n**
- **Right shift "n" : divide by 2^n**

```
li x5, 4          # x5 ← 4
li x3, 2          # x3 ← 2
sll x1, x5, x3    # x1 ← x5 << x3
```

```
li x5, 7          # x5 ← 7
li x3, 2          # x3 ← 2
sll x1, x5, x3    # x1 ← x5 << x3
```

```
li x5, 4          # x5 ← 4
li x3, 2          # x3 ← 2
srl x1, x5, x3    # x1 ← x5 << x3
```

```
li x5, 7            # x5 ← 7
li x3, 2            # x3 ← 2
srl x1, x5, x3      # x1 ← x5 << x3
```

```
li x5, -4         # x5 ← -4
li x3, 2          # x3 ← 2
sll x1, x5, x3    # x1 ← x5 << x3
```

```
li x5, -4         # x5 ← -4
li x3, 2          # x3 ← 2
srl x1, x5, x3    # x1 ← x5 << x3
```

```
li x5, -4         # x5 ← -4
li x3, 2          # x3 ← 2
sra x1, x5, x3    # x1 ← x5 << x3
```

Find the number stored in the register `x1` (represent in decimal)

(RISC-V ISA: R-Type Instruction) - Compare

- **`SLT`: Set Less Than**
  - `SLT` performs the *signed comparison* between (`rs1`) and (`rs2`) and stores the result in (`rd`).
  - `rd` is "1" if `rs1` < `rs2`, otherwise "0"
  - `SLTU` performs the *unsigned comparison*
  - Assembly

    `SLT(U) rd, rs1, rs2`

```
li x5, 4          # x5 ← 4
li x3, 2          # x3 ← 2
slt x1, x5, x3    # x1 ← x5 < x3
```

What is `x1` ?

What can we know from the below assembly line?

```
SLTU rd, x0, rs2
```

: sets `rd` to 1 if `rs2` is not equal to zero, otherwise sets `rd` to zero

(RISC-V ISA: R-Type Instruction) – Logical Bitwise

- **and**

```
and x9, x10, x11      // reg x9 = reg x10 & reg x11
```

```
x10  00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11  00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9   00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000
```

- **or**

```
or x9, x10, x11 // reg x9 = reg x10 | reg x11
```

```
x10  00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11  00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9   00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000
```

(RISC-V ISA: R-Type Instruction) - Logical Bitwise

- **xor**

```
xor x9, x10, x12 // reg x9 = reg x10 ^ reg x12
```

```
x10   00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x12   11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
x9    11111111 11111111 11111111 11111111 11111111 11111111 11110010 00111111
```

- NOT (→ will be in I-type instruction)
  - xor with 111…1111

(RISC-V ISA: R-Type Instruction) - Example 1

- **How to do the following *C statement*?**

  a = b + c + d - e;

- **Break into multiple instructions**

```
add x10, x1, x2     # a_temp = b + c
add x10, x10, x3    # a_temp = a_temp + d
sub x10, x10, x4    # a = a_temp - e
```

- **A single line of C may turn into several RISC-V instructions**

Constant or Immediate Operands

- **Arithmetic operation that uses a constant number**

  Ex) Adding 4 to a variable

  - Option 1: Memory has "4" stored somewhere. It must be loaded first.

```
var1:  .dword 4
...
ld x9, var1(x3)   # x3 + var1's offset
add x22, x22, x9
```

  - Option 2: Use immediate operands
    - "add immediate", `addi`

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

    - ***Immediate operand instructions are faster (due to no additional memory access) and use less energy***
    - **Constant "4" is included inside the instruction bits**

(RSIC-V ISA: I-Type Instruction) - Format

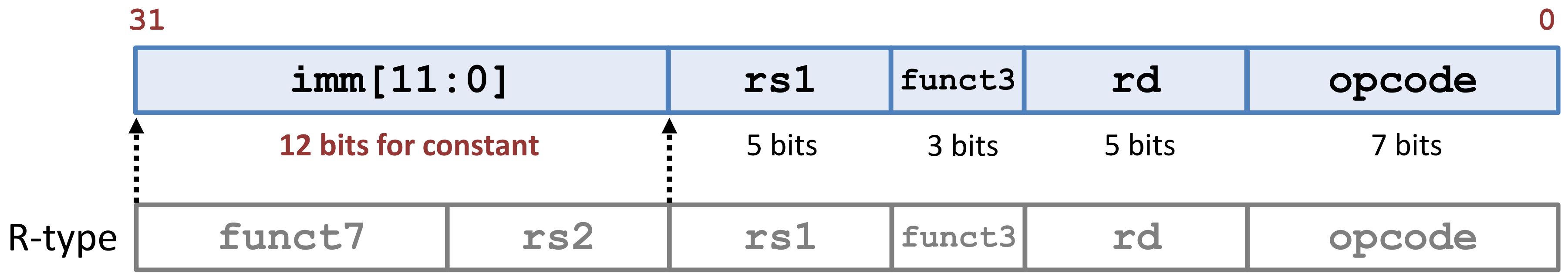
- **Arithmetic operation that uses a constant number**
  - (Ex.) Adding 4 to a variable `A = B + 4`
  - They appear ***frequently in code***, so there are special instructions for them
    - **Recall that** *“Make the common case fast”*
    - Syntax similar to `add` instruction, except that last argument is a number instead of a register
  - Assembly (e.g., register-register addition)

    ADDI rd, rs1, imm$_{12}$

  - Semantics
    - **GPR[rd] ← GPR[rs1] + sign-extend (imm)**
    - **PC ← PC** + 4
  - Variations
    - Arithmetic: {ADD**I**, ~~SUBI~~} X *Why?*
    - Compare: {signed, unsigned} X {SLT**I**(Set if Less Than)}
    - Logic: {AND**I**, OR**I**, XOR**I**}
    - Shift: {SLL**I**, SRL**I**, SRA**I** (Left, Right-Logical, Rigth-Arithmetic)}

(RISC-V ISA: I-Type Instruction) - Format

- **Register-Immediate Instructions**



- `rs2` and `funct7` replaced by 12-bit signed immediate, `imm[11:0]`
- `imm[11:0]` can hold values in range [-2048_{10} , $+2047_{10}$]
- Immediate is always ***sign-extended to 64-bits (i.e., Interpreted as 2's complement number)*** before use in an arithmetic operation.
- We'll later see how to handle immediate > 12 bits

Instruction encoding information

Instruction	Format	immediate	rs1	funct3	rd	opcode
`addi` (add immediate)	I	constant	reg	000	reg	0010011
`ld` (load doubleword)	I	address	reg	011	reg	0000011

$\longrightarrow$ Later slides

(RISC-V ISA: I-Type Instruction) - List

31 … 0

imm[11:0]	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

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

$\underbrace{\text{imm[11:0]}}_{\text{sign-extended}}$

※ SLTIU compares the values as unsigned numbers

000000	shamt	rs1	001	rd	0010011	SLLI
000000	shamt	rs1	101	rd	0010011	SLRI
010000	shamt	rs1	101	rd	0010011	SRAI

$\underbrace{\text{shamt}}_{\text{unsigned}}$

opcode = 19_{ten}

The lower 6 bit of imm_{12} is amount of shift (In RV32I, this field is 5 bit)

(RISC-V ISA: I-Type Instruction) – Example 1

Q) You write "`f=g`" in C statement. How to represent in RISC-V ISA using `add` instruction? (the value of "`g`" was loaded into the register `x3`, and the result "`f`" will be stored into the register `x4`)

```
add x4, x3, x0
```

