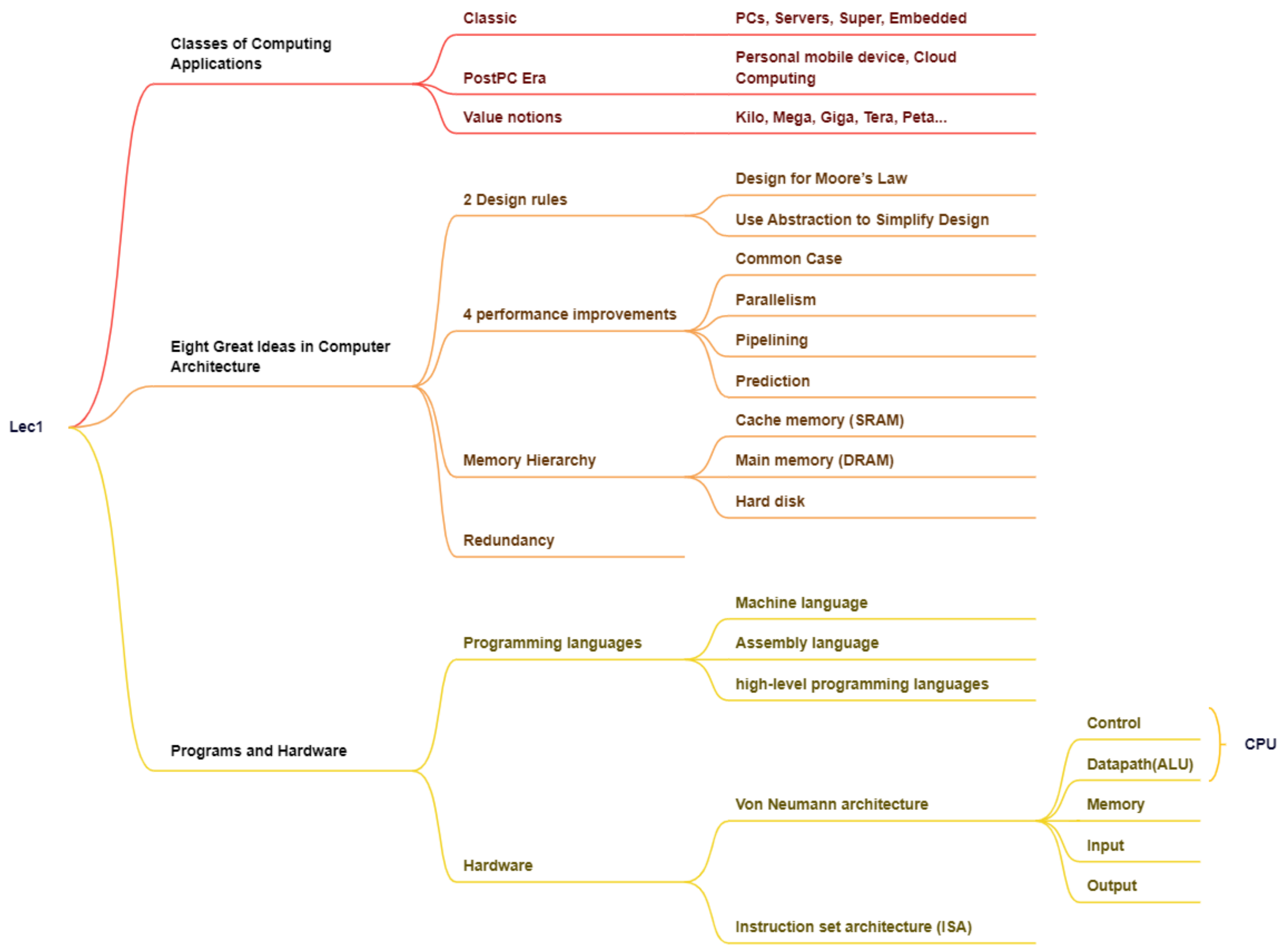
# COMPUTER ORGANIZATION

## Lecture 2
## RISC-V Introduction

2024 Spring

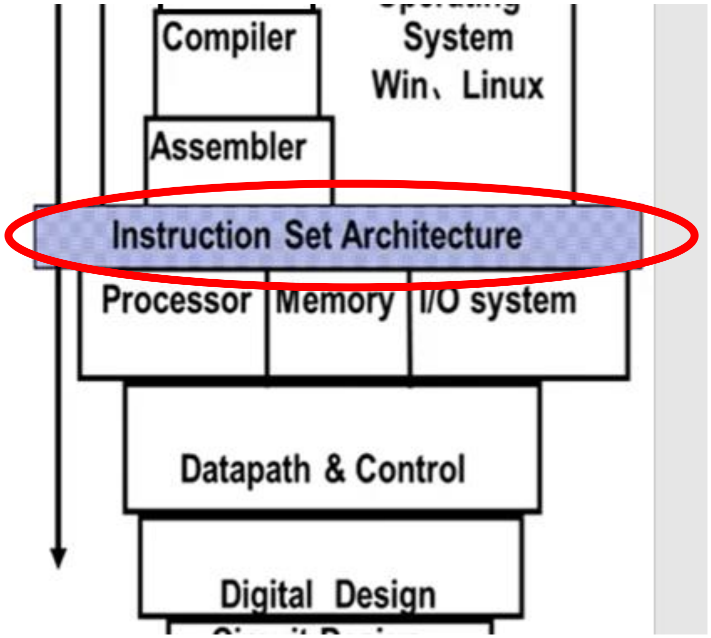
## Recap

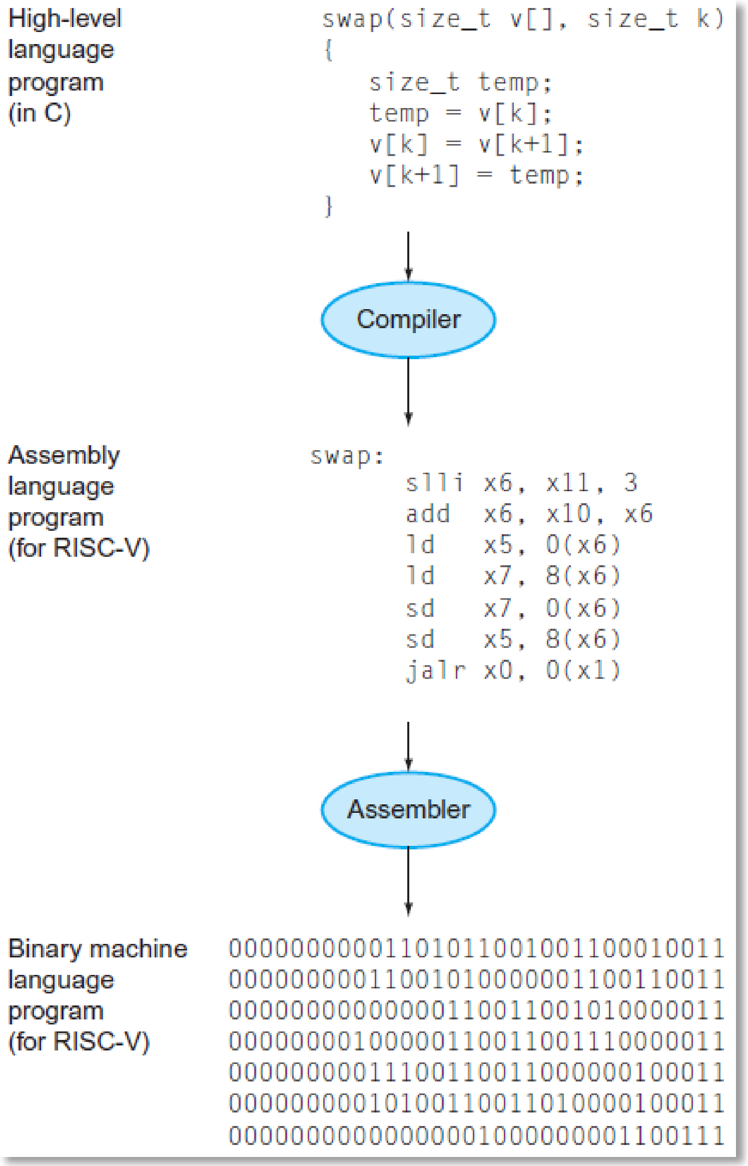

Lec1
Classes of Computing Applications
Classic
PCs, Servers, Super, Embedded
PostPC Era
Personal mobile device, Cloud Computing
Value notions
Kilo, Mega, Giga, Tera, Peta...
Eight Great Ideas in Computer Architecture
2 Design rules
Design for Moore's Law
Use Abstraction to Simplify Design
4 performance improvements
Common Case
Parallelism
Pipelining
Prediction
Memory Hierarchy
Cache memory (SRAM)
Main memory (DRAM)
Hard disk
Redundancy
Programs and Hardware
Programming languages
Machine language
Assembly language
high-level programming languages
Hardware
Von Neumann architecture
Control
Datapath(ALU)
CPU
Memory
Input
Output
Instruction set architecture (ISA)

## Abstractions

- Computer Architecture
  - Instruction Set Architecture
  - Machine Organization

- ISA => Assembly Language
  - From programmer's point of view





High-level language program (in C)
Compiler
Assembly language program (for RISC-V)
Assembler
Binary machine language program (for RISC-V)

```
swap(size_t v[], size_t k)
{
    size_t temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

```
swap:
      slli x6, x11, 3
      add  x6, x10, x6
      ld   x5, 0(x6)
      ld   x7, 8(x6)
      sd   x7, 0(x6)
      sd   x5, 8(x6)
      jalr x0, 0(x1)
```

```
00000000001101011001001100010011
00000000011001010000001100110011
00000000000000110011001010000011
00000000100000110011001110000011
00000000011100110011000000100011
00000000010100110011010000100011
00000000000000001000000001100111
```

## Instruction Set Architecture (ISA)

- Instructions: CPU’s primitive operations
  - Instructions performed one after another in sequence
  - Each instruction does a small amount of work (a tiny part of a larger program).
  - Each instruction has an operation applied to operands,
  - and might be used to change the sequence of instructions.
- CPUs belong to “families,” each implementing its own set of instructions
- CPU’s particular set of instructions implements an Instruction Set Architecture (ISA)
