COMPUTER ORGANIZATION

Lecture 2
RISC-V Introduction

2025 Spring

baiyh@sustech.edu.cn

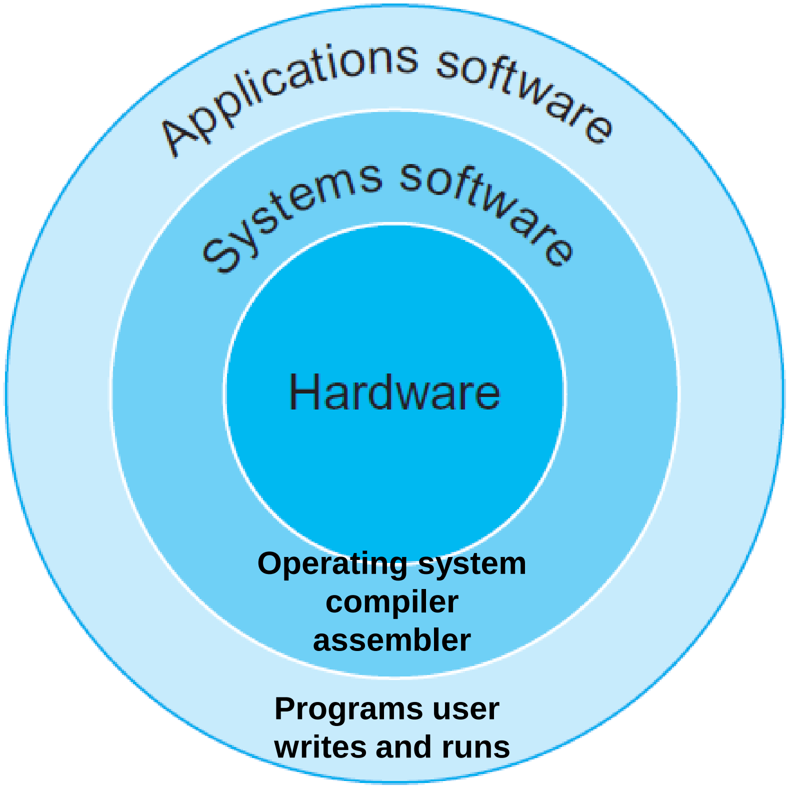
Recap

- 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)
        - Memory
        - Input
        - Output
      - Instruction set architecture (ISA)

$\left.\begin{array}{l}\text{Control}\\ \text{Datapath(ALU)}\end{array}\right\}$ CPU