Q) What can we know from the below assembly line?

```
SLTUI rd, rs1, 1
```

: sets `rd` to 1 if `rs1` is equal to zero, otherwise sets `rd` to zero

Q) What can we know from the below assembly line?

```
XORI rd, rs1, -1
```

: performs a bitwise logical inversion of register `rs1`

(RISC-V ISA: I-Type Instruction) – Example 2

Q) Fill in the each field in RISC-V instruction

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

111111001110	00001	000	01111	0010011
imm. = -50	rs1=1	ADD	rd=15	Opcode

Appendix: Arithmetic Operations @RISC-V

Instruction	Type	Example	Meaning
Add	**R**	`add   rd, rs1, rs2`	`R[rd] = R[rs1] + R[rs2]`
Subtract	**R**	`sub   rd, rs1, rs2`	`R[rd] = R[rs1] – R[rs2]`
Add immediate	**I**	`addi  rd, rs1, imm12`	`R[rd] = R[rs1] + SignExt(imm12)`
Set less than	**R**	`slt   rd, rs1, rs2`	`R[rd] = (R[rs1] < R[rs2])? 1 : 0`
Set less than immediate	**I**	`slti  rd, rs1, imm12`	`R[rd] = (R[rs1] < SignExt(imm12))? 1 : 0`
Set less than unsigned	**R**	`sltu  rd, rs1, rs2`	R[rd] = (R[rs1] $<_u$ R[rs2])? 1 : 0
Set less than immediate unsigned	**I**	`sltiu rd, rs1, imm12`	R[rd] = (R[rs1] $<_u$ SignExt(imm12))? 1 : 0
Load upper immediate	**U**	`lui   rd, imm20`	`R[rd] = SignExt(imm20 << 12)`
Add upper immediate to PC	**U**	`auipc rd, imm20`	`R[rd] = PC + SignExt(imm20 << 12)`

Will be next slides

Appendix: Logical Operations @RISC-V

Instruction	Type	Example	Meaning
AND	**R**	`and  rd, rs1, rs2`	`R[rd] = R[rs1] & R[rs2]`
OR	**R**	`or   rd, rs1, rs2`	`R[rd] = R[rs1] | R[rs2]`
XOR	**R**	`xor  rd, rs1, rs2`	`R[rd] = R[rs1] ^ R[rs2]`
AND immediate	**I**	`andi rd, rs1, imm12`	`R[rd] = R[rs1] & SignExt(imm12)`
OR immediate	**I**	`ori  rd, rs1, imm12`	`R[rd] = R[rs1] | SignExt(imm12)`
XOR immediate	**I**	`xori rd, rs1, imm12`	`R[rd] = R[rs1] ^ SignExt(imm12)`
Shift left logical	**R**	`sll  rd, rs1, rs2`	`R[rd] = R[rs1] << R[rs2]`
Shift right logical	**R**	`srl  rd, rs1, rs2`	`R[rd] = R[rs1] >> R[rs2]` *(logical)*
Shift right arithmetic	**R**	`sra  rd, rs1, rs2`	`R[rd] = R[rs1] >> R[rs2]` *(arithmetic)*
Shift left logical immediate	**I**	`slli rd, rs1, shamt`	`R[rd] = R[rs1] << shamt`
Shift right logical imm.	**I**	`srli rd, rs1, shamt`	`R[rd] = R[rs1] >> shamt` *(logical)*
Shift right arithmetic immediate	**I**	`srai rd, rs1, shamt`	`R[rd] = R[rs1] >> shamt` *(arithmetic)*

Example: Try !!

```
long arith (long x,
            long y,
            long z) {
   long t1 = x + y;
   long t2 = z + t1;
   long t3 = x + 4;
   long t4 = y * 48;
   long t5 = t3 + t4;
   long rval = t2 - t5;
   return rval;
}
```

```
arith:
  add   a5, a0, a1     # a5 = x + y (t1)
  add   a2, a5, a2     # a2 = t1 + z (t2)
  addi  a0, a0, 4      # a0 = x + 4 (t3)
  slli  a5, a1, 1      # a5 = y * 2
  add   a1, a5, a1     # a1 = a5 + y
  slli  a5, a1, 4      # a5 = a1 * 16 (t4)
  add   a0, a0, a5     # a0 = t3 + t4 (t5)
  sub   a0, a2, a0     # a0 = t2 – t5 (rval)
  ret
```

```
x in a0
y in a1
z in a2
```

Example: Try !!

```
long logical (long x,
              long y) {
   long t1 = x ^ y;
   long t2 = t1 >> 17;
   long mask = (1 << 8) - 7;
   long rval = t2 & mask;
   return rval;
}
```

```
logical:
   xor    a0, a0, a1    # a0 = x ^ y (t1)
   srai   a0, a0, 17    # a0 = t1 >> 17 (t2)
   andi   a0, a0, 249   # a0 = t2 & ((1 << 8) - 7)
   ret
```

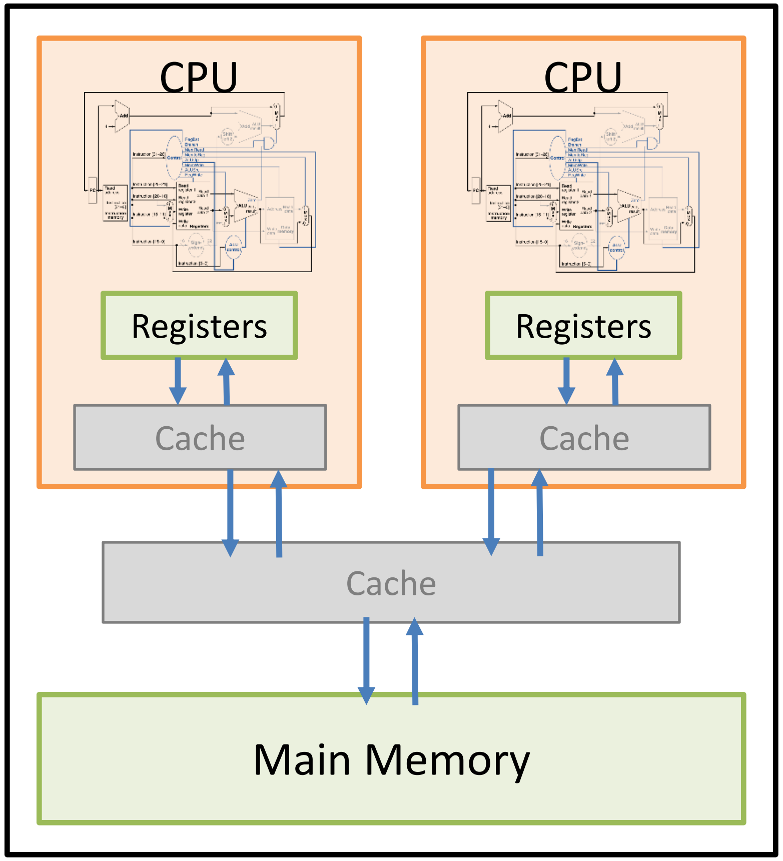
```
x in a0
y in a1
```

Register vs. Memory

- **Access speed (*latency*)**
  - Register > Memory (100~500 times)
- **Capacity**
  - Register (128B or 256B) < Memory (~16GB)
- **Energy consumption**
  - Register < Memory

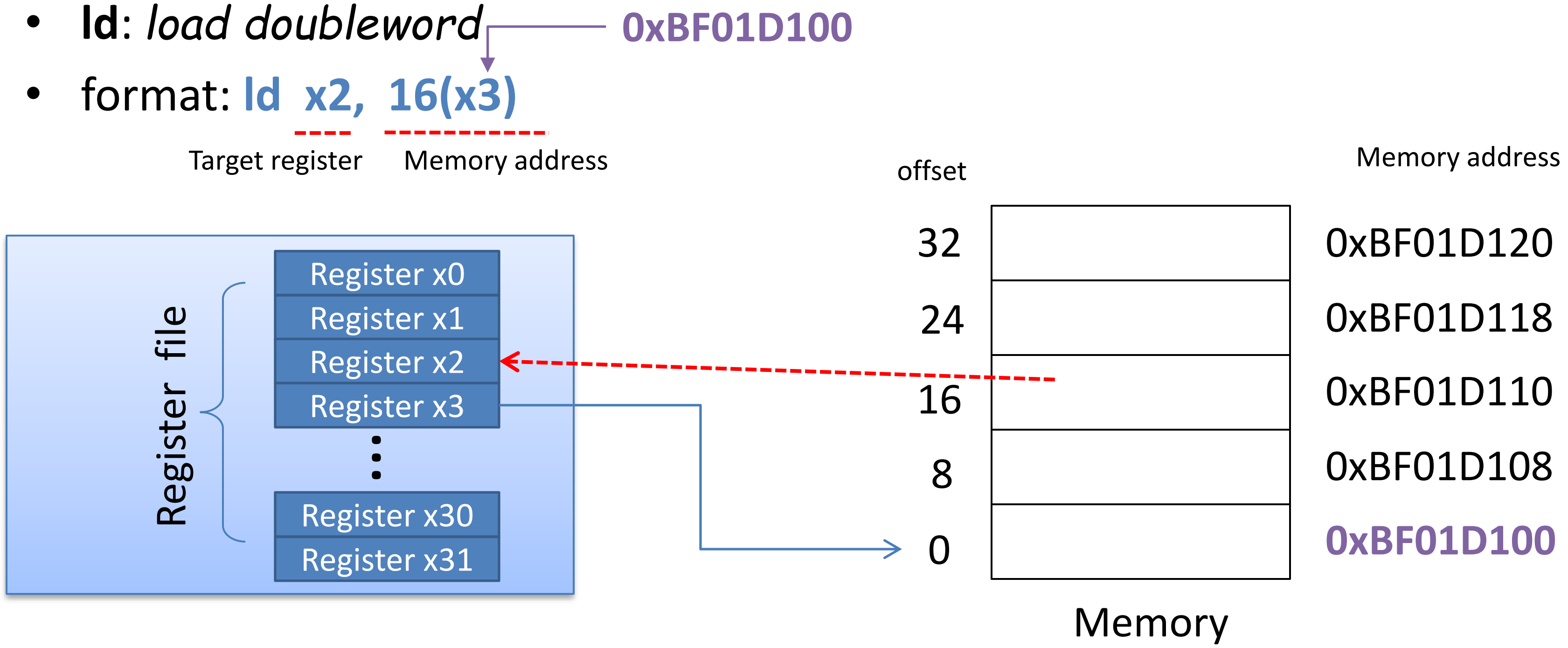
- **Programs most likely have more variables than the number of registers.**