  - Examples: ARM, Intel x86, MIPS, RISC-V, PowerPC...

## Instruction Set Architecture

- CISC
  - Complex Instruction Set Computer
  - Variable instruction length
  - Much more powerful instructions
  - Hardware intensive instructions (more transistors
  - e.g. x86
- RISC
  - Reduced Instruction Set Computer
  - Fixed instruction size
  - Simple instructions (load/store)
  - Emphasizes more on software (compiler)
  - e.g. MIPS, ARM, PowerPC, RISC-V

## Which ISAs “win”

- The big winners: x86/x64 (servers) and Arm (phones/embedded)
  - Neither are the cheapest nor the best architectures available...
  - They won because of the legacy ecosystem
- But since our focus is understanding how computers work, we choose learning RISC-V
- Learn to program in assembly language, e.g. RISC-V
  - Best way to understand what compilers do to generate machine code
  - Best way to understand what the CPU hardware does

## And the Road To Future Classes

- CS302 Operation Systems
  - OS needs a small amount of assembly for doing things the "high level" language doesn't support
    - Such as accessing special resources
- CS323 Compilers
  - Learn how to build compilers. A compiler goes from source code to assembly language.
- CS301 Embedded System
  - Assembly or a combination of high-level languages and inline assembly are commonly used to achieve efficient execution in resource-constrained environments.
- CS315 Computer Security
  - Exploit code ("shell code") is often in assembly and exploitation often requires understanding the assembly language & calling-convention of the target

## What is RISC-V

- Fifth generation of RISC design from UC Berkeley
- A high-quality, license-free, royalty-free RISC ISA specification
  - Implementers do not pay any royalties
  - Large community of users riscv.org: industry, academia
  - Full software stack
- Appropriate for all levels of computing system, from microcontrollers to supercomputers
  - 32-bit, 64-bit, and 128-bit variants (we’re using 32-bit(RV32) in lectures and labs, textbook uses 64-bit)
- Standard maintained by non-profit RISC-V Foundation

## A Basic Assembly Instruction

- C code: a = b + c ;
- Assembly code: (human-friendly machine instructions)