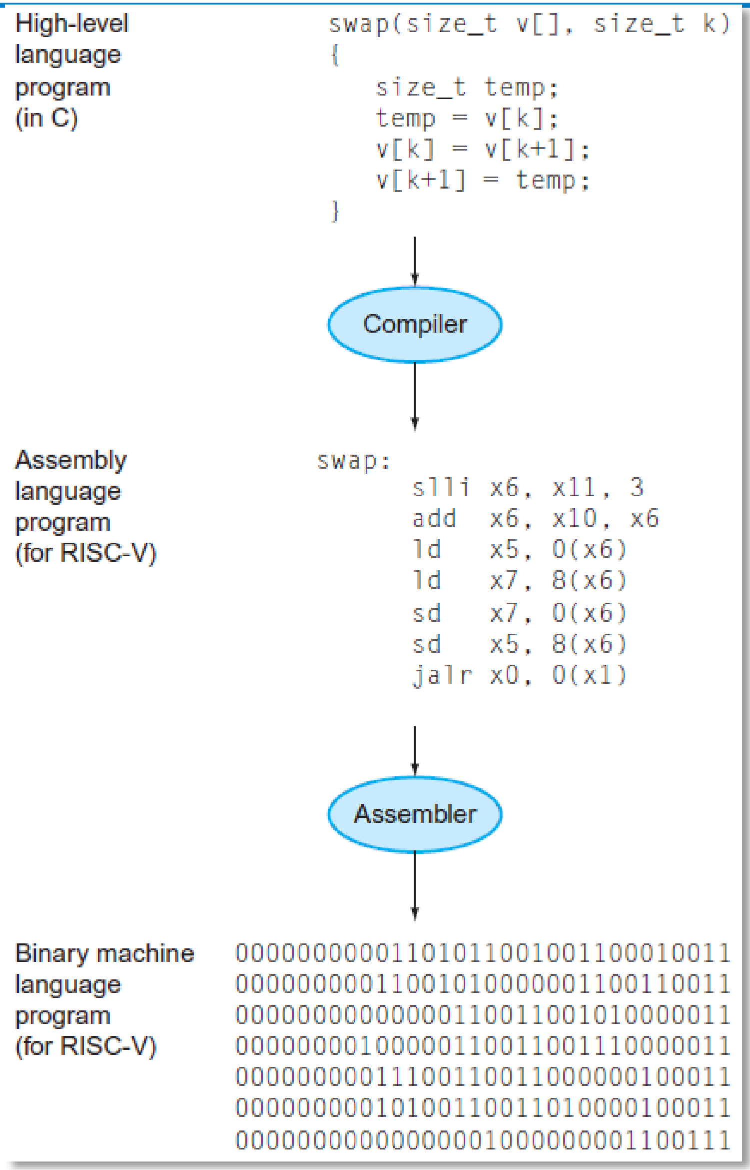
The Concept of a Computer

- Application software
  - Written in high-level language
- System software
  - Compiler: translates HLL code to machine code
- Operating System:
  - Handling input/output
  - Managing memory and storage
  - Scheduling tasks & sharing resources
- Hardware
  - Processor, memory, I/O controllers