- ***Frequently used variables:***
  - Better to be in register than memory
    - Compilers try to optimize for this
  - *Spilling*: process of putting less commonly used variables into memory



Data Transfer Instruction - Memory Operand

- *Registers are limited,* **and data is usually larger.**
  - ***Arithmetic operations must be done on registers.***
  - Therefore, we need *"data transfer instructions"*
    - To/From memory and registers
- **Transfer (move or copy) data from memory to register: `load`**
  - Address: Location of data element within a memory array
  - **ld**: *load doubleword*
  - format: ld x2, 16(x3)



(RISC-V ISA: I-Type Instruction) – Load

- **load instruction format**
  - Assembly (e.g., load 4-byte word)

    rs1

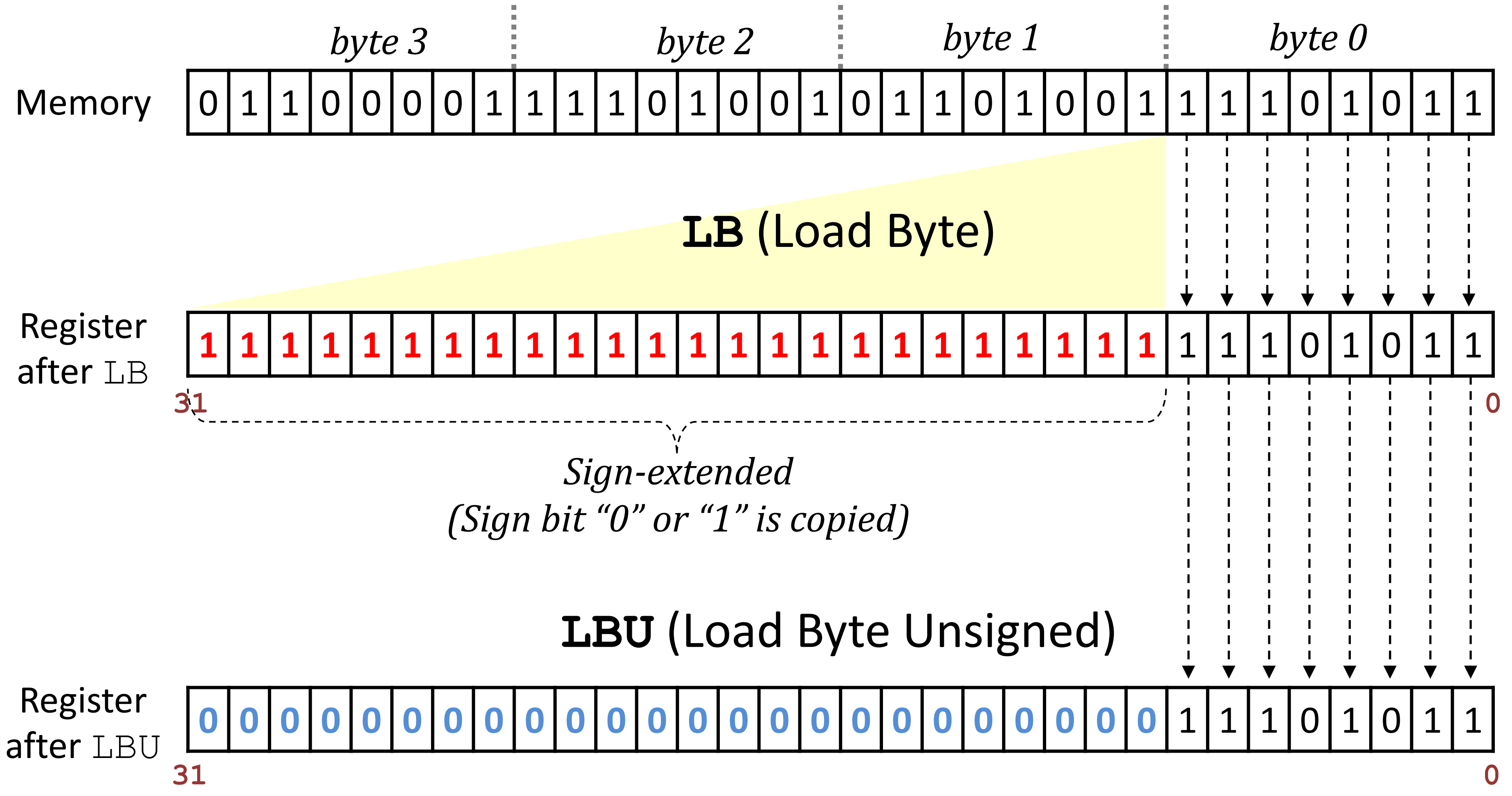
    LW rd, offset$_{12}$(base register)
  - Semantics
    - **byte_address$_{64}$ = sign-extend(offset$_{12}$) + GPR[base]**
    - **GPR[rd] ← Mem$_{64}$[byte_address]**
    - **PC ← PC** + 4
  - Variations

    Transfer unit *Extend type*
    - **LD, LW, LWU, LH, LHU, LB, LBU**
    - (ex.) LB :: GPR[rd] ← sign-extend(MEM$_8$[byte_address])
    - (ex.) LBU :: GPR[rd] ← zero-extend(MEM$_8$[byte_address])

- **LBU** is "load unsigned byte"
- **LH** is "load *halfword*", which loads 16 bits (2 bytes) and ***sign-extends*** to fill destination 64-bit register
- **LHU** is "load unsigned *halfword*", which ***zero-extends*** 16 bits to fill destination 64-bit register
- There is no **LDU** in RV64, because there is no sign/zero extension needed when copying 64 bits from a memory location into a 64-bit register

Appendix: Example of `Load` Instruction

- For simplicity, we assume 32-bit wide register:



(RISC-V ISA: I-Type Instruction) – Load

- `load` Instruction

31				0
imm[11:0]	rs1	funct3	rd	opcode
12 bits for "offset"	5 bits	3 bits	5 bits	7 bits

imm[11:0]→ constant	rs1	funct3	rd	opcode

- 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`

Instruction encoding information

Instruction	Format	immediate	rs1	funct3	rd	opcode
`addi` (add immediate)	I	constant	reg	000	reg	0010011
`ld` (load doubleword)	I	address	reg	011	reg	0000011

(RISC-V ISA: I-Type Instruction) – List of Load

31				0
imm[11:0]	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

imm	rs1	funct3	rd	opcode	
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

opcode = 3_{ten}

funct3 *field encodes size and signedness of load data*

Q) Fill in blank for RISC-V assembly instruction: **`lw x14, 8(x2)`**

000000001000	00010	010	01110	0000011
Imm. = +8	rs1=2	LW	rd=14	LOAD

Q) Can you expect instruction format for LD or LWU? **011 / 110**

imm[11:0]	rs1	???	rd	0000011

(RISC-V ISA: S-Type Instruction) – Store

- **Store instruction format**
  - Assembly (e.g., store 4-byte word)

    SW rs2, offset_{12}(base register)