```
add   a, b, c      #  a is the sum of b and c
```

- Machine code: (hardware-friendly machine instructions)

```
0000 0000 1100 0101 1000 0101 0011 0011
```

- Translate the following C code into assembly code:

  a = b + c + d + e;

```
add  a, b, c              add  a, b, c
add  a, a, d      or      add  f, d, e
add  a, a, e              add  a, a, f
```

  - Instructions are simple: fixed number of operands (unlike C)
  - A single line of C code is converted into multiple lines of assembly code
  - Some sequences are better than others… the second sequence needs one more (temporary) variable  f

# A Basic Assembly Instruction

- In Previous example
  - add   a, b, c
  - All arithmetic operations have this form
- *Design Principle 1: Simplicity favors regularity*
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost
- Example
  - C code: f = (g + h) - (i + j);
  - Assembly code:

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

## Assembly Variables: Registers

- Unlike HLL like C or Java, assembly does not have variables
- Assembly language operands are objects called registers
  - Limited number of special places to hold values, built directly into the hardware
  - 32 registers in RISC-V
  - Each RISC-V register is 32 bits wide called a “word” (RV32 variant of RISC-V ISA)
  - Registers have **no type**
  - Operation determines how register contents are interpreted
- *Design Principle 2: Smaller is faster*
  - registers: 32
  - main memory: millions of locations

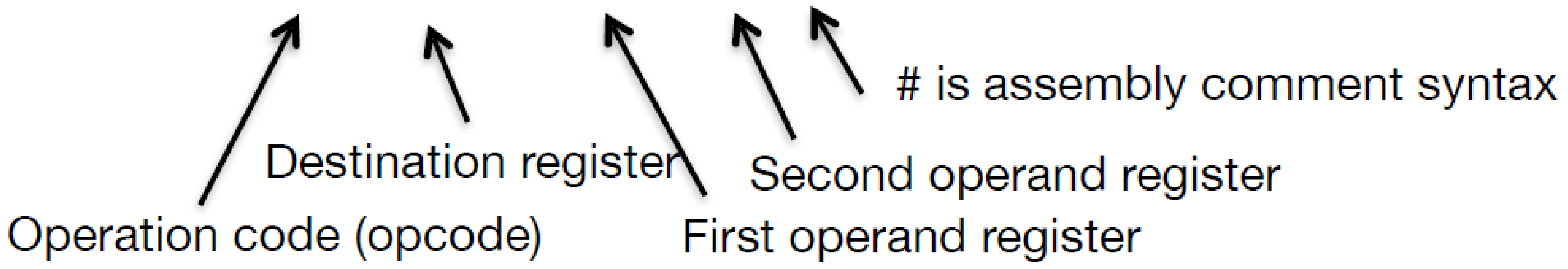
# RISC-V Registers

- x0 is special, always holds the value zero and can't be changed

| Register | Alternative name | Description |
|---|---|---|
| x0 | zero | the constant value 0 |
| x1 | ra | Return address |
| x2 | sp | Stack pointer |
| x3 | gp | Global pointer |
| x4 | tp | Thread pointer |
| x5 – x7 | t0 – t2 | Temporaries |
| x8 | s0/fp | Saved register/Frame pointer |
| x9 | s1 | Saved register |
| x10-17 | a0-7 | Function arguments/Return values |
| x18-27 | s2-11 | Saved registers |
| x28-31 | t3-6 | Temporaries |

# RISC-V Instructions

- Instructions have an opcode and operands
- `E.g., add x1, x2, x3 # x1 = x2 + x3`



- Instructions are fixed, 32bit long (machine code)
  - Note: Conversions between assembly to corresponding machine code will be taught in future lecture
- Each instruction uses one of these predefined formats:

| Name | Field | | | | | | Comments |
|---|---|---|---|---|---|---|---|
| (Field Size) | 7 bits | 5 bits | 5 bits | 3 bits | 5 bits | 7 bits | |
| R-type | funct7 | rs2 | rs1 | funct3 | rd | opcode | Arithmetic instruction format |
| I-type | immediate[11:0] | | rs1 | funct3 | rd | opcode | Loads & immediate arithmetic |
| S-type | immed[11:5] | rs2 | rs1 | funct3 | immed[4:0] | opcode | Stores |
| SB-type | immed[12,10:5] | rs2 | rs1 | funct3 | immed[4:1,11] | opcode | Conditional branch format |
| UJ-type | immediate[20,10:1,11,19:12] | | | | rd | opcode | Unconditional jump format |
| U-type | immediate[31:12] | | | | rd | opcode | Upper immediate format |

## Arithmetic Operations

- Addition:
  - Example: `add x1,x2,x3` (in RISC-V)
  - Equivalent to: `a = b + c` (in C) , where a, b, c in x1,x2,x3
- Subtraction
  - Example: `sub x3,x4,x5` (in RISC-V)
  - Equivalent to: `d = e - f` (in C), where d, e, f in x3, x4, x5
- Example: how to do the following C statement?

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

  - f, …, j in x19, x20, …, x23
  - Break into multiple instructions:

  ```
  add x5, x20, x21    # temp t0 = g + h
  add x6, x22, x23    # temp t1 = i + j
  sub x19, x5, x6     # f = t0 - t1
  ```

## Register x0

- Very useful: always holds zero and can never be changed (does not require initialization)
- Ex: Copy a value from one register to another:

```
add x3,x4,x0 (in RISC-V)
same as
f = g (in C)
```

- Or, whenever a value is produced and we want to throw it away, write to x0:
- By convention RISC-V has a specific no-op instruction `add x0,x0,x0`
- Also, we will see x0 used later with "jump-and-link" instruction

## Immediates

- Immediates are used to provide numerical constants
- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids loading from memory
- Syntax similar to add instruction, except that last argument is a number instead of a register
- Example: Add Immediate:

```
addi x3,x4,10
same as
f = g + 10 (in C)
```

```
addi x3,x4,0
same as
f = g (in C)
```

- No subtract immediate instruction, why?

## Recall: Numeric Representations

- Decimal $35_{10}$ or $35_{ten}$
- Binary $00100011_2$ or $00100011_{two}$
- Hexadecimal 0x23 or $23_{hex}$

  0-15 (decimal) → 0-9, a-f (hex)

## Recall: Numeric Representations

- Unsigned Binary 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: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 1\times 2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

## Signed Numeric Representations

- 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
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

## Signed Numeric Representations

- Signed Negation
- Complement and add 1
  - Complement means $1 \rightarrow 0$, $0 \rightarrow 1$

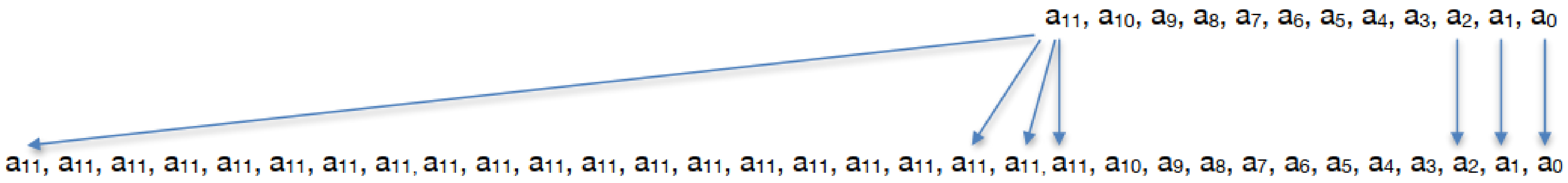
$$x + \bar{x} = 1111...111_2 = -1$$

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

- Example: negate +2
  - $+2 = 0000\ 0000 \ldots 0010_2$
  - $-2 = 1111\ 1111 \ldots 1101_2 + 1$
    $= 1111\ 1111 \ldots 1110_2$

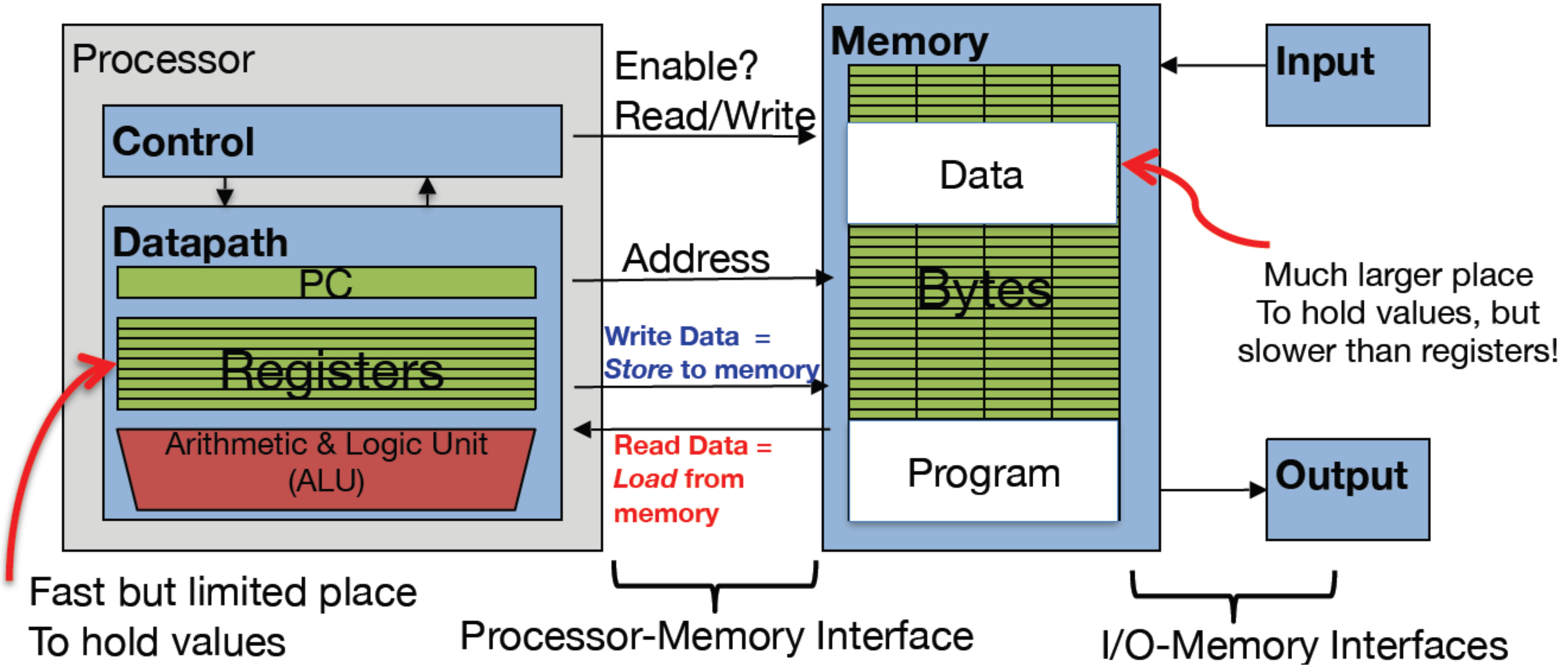
# Immediates & Sign Extension

- Immediates are necessarily small
  - An I-type instruction can only have 12 bits of immediate (We'll see more details in future lecture)
- In RISC-V immediates are "sign extended"
  - So the upper bits are the same as the top bit
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110
- So for a 12bit immediate...
  - Bits [31:12] get the same value as Bit 11


$a_{11}, a_{10}, a_9, a_8, a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0$
$a_{11}, a_{11}, a_{11}, a_{11}, a_{11}, a_{11}, a_{11}, a_{11}, a_{11}, a_{11}, a_{11}, a_{11}, a_{11}, a_{11}, a_{11}, a_{11}, a_{11}, a_{11}, a_{11}, a_{11}, a_{11}, a_{10}, a_9, a_8, a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0$

## Data Transfer Operations

- Registers vs. Memory
  - Arithmetic operations can only be performed on registers
  - Thus, the only memory actions are loads & stores

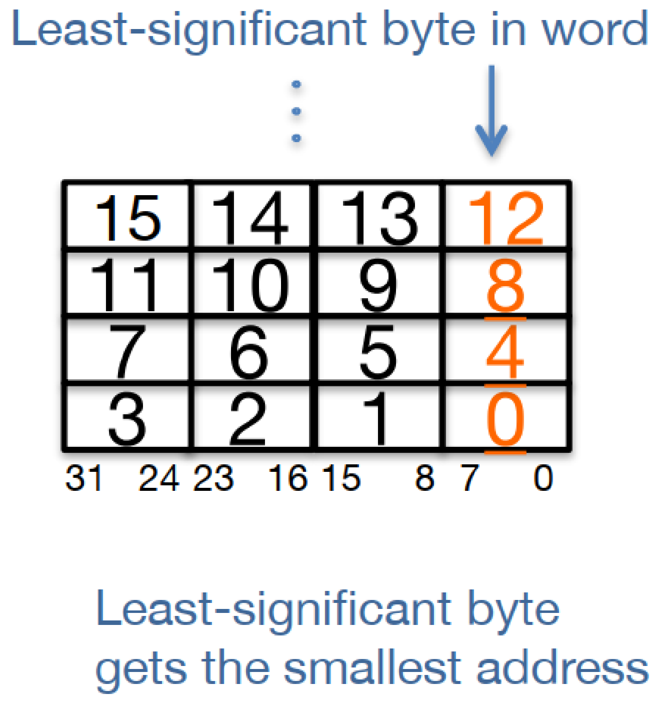

Processor
Control
Datapath
PC
Registers
Arithmetic & Logic Unit (ALU)
Enable?
Read/Write
Address
Write Data = Store to memory
Read Data = Load from memory
Memory
Data
Bytes
Program
Input
Output
Much larger place To hold values, but slower than registers!
Fast but limited place To hold values
Processor-Memory Interface
I/O-Memory Interfaces

## Speed of Registers vs. Memory

- Registers vs. Memory
  - Arithmetic operations can only be performed on registers
  - Thus, the only memory actions are loads & stores
- Given that
  - Registers: 32 words (128 Bytes)
  - Memory (DRAM): Billions of bytes (2 GB to 16 GB on laptop)
- How much faster are registers than DRAM??
- About 100-500 times faster!
  - in terms of latency of one access

# Memory Addresses are in Bytes

- Data typically smaller than 32 bits, but rarely smaller than 8 bits (e.g., char type)
  - So everything is a multiple of 8 bits
- Remember, size of word is 4 bytes
- Memory is addressable to individual bytes
- Word addresses are 4 bytes apart
  - words take on the address of their least-significant byte
  - remember to keep words aligned
- RISC-V does not require words to be aligned in memory
  - But it is very ***very bad !!!***
  - So in ***practice***, RISC-V requires integers word-aligned !!!



Least-significant byte in word
15 14 13 12
11 10 9 8
7 6 5 4
3 2 1 0
31 24 23 16 15 8 7 0
Least-significant byte gets the smallest address

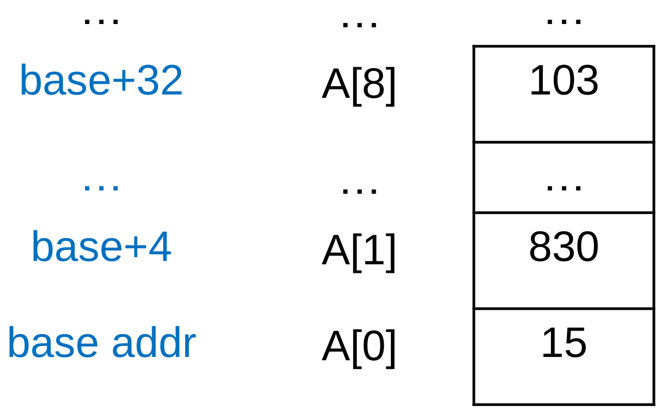
# Transfer from Memory to Register

- C code