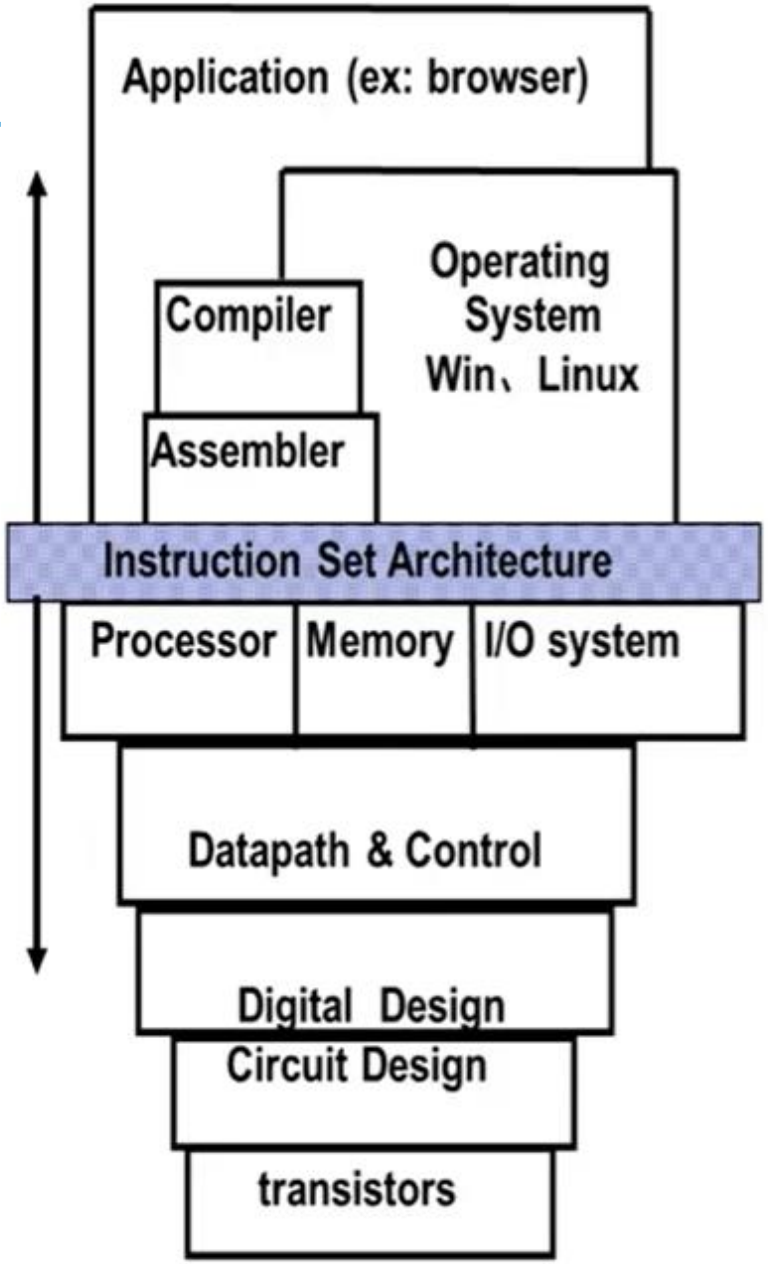
Levels of Program Code

- C program compiled into assembly language and then assembled into binary machine language.
- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
- Machine language
  - Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data



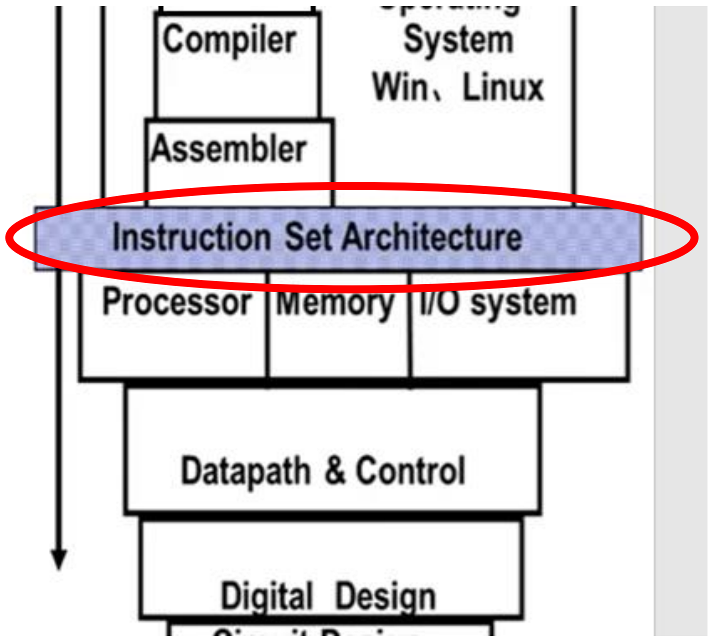
Abstractions

- Abstraction helps us deal with complexity
  - Hides lower-level details
- Instruction Set Architecture (ISA) or Computer Architecture
  - The hardware/software interface
  - Includes instructions, registers, memory access, I/O, and so on
- Operating system hides details of doing I/O, allocating memory from programmers



Instruction Set Architecture (ISA)

- A set of assembly language instructions (ISA) provides a link between software and hardware.
- Given an instruction set, software programmers and hardware engineers work more or less independently.
- Common types of ISA: RISC, CISC
- Examples:
  - IBM370/X86 (CISC)
  - RISC-V (RISC)
  - MIPS (RISC)
  - ARM (RISC)



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`

is assembly comment syntax

Destination register

Second operand register

Operation code (opcode)

First operand register

- 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
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- 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 → 0, 0 → 1