  - Semantics
    - byte_address$_{64}$ = sign-extend(offset$_{12}$) + GPR[base]
    - Mem$_{64}$[byte_address] ← GPR[rs2]
    - PC ← PC + 4
  - Variations
    - **SD, SW, SH, SB**
    - (ex.) SB :: MEM$_8$[byte_address] ← (GPR[rs2])[7:0]

(RISC-V ISA: S-Type Instruction) – Store

- store Instruction

31					0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

R-type						
R-type	funct7	rs2	rs1	funct3	rd	opcode

- Store needs to read two registers (rs1 for base memory address, and rs2 for data to be stored) + 1 immediate offset.
- Note that store does not 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 !!!***
    - Immediate field is divided into two fields: 7bits + 5bits = 12bits
    - ***Keeping the instruction formats as similar reduces hardware complexity.***
    - Register names more critical than immediate bits in hardware design

Instruction encoding information

Instruction	Format	immed-iate	rs2	rs1	funct3	immed-iate	opcode
sd (store doubleword)	S	address	reg	reg	011	address	0100011

(RISC-V ISA: S-Type Instruction) – List of Store

31					0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

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

opcode = 35_{ten}

funct3 *field encodes size of stored data*

Q) Fill in blank for RISC-V assembly instruction: **sw x14, 8(x2)**

0000000	01110	00010	010	01000	0100011
	rs2=14	rs1=2	word		STORE

000000	01000

Try!! - Example Of Data Transfer Operation

```
addi x11,x0,0x3f5
sw x11,0(x5)
lb x12,1(x5)
```

Answer	x12
RED	0x5
GREEN	0xf
Blue	0x3
YELLOW	0xffffffff

Stored value
in register x5 = a

a+3	Byte [3] = ____________
a+2	Byte [2] = ____________
a+1	Byte [1] = ____________
a+0	Byte [0] = ____________

(Memory)

Example: Transfer From Memory To Register

- **C code**

```
int  A[100];
g = h + A[3];
```

- **Using Load Word (`lw`) in RISC-V:**

```
lw  x10,12(x13)  # Register x10 gets A[3]
add x11,x12,x10  # g = h + A[3]
```

Note: `x13` – base register (pointer to `A[0]`)

12 – offset in bytes

The start address of A[***i***]
= Start address (A[0]) + 4 (data type) x *i* →

Will be in slides of the next 2-4 (data structure)

Example : Transfer From Register To Memory

- C code

```
int  A[100];
A[10] = h + A[3];
```

- Using Store Word (`sw`) in RISC-V:

```
lw   x10,12(x13)     # Temp reg x10 gets A[3]
add  x10,x12,x10     # Temp reg x10 gets h + A[3]
sw   x10,40(x13)     # A[10] = h + A[3]
```

Note: `x13` – base register (pointer to `A[0]`)

12,40 – offset in bytes

`x13`+12 and `x13`+40 must be multiples of "4" because `int` is 32-bit size

Recall – Type And Size of Operands @RISC-V

- **The type of the operand is usually encoded in the *opcode***
  - `ldb` : load byte; `ldw` : load word
- **Common operand types: (imply their sizes)**
  - ***Character*** (8 bits or 1 byte) → *Characters are almost always in ASCII*
  - Half word (16 bits or 2 bytes)
  - Word (32 bits or 4 bytes)
  - Double word (64 bits or 8 bytes)
  - Single precision floating point (4 bytes or 1 word)
  - Double precision floating point (8 bytes or 2 words)

C compiler datatypes for base RISC-V ISA

C type	Description	Bytes in RV32	Bytes in RV64
`char`	Character value/byte	1	1
`short`	Short integer	2	2
`int`	Integer	4	4
`long`	Long integer	4	8
`long long`	Long long integer	8	8
`void*`	Pointer	4	8
`float`	Single-precision float	4	4
`double`	Double-precision float	8	8
`long double`	Extended-precision float	16	16

Example: Understanding SWAP

- Source code in C:

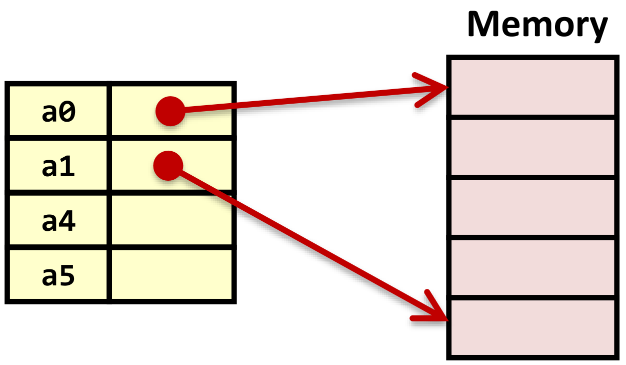
```
void swap(long *xp, long *yp)
{
   long t0 = *xp;
   long t1 = *yp;
   *xp = t1;
   *yp = t0;
}
```

- Corresponding assembly code:

```
swap:
   ld     a4, 0(a0)
   ld     a5, 0(a1)
   sd     a5, 0(a0)
   sd     a4, 0(a1)
   ret
```

Example: Understanding SWAP (1)

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



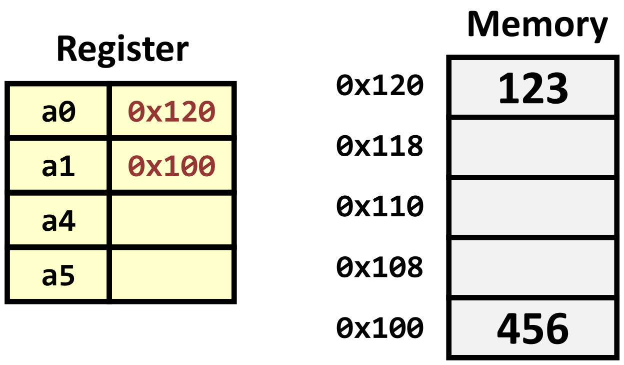
Register Allocation
(By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
    ld      a4, 0(a0)       # t0 = *xp
    ld      a5, 0(a1)       # t1 = *yp
    sd      a5, 0(a0)       # *xp = t1
    sd      a4, 0(a1)       # *yp = t0
    ret
```

Example: Understanding SWAP (2)

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



Register	
a0	0x120
a1	0x100
a4	
a5	

Memory	
0x120	123
0x118	
0x110	
0x108	
0x100	456

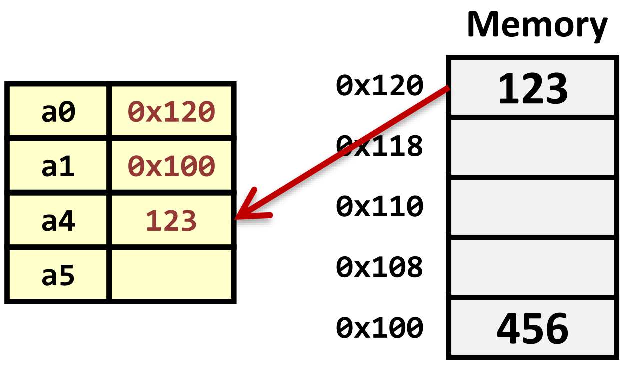
Register Allocation
(By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
   ld     a4, 0(a0)      # t0 = *xp
   ld     a5, 0(a1)      # t1 = *yp
   sd     a5, 0(a0)      # *xp = t1
   sd     a4, 0(a1)      # *yp = t0
   ret
```

Example: Understanding SWAP (3)

```
void swap(long *xp, long *yp)
{
   long t0 = *xp;
   long t1 = *yp;
   *xp = t1;
   *yp = t0;
}
```



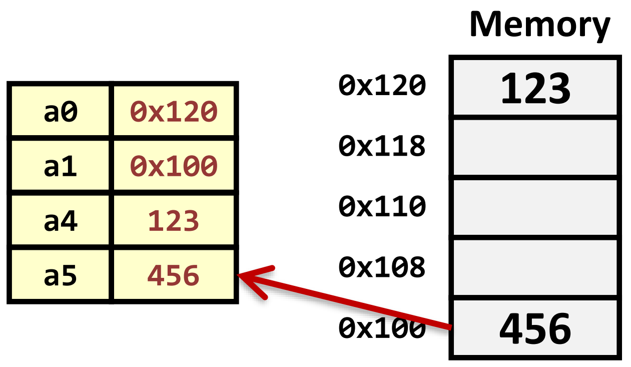
Register Allocation
(By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
    ld     a4, 0(a0)      # t0 = *xp
    ld     a5, 0(a1)      # t1 = *yp
    sd     a5, 0(a0)      # *xp = t1
    sd     a4, 0(a1)      # *yp = t0
    ret
```

Example: Understanding SWAP (4)