```
int A[100];
g = h + A[8];
```

  - Assume: x13 holds base register (pointer to A[0])
  - Note: 32 is offset in bytes
  - Offset must be a constant known at assembly time
- Using Load Word (**lw**) in RISC-V:

```
lw   x10,32(x13)       # reg x10 gets A[8]
add x11,x12,x10        # g = h + A[8]
```



# Transfer from Register to Memory

- C code

```
int A[100];
A[10] = h + A[8];
```

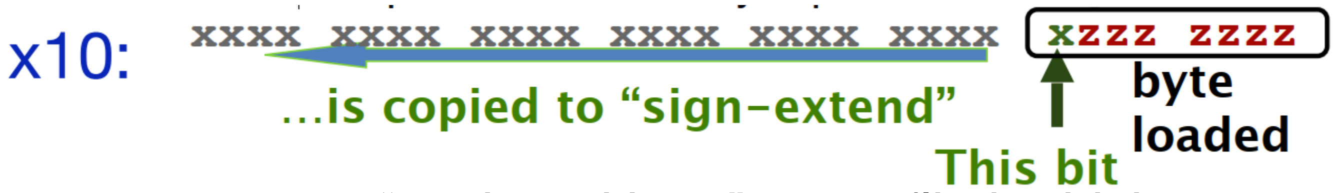
  - Assume: x13 holds base register (pointer to A[0])
  - Note: 32, 40 is offset in bytes
  - Offset must be a constant known at assembly time
- Using Store Word (**sw**) in RISC-V:

```
lw  x10,32(x13)      # reg x10 gets A[8]
add x11,x12,x10      # g = h + A[8]
sw  x11,40(x13)      # A[10] = h + A[3]
```

- x13+32 and x13+40 must be multiples of 4 to maintain alignment

# Loading and Storing Bytes

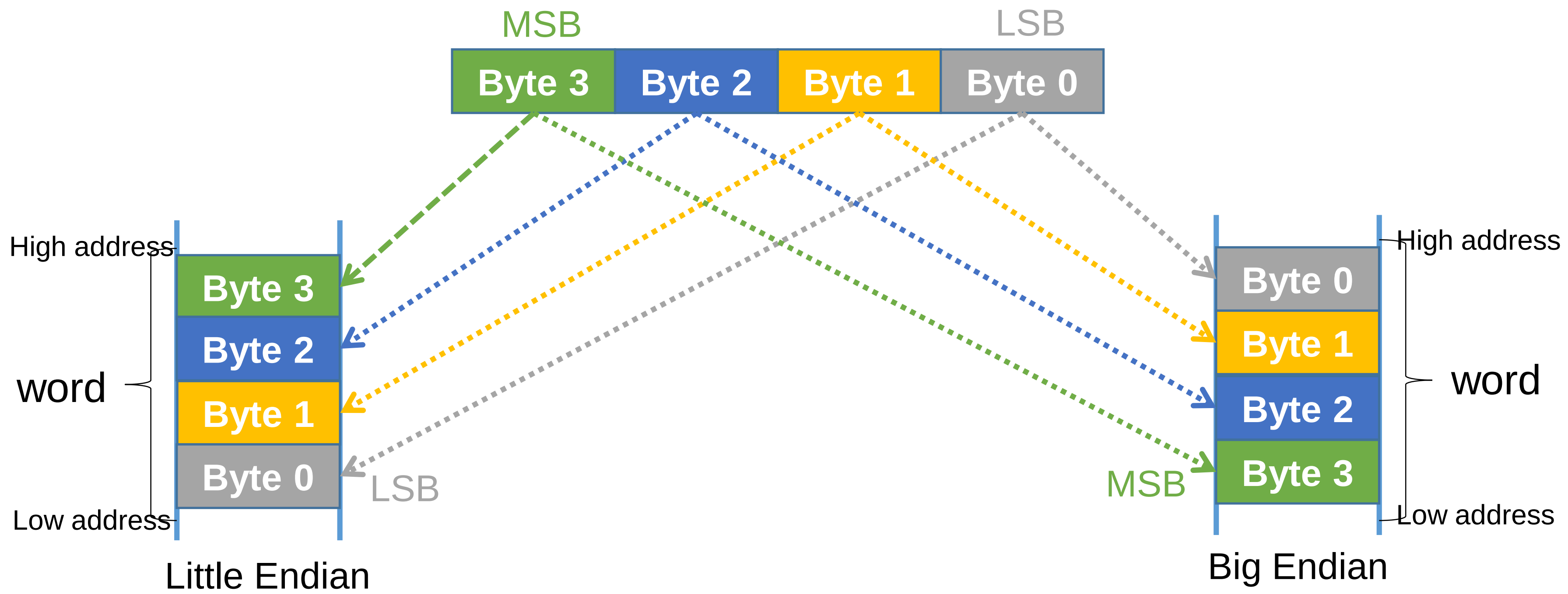
- In addition to word data transfers (lw, sw), RISC-V has byte data transfers:
  - load byte: lb
  - store byte: sb
- Same format as lw, sw
- E.g., `lb x10,3(x11)`
  - contents of memory location (whose address = contents of register x11 + 3), is copied to the low byte position of x10.


x10:
xxxx xxxx xxxx xxxx xxxx xxxx xzzz zzzz
…is copied to "sign-extend"
This bit
byte loaded

  - RISC-V also has "unsigned byte" loads (lbu) which zero extend to fill register.

# Little Endian vs Big Endian

- Endianness: byte ordering within a word
  - Little-endian (e.g. RISC-V)
    - LSB of a word is at least memory address
  - Big-endian (e.g. MIPS)
    - MSB of a word is at least memory address



*LSB is at least address!*

*MSB is at least address!*

# Endianness Example

- Example: For the following RISC-V code, What’s the final value in x12?

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

| Answer | x12 |
|---|---|
| A | 0x5 |
| B | 0xf |
| C | 0x3 |
| D | 0xffffffff |

- for this example
  - byte[0] = 0xf5
  - byte[1] = 0x03
  - byte[2] = 0x00
  - byte[3] = 0x00

## Another Example

- Example: For the following RISC-V code, What's the final value in x12?

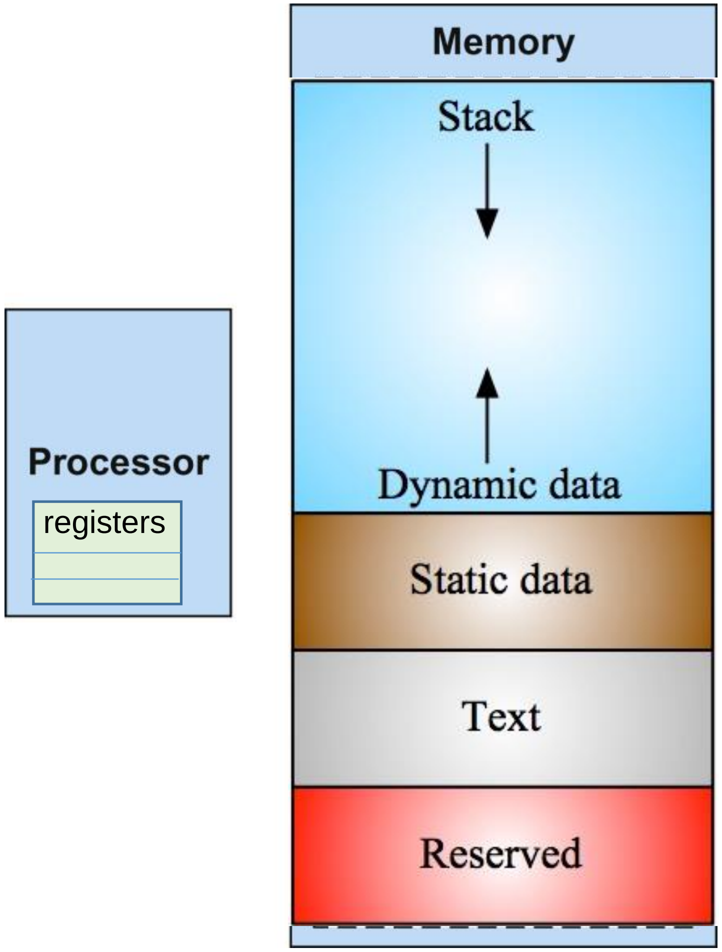
```
addi x11,x0,0x8f5
sw x11,0(x5)
lb x12,1(x5)
```

| Answer | x12 |
|---|---|
| A | 0x8 |
| B | 0xf8 |
| C | 0x5 |
| D | 0xfffffff8 |

- The immediate got sign extended...
  - So 0xfffff8f5 got written
- Then load byte is called
  - So it will load byte[1], which is 0xf8
- But load byte sign extends too...
  - So what gets loaded into the register is 0xfffffff8
- If we did lbu we'd instead get 0xf8

## Memory Layout

- Instructions(programs) are represented in binary, just like data
- Programs are stored in *Text Segment*
- Constants and other static variables are stored in *Static data segment*
- Dynamic data: *Heap*
  - E.g., malloc in C, new in Java
- Automatic data: *Stack*



# Logical Operations

- Useful for extracting and inserting groups of bits in a word

| Operation | C | Java | RISC-V |
|---|---|---|---|
| Shift left logical | << | << | sll |
| Shift right | >> | >> | srl/sra |
| Bitwise AND | & | & | and |
| Bitwise OR | \| | \| | or |
| Bitwise XOR | ~ | ~ | xor |

- Shift left logical
  - Shift left and fill with 0 bits
  - `slli` by *i* bits multiplies by $2^i$
- Shift right logical
  - Shift right and fill with 0 bits
  - `srli` by *i* bits divides by $2^i$ (unsigned only)
- Shift right arithmetic
  - Shift right and fill with sign bits

# Logical vs. Arithmetic Shift

- `slli x2, x1, 4    # reg x2 = reg x1 << 4 bits`

  1001 0010 0011 0100 0101 0110 0111 1000

  0010 0011 0100 0101 0110 0111 1000 0000

- `srli x2, x1, 4    # reg x2 = reg x1 >> 4 bits`

  1001 0010 0011 0100 0101 0110 0111 1000

  0000 1001 0010 0011 0100 0101 0110 0111

- `srai x2, x1, 4`

  1001 0010 0011 0100 0101 0110 0111 1000

  1111 1001 0010 0011 0100 0101 0110 0111

# Logical Instructions

- AND: clear some bits
  - `and x9,x10,x11`