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

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

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

Immediates & Sign Extension

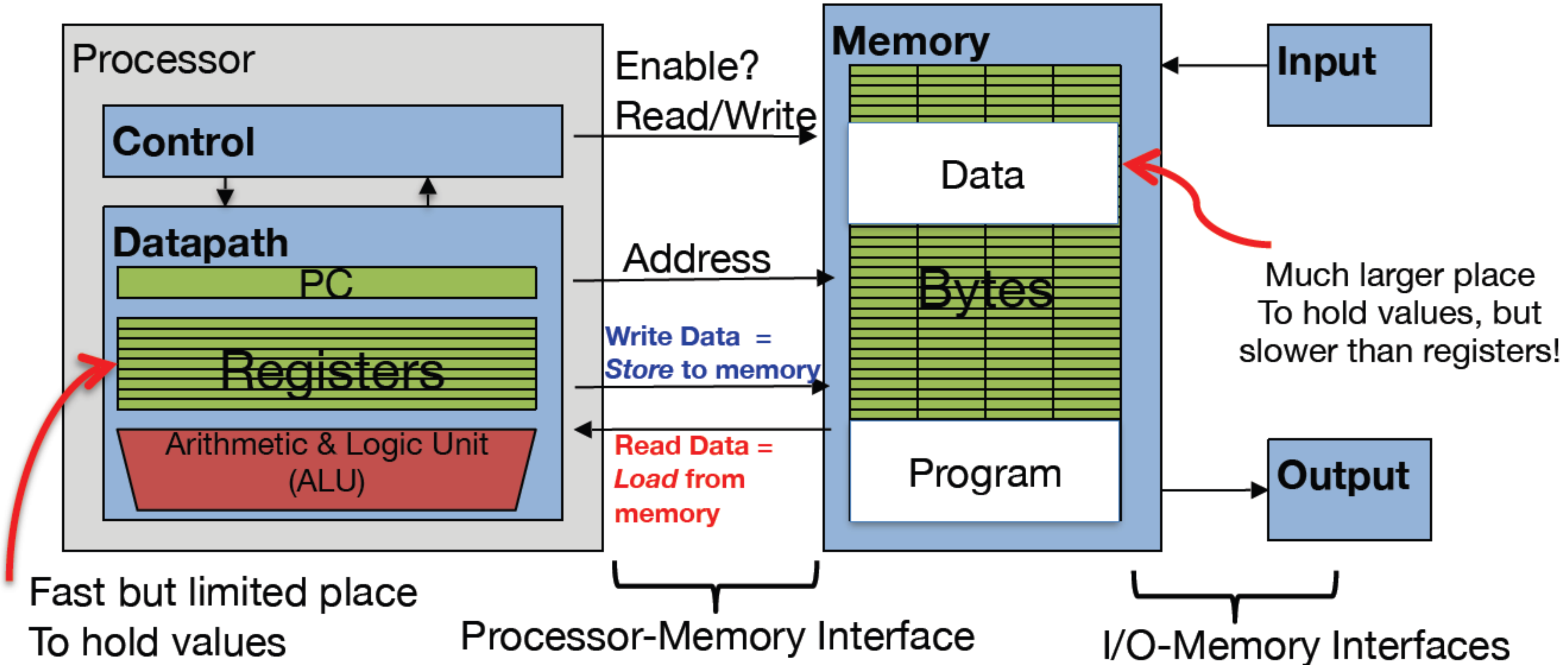
- Immediates are necessarily small
  - An I-type instruction can only have 12 bits of immediate (We'll see more details in future lecture)
- In RISC-V immediates are "sign extended"
  - So the upper bits are the same as the top bit
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010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