```
void swap(long *xp, long *yp)
{
   long t0 = *xp;
   long t1 = *yp;
   *xp = t1;
   *yp = t0;
}
```



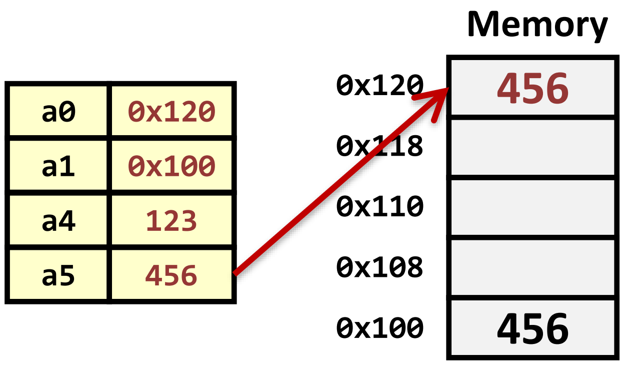
Register Allocation
(By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
   ld      a4, 0(a0)      # t0 = *xp
   ld      a5, 0(a1)      # t1 = *yp
   sd      a5, 0(a0)      # *xp = t1
   sd      a4, 0(a1)      # *yp = t0
   ret
```

Example: Understanding SWAP (5)

```
void swap(long *xp, long *yp)
{
   long t0 = *xp;
   long t1 = *yp;
   *xp = t1;
   *yp = t0;
}
```



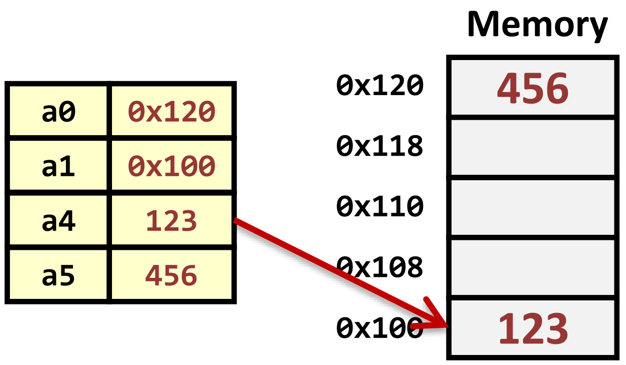
Register Allocation
(By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
   ld     a4, 0(a0)      # t0 = *xp
   ld     a5, 0(a1)      # t1 = *yp
   sd     a5, 0(a0)      # *xp = t1
   sd     a4, 0(a1)      # *yp = t0
   ret
```

Example: Understanding SWAP (6)

```
void swap(long *xp, long *yp)
{
   long t0 = *xp;
   long t1 = *yp;
   *xp = t1;
   *yp = t0;
}
```



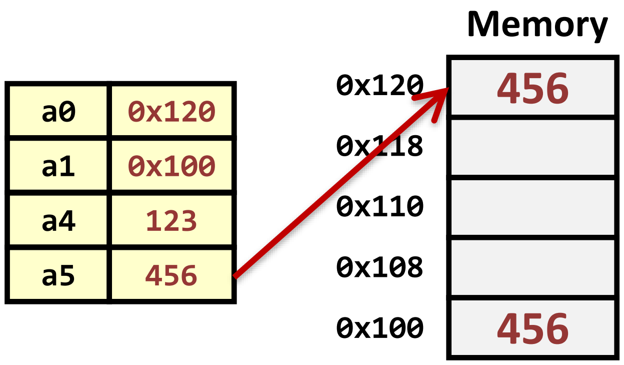
Register Allocation
(By compiler)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
   ld      a4, 0(a0)      # t0 = *xp
   ld      a5, 0(a1)      # t1 = *yp
   sd      a5, 0(a0)      # *xp = t1
   sd      a4, 0(a1)      # *yp = t0
   ret
```

Example: Understanding SWAP (6)

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```



Register Allocation
(By compiler)

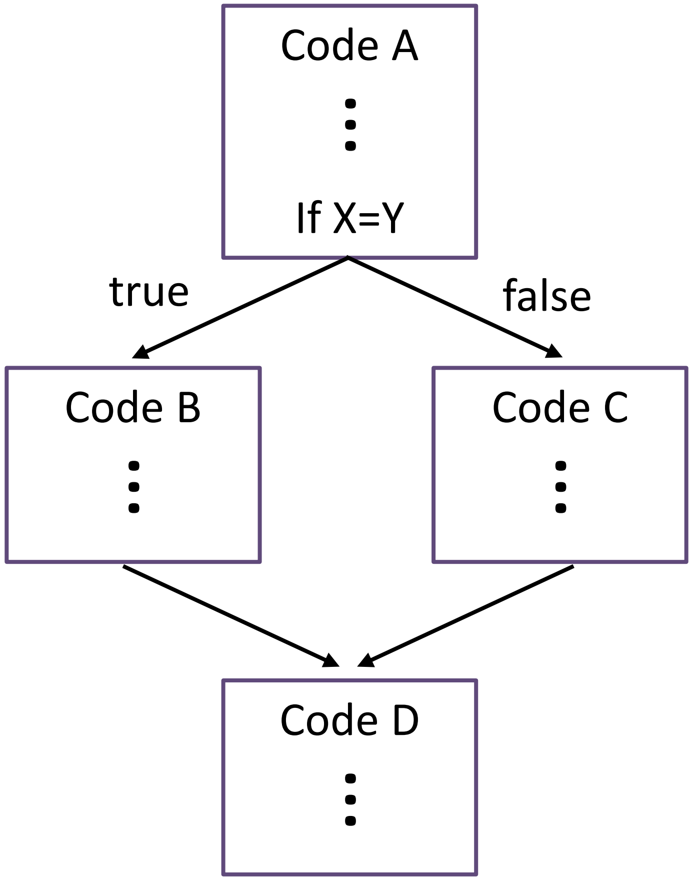
Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1

```
swap:
    ld      a4, 0(a0)      # t0 = *xp
    ld      a5, 0(a1)      # t1 = *yp
    sd      a5, 0(a0)      # *xp = t1
    sd      a4, 0(a1)      # *yp = t0
    ret
```

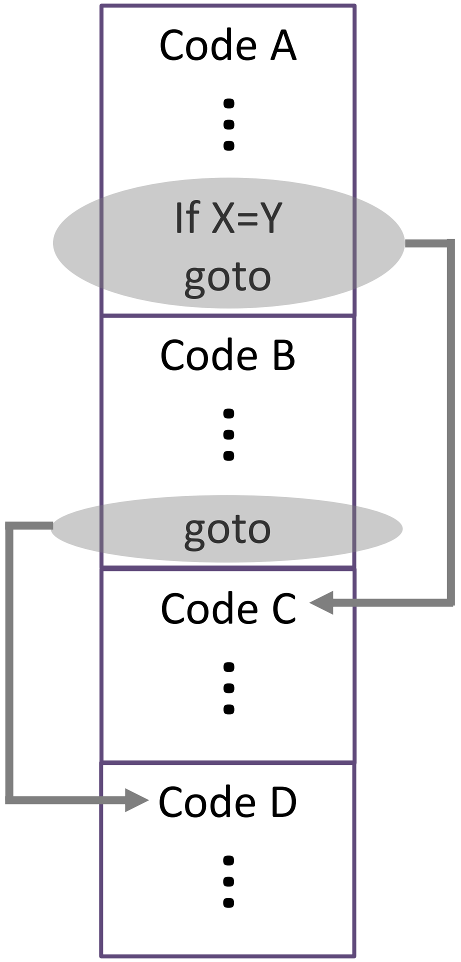
Control Flow Instructions

```
C-Code:

{ code A }

  if X==Y then
      { code B }
  else
      { code C }

{ code D }
```

Control Flow Graph



Assembly Code (linearized)



Types Of Branches

- **Branch:** ***"change of control flow"***

- ***Conditional Branch*** **– change control flow depending on outcome of comparison**
  - branch *if* equal (**`beq`**) or branch *if not* equal (**`bne`**)
  - Also branch if less than (**`blt`**) and branch if greater than or equal (**`bge`**)

- ***Unconditional Branch*** **– always branch**
  - A RISC-V instruction for this*: jump (`j`)*

(RISC-V ISA: SB-Type Instruction) – Cond. Branch

- **Assembly (e.g., branch if equal)**
  - **BEQ rs1, rs2, Label** ← *determined by imm$_{13}$*
  - Branches read ***two (soruce) registers*** but don't write a (destination) register (similar to stores)
  - How to encode label, i.e., where to branch to?