```
      0x35   0 0 1 1 0 1 0 1
AND   0x0F   0 0 0 0 1 1 1 1
      ----   ---------------
      0x05   0 0 0 0 0 1 0 1
```

- OR: set some bits
  - `or x9,x10,x11`

```
      0x04   0 0 0 0 0 1 0 0
OR    0x30   0 0 1 1 0 0 0 0
      ----   ---------------
      0x34   0 0 1 1 0 1 0 0
```

- XOR: toggle some bits
  - `xor x9,x10,x12`

```
      0x44   0 1 0 0 0 1 0 0
XOR   0x06   0 0 0 0 0 1 1 0
      ----   ---------------
      0x34   0 1 0 0 0 0 1 0
```

- How about NOT?
  - Can be implemented with XOR
  - `xori x15,x14,-1`

## Data Transfer with Variable Indexing

- C code

```
int A[100];    /* A[0] address is in x13 */
int i;         /* i in x14 */
...
g = h + A[i]; /* h = x12, g = x11, tmp = x15 */
```

- Using Load Word (lw) in RISC-V with pointer arithmetic:

```
sll x15,x14,2     # Multiply i by 4 for ints
add x15,x15,x13  # A + 4 * i
lw x10,0(x15)
add x11,x12,x10
```

# Conditional Operations

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

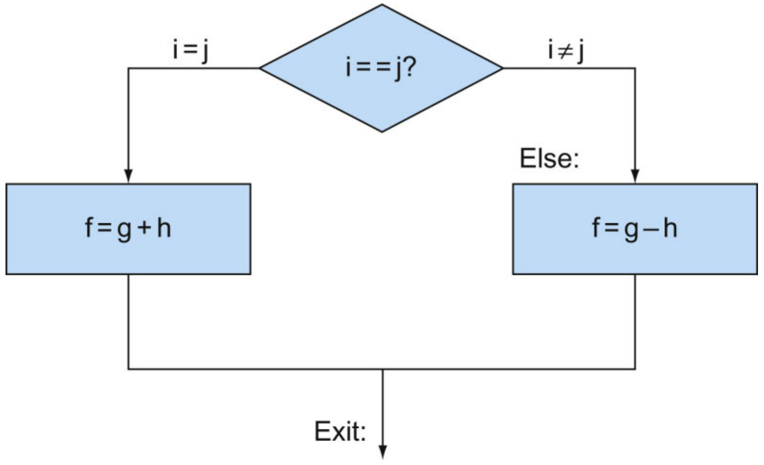
    if (rs1 == rs2) branch to instruction labeled L1;
  - *bne rs1, rs2, L1*
    - if (rs1 != rs2) branch to instruction labeled L1;
- Unconditional branch
  - *beq x0, x0, L1*
    - unconditional jump to instruction labeled L1

# Compiling If Statements

- C code

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

  - i and j are in x22 and x23,
  - f,g and h are in x19, x20 and x21



- Compiled RISC-V code:

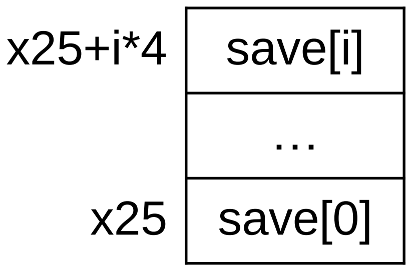
```
      bne x22, x23, Else  # go to Else if i ≠ j
      add x19, x20, x21   # f=g+h, skipped if i ≠ j
      beq x0, x0, Exit    # unconditional go to Exit
Else: sub x19, x20, x21   # f=g-h, skipped if i = j
Exit:
```

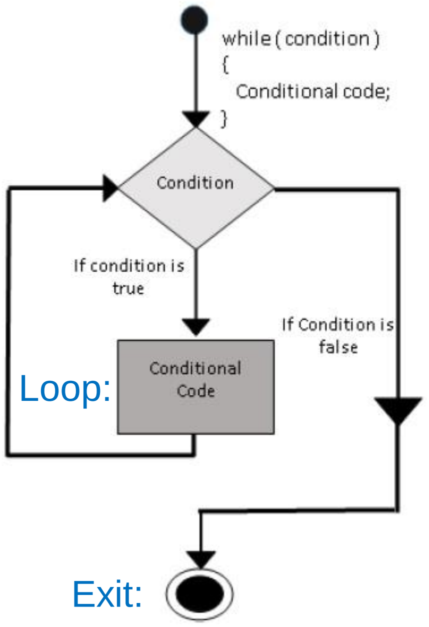
# Compiling Loop Statements

- C code:

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

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





- Compiled MIPS code:

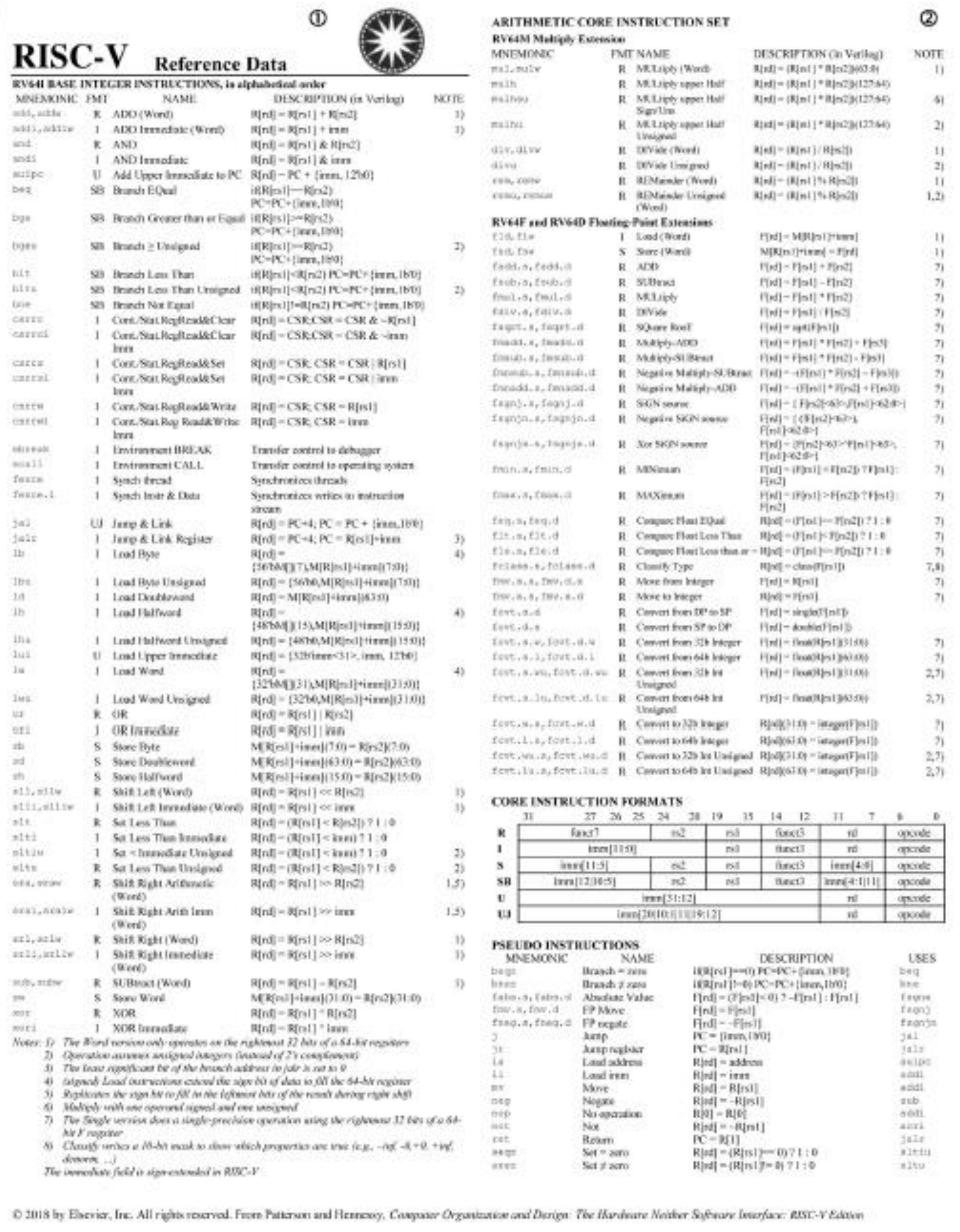
```
Loop: sll   x10, x22, 2    # Temp reg x10 = i * 4
      add   x10, x10, x25  # x10 = address of save[i]
      lw    x9, 0(x10)     # Temp reg x9 = save[i]