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

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

Data Transfer Operations

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

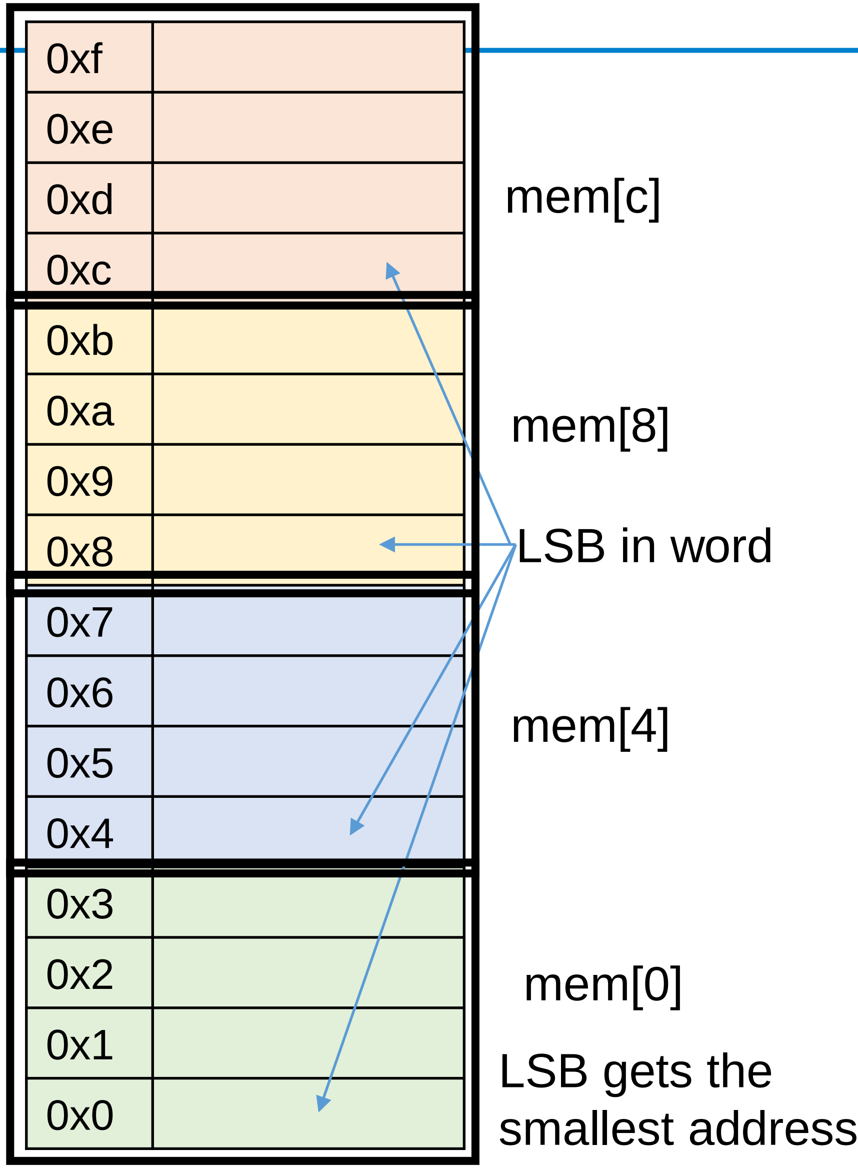


Speed of Registers vs. Memory

- 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 !!!