- **Semantics**
  - Target = PC + sign-extended (imm$_{13}$)
  - If GPR(rs1) = GPR (rs2) then PC ← target
    else PC ← PC+4

How far can we jump?

31 … 0

imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Variations
  - **BEQ, BNE, BLT, BGE, BLTU, BGEU**

(RISC-V ISA: SB-Type Instruction) – Cond. Branch

- **Branch instructions**
  - *if* statement

```
beq rs1, rs2, L1
```

  - `beq`: "branch if equal"
  - Go to the statement labeled `L1` if the value in register `rs1` equals the value in register `rs2`

```
bne rs1, rs2, L1
```

  - `bne`: "branch if not equal"

- **`beq`, `bne`: *conditional* branches instructions**

(RISC-V ISA: SB-Type Instruction) – Cond. Branch

- **`blt`: "branch if less than"**

```
blt rs1, rs2, L1
```

  - ***Compare the values (2's complement number) in `rs1` and `rs2`***
  - Branch if `rs1` is smaller
- **`bge`: "branch if greater than or equal"**

```
bge rs1, rs2, L1
```

  - Branch if `rs2` is smaller
- **`bltu, bgeu`**
  - *Numbers in registers are treated* ***as unsigned.***

(RISC-V ISA: SB-Type Instruction) – Cond. Branch

- ***How to encode label, i.e., where to branch to?***

- **Use the `immediate` field as a two's complement offset to "`PC`"**
  - Branches generally change the `PC` by a small amount
    - 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)
  - Can specify ***$\pm 2^{11}$ addresses*** from the `PC` by *12-bit wide immediate field*
    - `Immediate` is # of instructions to move either forward (+) or backwards (–)
  - ***Instructions are "word-aligned" (32-bit wide):***
    - Address is always a multiple of 4 (in bytes)
  - Let `immediate` specify ***#words*** instead of ***#bytes***
    - Instead of specifying $\pm 2^{11}$ bytes from the PC, we will now specify $\pm 2^{11}$ words = $\pm 2^{13}$ byte addresses around PC

► If we don't take the branch: ***PC = PC+4 = next instruction***
► If we do take the branch: ***PC = PC + (immediate*4)***

PC-relative addressing !!!

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 or 64-bit instructions (as used in this class)
- RISC-V conditional ***branches can only reach*** $\pm 2^{10} \times$ ***word (=*** $\pm 2^{12} \times$ ***byte) either side of PC***

(RISC-V ISA: SB-Type Instruction) – Cond. Branch

- (S)B-format is mostly same as SB-Format, with two register sources (`rs1/rs2`) and a 12-bit immediate.
- But now immediate represents values -2^{12} to $+2^{12}-2$ in 2-byte increments ~ $(-2^{11}$ to $+2^{11}-1)$ x 2
  - Can represent branch addresses from -4096 to 4094 in multiple of 2 (i.e. even address)
- ***The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)***

```
bne x10, x11, 2000    // if x10 != x11, go to location 2000 ten = 0 0111 1101 0000 (13bit)
```

0	111110	01011	01010	001	1000	0	1100111
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

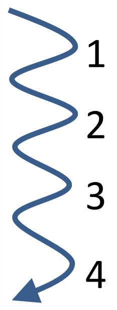
- `imm[0]` is assumed to be always 0, and thus not encoded
- immediate field is 13 bits

See the next example !!!

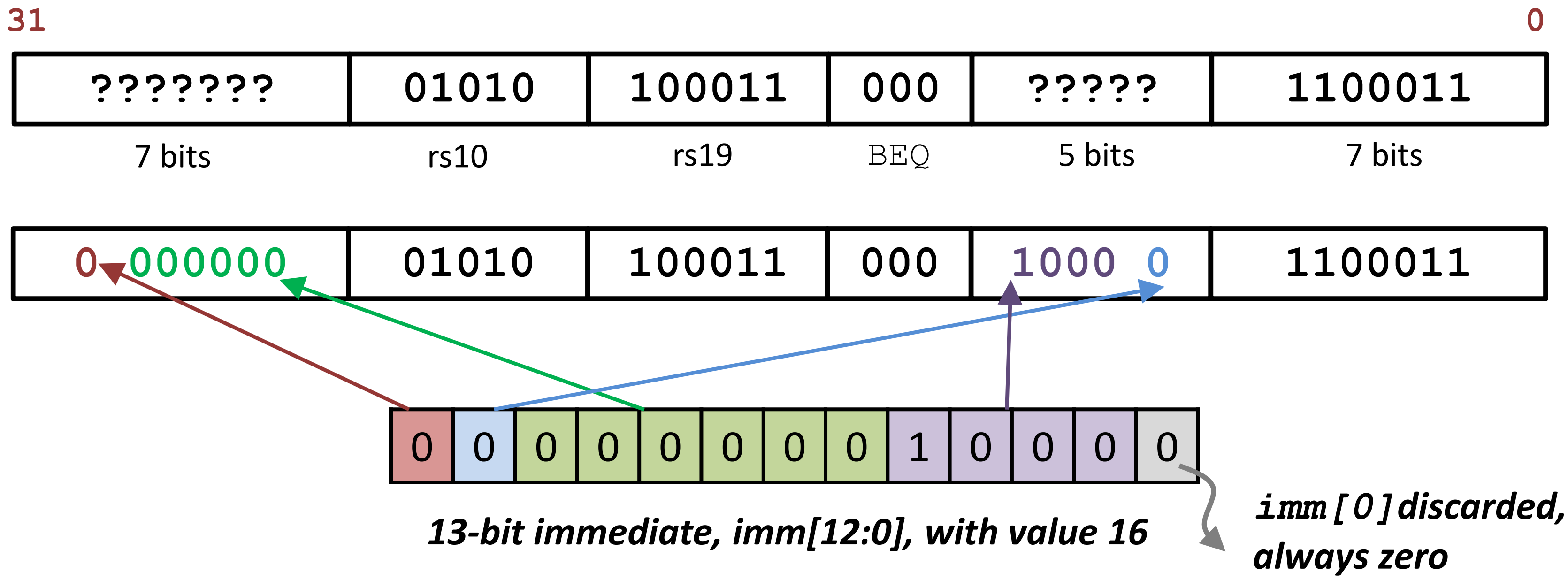
(RISC-V ISA: SB-Type Instruction) – Examples

```
RISCV Code:
L1: beq x19,x10,End
      add x18,x18,x10
      addi x19,x19,-1
      j L1
End: <target instruction>
```

Start counting from instruction "AFTER" the branch



Branch offset = 4 X word(32-bit instructions) = 16 bytes
(Branch with offset of 0, branches to itself)



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

imm[0] discarded, always zero

(RISC-V ISA: SB-Type Instruction) – List

31 … 0

imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

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

opcode = 99_{ten}

31 … 0

imm[12]	imm[10:5]	rs2	rs1	000	imm[4:1]	imm[11]	opcode
1 bits	6 bits	5 bits	5 bits	3 bits	4 bits	1 bits	7 bits

(RISC-V ISA: SB-Type Instruction) – Question

- Does the value in branch immediate field change if we move the code?
  - If moving individual lines of code, then yes (why?)
  - If moving all of code, then no (why? - because PC-relative offsets)