      bne   x9, x24, Exit  # go to Exit if save[i]≠k
      addi  x22, x22, 1    # i = i + 1
      j     Loop           # go to Loop
Exit:
```

## Summary

1. Instruction set architecture (ISA) specifies the set of commands (instructions) a computer can execute
2. Hardware registers provide a few very fast variables for instructions to operate on
3. RISC-V ISA requires software to break complex operations into a string of simple instructions, but enables faster, simple hardware
4. Assembly code is human-readable version of computer’s native machine code, converted to binary by an assembler

# RISC-V Reference Card

- In textbook
- Available on Blackboard



①

# RISC-V Reference Data

**RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order**

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

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

②

**ARITHMETIC CORE INSTRUCTION SET**

**RV64M Multiply Extension**

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

**RV64F and RV64D Floating-Point Extensions**

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

**CORE INSTRUCTION FORMATS**

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

**PSEUDO INSTRUCTIONS**

| MNEMONIC | NAME | DESCRIPTION | USES |
|---|---|---|---|
| beqz | Branch = zero | if(R[rs1]==0) PC=PC+{imm,1b'0} | beq |
| bnez | Branch ≠ zero | if(R[rs1]!=0) PC=PC+{imm,1b'0} | bne |
| fabs.s,fabs.d | Absolute Value | F[rd] = (F[rs1]<0) ? -F[rs1] : F[rs1] | fsgnx |
| fmv.s,fmv.d | FP Move | F[rd] = F[rs1] | fsgnj |
| fneg.s,fneg.d | FP negate | F[rd] = -F[rs1] | fsgnjn |
| j | Jump | PC = {imm,1b'0} | jal |
| jr | Jump register | PC = R[rs1] | jalr |
| la | Load address | R[rd] = address | auipc |
| li | Load imm | R[rd] = imm | addi |
| mv | Move | R[rd] = R[rs1] | addi |
| neg | Negate | R[rd] = -R[rs1] | sub |
| nop | No operation | R[0] = R[0] | addi |
| not | Not | R[rd] = ~R[rs1] | xori |
| ret | Return | PC = R[1] | jalr |
| seqz | Set = zero | R[rd] = (R[rs1]== 0) ? 1 : 0 | sltiu |
| snez | Set ≠ zero | R[rd] = (R[rs1]!= 0) ? 1 : 0 | sltu |