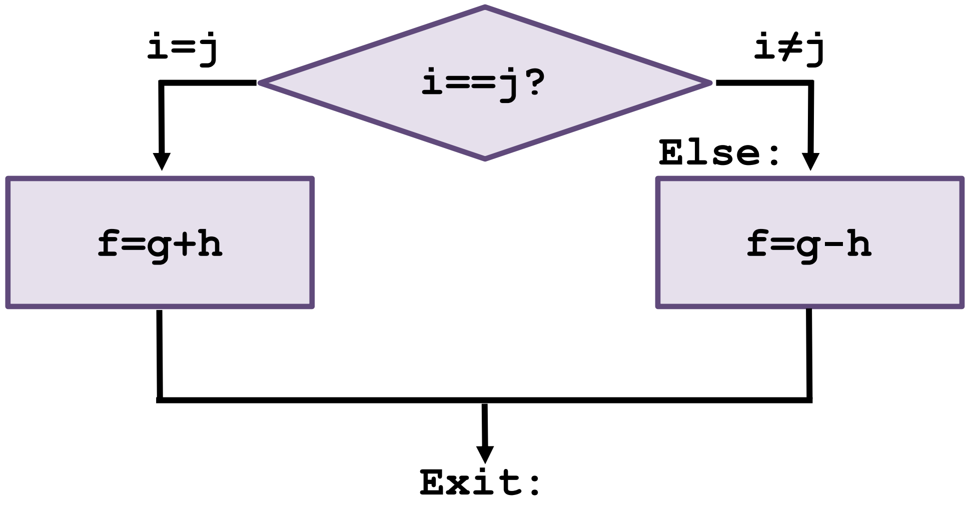
- What do we do if destination is > 2^{10} instructions away from
  - ***Other instructions*** save us:

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

$\Rightarrow$

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

Example: Compiling `If` Statements



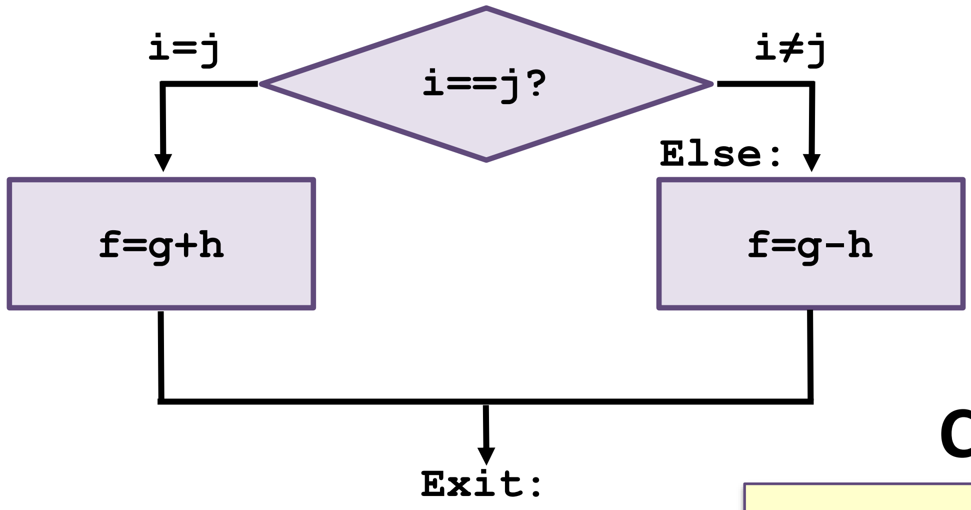
C code:

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

Pseudo code:

```
if (i != j) goto Else;
   f = g + h;
   goto Exit:
Else:
   f = g - h;
Exit:
```

Example: Compiling `If` Statements



C code:

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

Compiled RISC-V code:

```
// i in x22, j in x23
// f in x19, g in x20, h in x21

   bne  x22, x23, L1
   add  x19, x20, x21
   beq  x0, x0, Exit // unconditional
 L1:   sub  x19, x20, x21
 Exit: ...
```

Assembler calculates addresses

Example: Compiling Loop Statements

C code:

Assume `double A[i]`

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

```
Loop:
  if (A[i] != k) goto Exit;
  i += 1;
  goto Loop;
Exit:
```

Example: Compiling Loop Statements

C code:

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

Pseudo code:

```
Loop:
  if (A[i] != k) goto Exit;
  i += 1;
  goto Loop;
Exit:
```

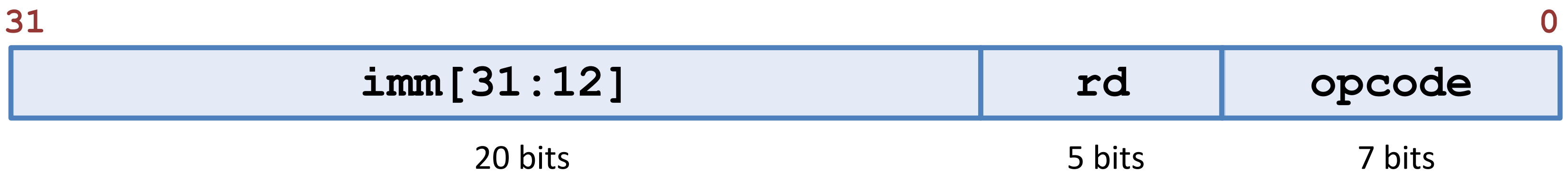
Compiled RISC-V code:

```
// i in x22, k in x24
// address of A[] in x25
// A[] store “double” number

Loop: slli  x10, x22, 3
      add   x10, x10, x25
      ld    x9, 0(x10)
      bne   x9, x24, Exit
      addi  x22, x22, 1
      beq   x0, x0, Loop
 Exit: ...
```

Dealing With Large Immediate: U-type Instruction

- There are times when constants are too big to fit into 12 bits
- **How do we deal with 32-bit immediate**
  - Our **I-type** instructions *only give us 12 bits.*
  - Need a new instruction format for dealing with the rest of the 20 bits

- **RISC-V provides `lui` (*Load Upper Immediate*) instruction**
  - U-type instruction



  - ***Load 20 bit constant into bit position 31~12 from a register***
  - Left 32 bits are extended with 31th bit value
  - Rightmost 12 bits are filled with ***zeros***

(RISC-V ISA: U-Type Instruction) – Example

- **`LUI` to create "long (32bit-wide)" immediate**
  - `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`).

► ***How do we make this value in a register?***

```
00000000 00000000 00000000 00000000 00000000 00111101 00000101 00000000
```

(1) `lui` x19, 976 // 976 = 0000 0000 0011 1101 0000

```
0000000 00000000 00000000 00000000|00000000 00111101 00000000 00000000
```

(2) `addi` x19, x19, 1280 // 1280 = 0101 00000000

```
00000000 00000000 00000000 00000000|00000000 00111101 00000101 00000000
```

Another Example:

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

(RISC-V ISA: U-Type Instruction) – 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 is set, will subtract -1 from upper 20 bits

(Solution)

Pre-increment value placed in upper 20 bits, if sign bit will be set on immediate in lower 12 bits.

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

► ***Assembler pseudo-insructions handles all of this:***

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

Appendix: AUIPC

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

```
Label: AUIPC x10, 0    # Puts address of label in x10
```

(RISC-V ISA: UJ-Type Instruction) – Jump & Link

- **Assembly**
  - **JAL   rd,   imm$_{21}$**   (imm[0] = 0, implicitly)
  - **JAL saves PC+4 in register rd (the return address, usually x1)**
    - ***Branch unconditionally and save the address of the next instruction (return address) to the designated register***
    - Assembler "j" jump is ***pseudo-instruction***, uses JAL but sets rd=x0 to discard return address
  - Target somewhere within ±2^{19} locations, 2 bytes apart (similar to branch)
    - **±2^{18} x 32-bit instructions (word) → ±2^{20} byte address**
- **Semantics**
  - Target = PC + sign-extended (imm$_{21}$)
  - GPR(rd) ← PC+4 (return address), PC ← target

31 … 0

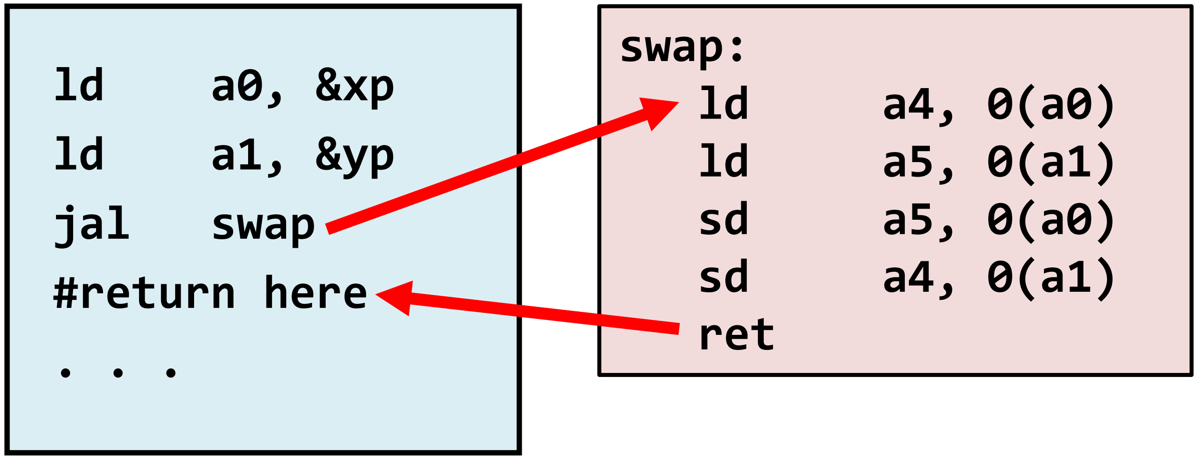
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1 bits	10 bits	1 bits	8 bits	5 bits	7 bits

U-type	imm[31:12]	rd	opcode

Function Call and Return

- **Suppose we call function `swap()`**
  - Pass pointer of `&xp` and `&yp` as arguments
  - `main()` call the function `swap` saving return address in `x1` = `ra`
  - Execute function `swap`
  - Return control to the point of origin (return address)

Register	Variable
a0	xp
a1	yp
a4	t0
a5	t1
x1(ra)	ret addr



```
ld     a0, &xp
ld     a1, &yp
jal    swap
#return here
. . .
```

```
swap:
   ld      a4, 0(a0)
   ld      a5, 0(a1)
   sd      a5, 0(a0)
   sd      a4, 0(a1)
   ret
```

Appendix: JALR (I-Format)

- **Assembly**
  - **JALR rd, rs1(base), imm$_{12}$**
  - Writes next instruction (`PC + 4`) to `rd` / Set `PC = rs1 + imm`$_{12}$
  - Uses same `immediate` as arithmetic and loads
    - No multiplication by 2 bytes

imm[11:0]	rs1	funct3	rd	opcode
12 bits for "offset"	5 bits	3 bits	5 bits	7 bits

(bit 31 at left, bit 0 at right)

- **Jumping far away**
  - RISC-V allows very long jumps to any 32-bit address by using `jarl`
    - `lui` write bits 12~31 of the address to a temporary register
    - `jalr` adds the lower 12 bits of the address to the register and jumps to the sum

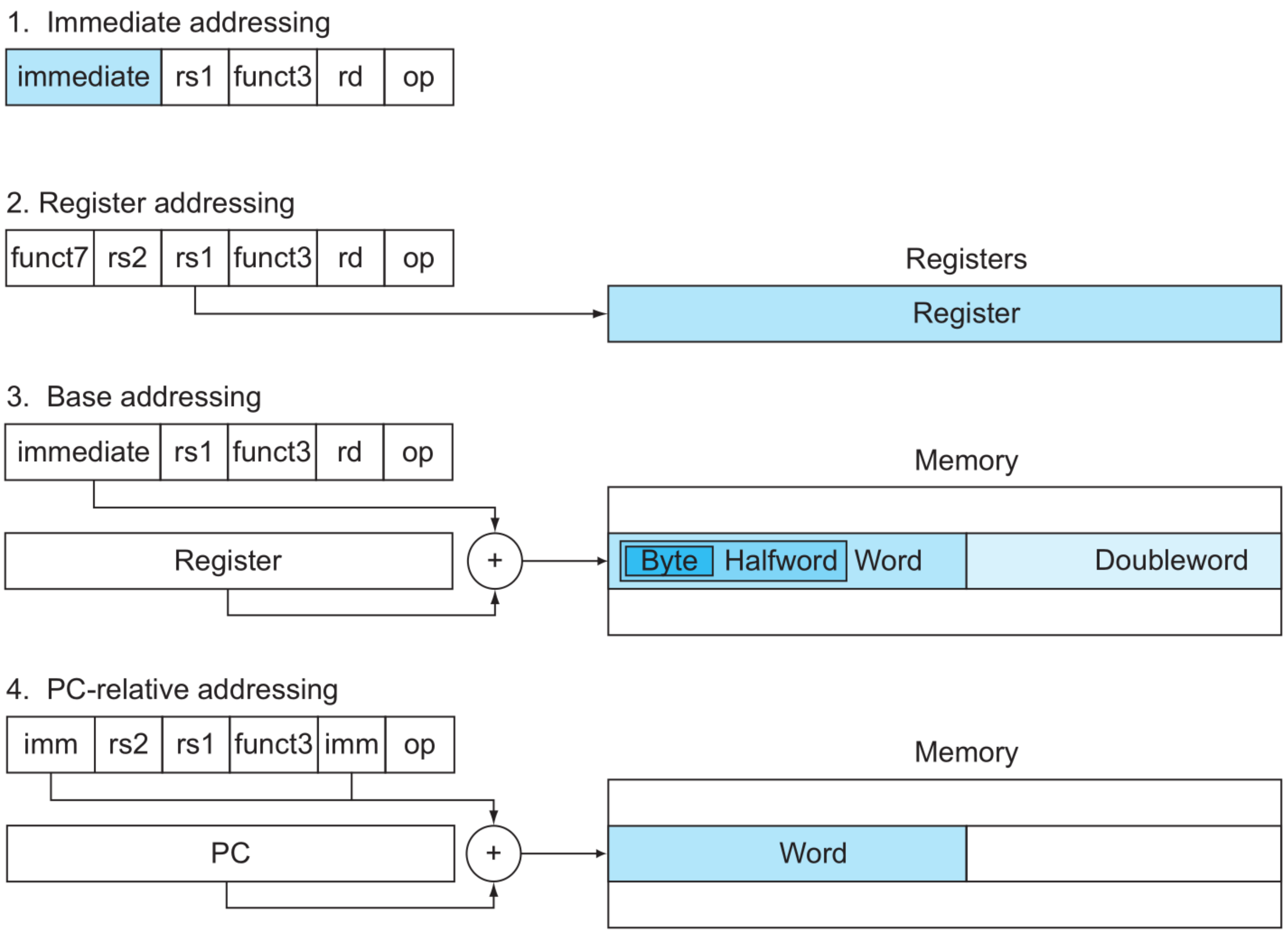
```
beq x10, x12, L1
```

⇒

```
     lui  x9, L1_U20
     addi x9,x9, L1_L12
     bne  x10,x12, L2
     jalr x0, 16(x9)
L2:
```

Summary: RISC-V Addressing Modes

- **Addressing mode: How to locate data needed as operands**



Summary: PC-relative Addressing

- **Implication of immediate field size**
  - Conditional branch: 13 bits
  - Unconditional branch: 21 bits

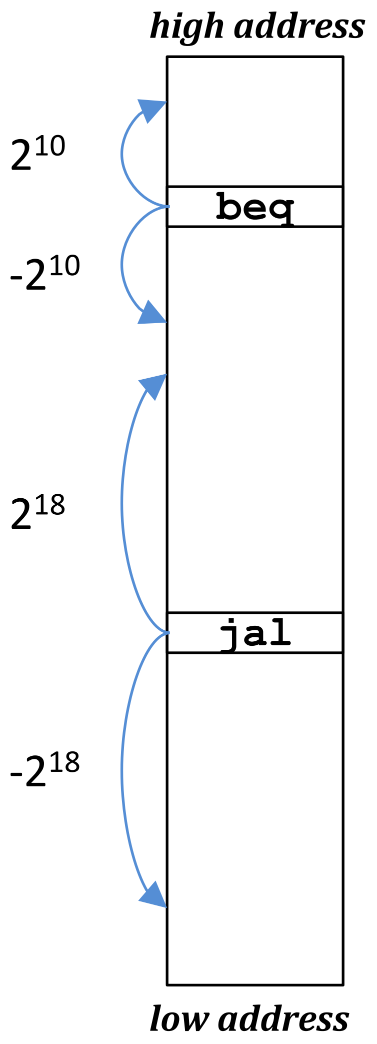
  → Program size = $2^{\text{immediate\_field\_size}}$
- **new PC ← register + branch offset**

  PC
  - Allow program to be 2^{64} and use branches
- **PC-relative addressing enables:**
  - Branch within $\pm 2^{10}$ instructions
  - Jumps within $\pm 2^{18}$ instructions
- **Branching far away**

```
beq x10, x0, L1
```

→

```
      bne x10, x0, L2
      jal x0, L1
L2:
```


high address
2^{10}
beq
-2^{10}
2^{18}
jal
-2^{18}
low address