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]
```

…	…	…
base+32	A[8]	103
…	…	…
base+4	A[1]	830
base addr	A[0]	15

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:

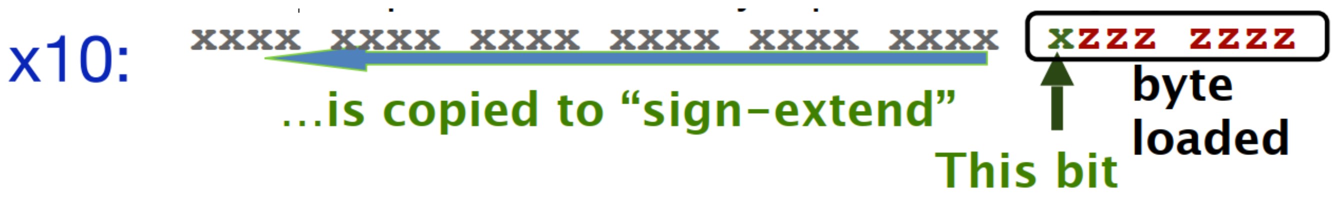
```
lw  x10,32(x13)      # reg x10 gets A[8]
add x11,x12,x10      # g = h + A[8]
sw  x11,40(x13)      # A[10] = h + A[8]
```

- 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.

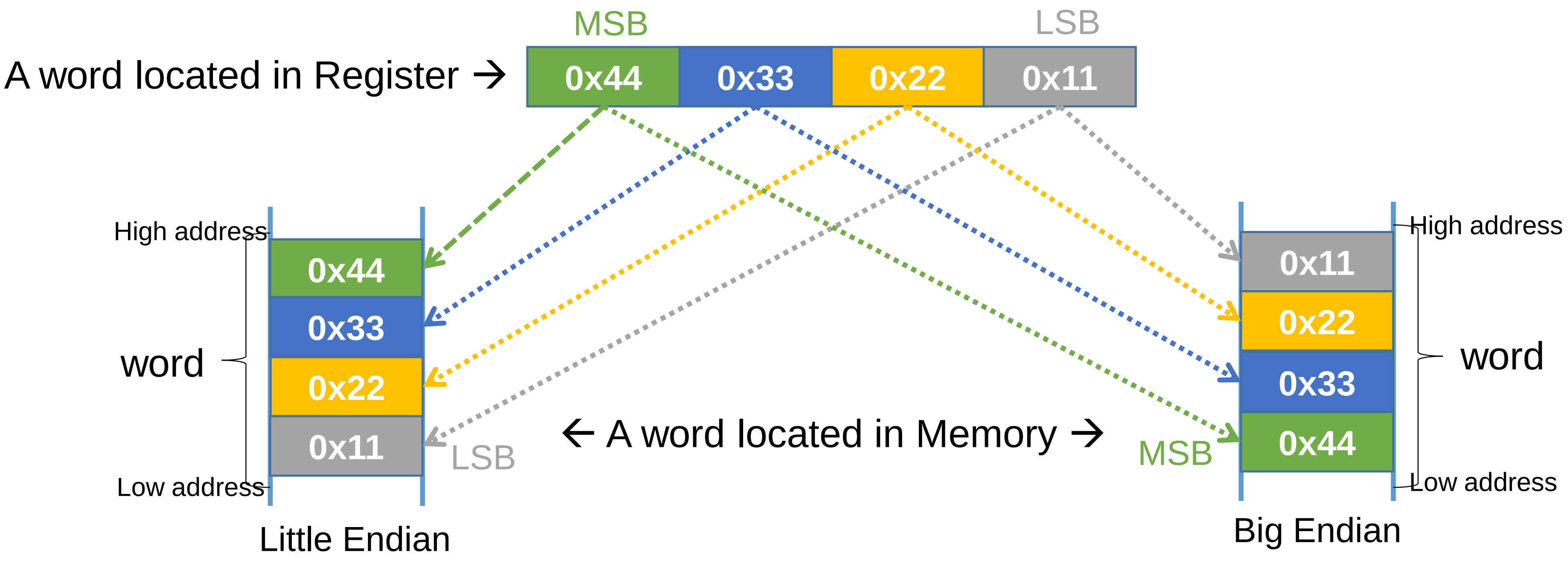
  - RISC-V also



egister.

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?

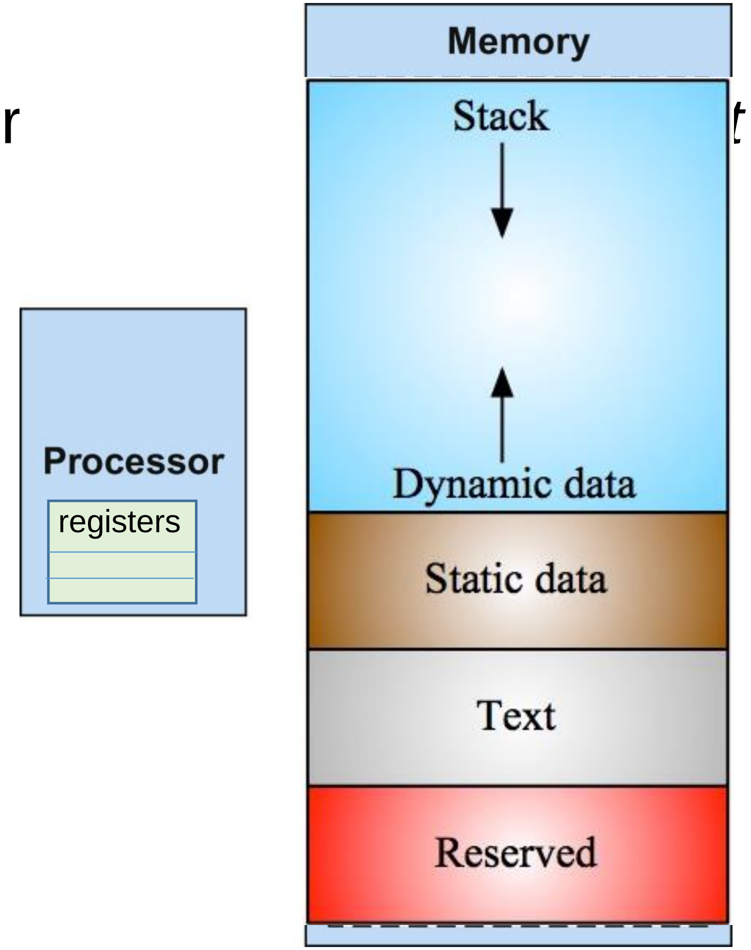
```
addi x11,x0,-512
sw x11,0(x5)
lb x12,1(x5)
```

Answer	x12
A	0xe
B	0xfe
C	0x0
D	0xfffffffe

- The immediate got sign extended...
  - So $-512_{10} = -2^9$ = 0xfffffe00 got written
- Then load byte is called
  - So it will load byte[1], which is 0xfe
- But load byte sign extends too...
  - So what gets loaded into the register is 0xfffffffe
- If we did lbu we'd instead get 0xfe

Memory Layout

- Instructions(programs) are represented in binary, just like data
- Programs are stored in *Text Segment*
- Constants and other static variables are stor *t*
- 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:

```
slli 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*

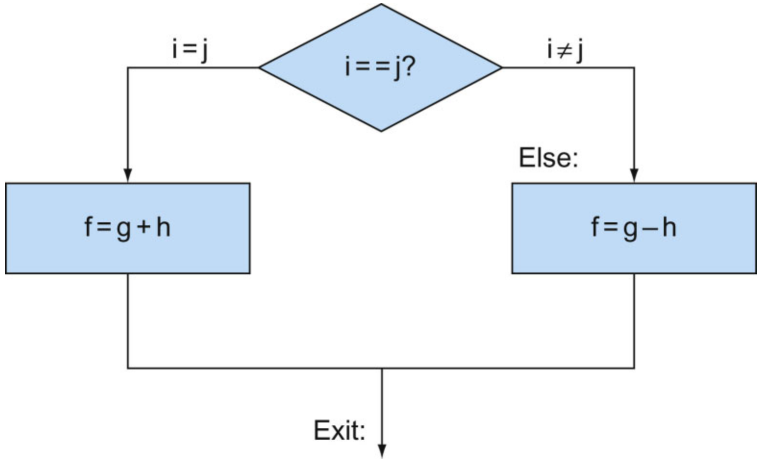
    if (rs1 == rs2) branch to instruction labeled L1;
  - *bne rs1, rs2, L1*
    - if (rs1 != rs2) branch to instruction labeled L1;
- Unconditional branch
  - *beq x0, x0, L1*
    - unconditional jump to instruction labeled L1

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:
```

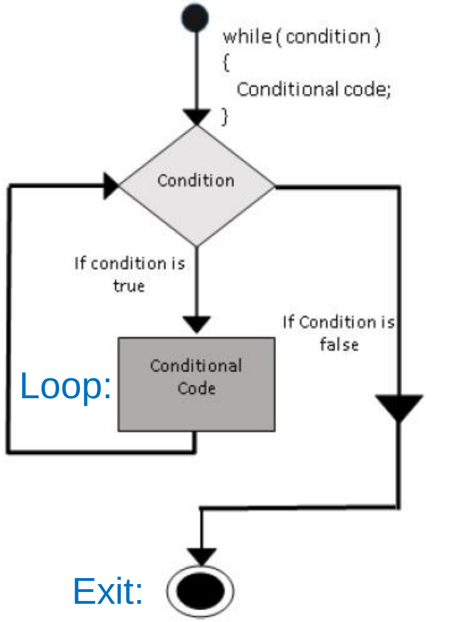
Compiling Loop Statements

- C code:

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

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

x25+i*4	save[i]
	…
x25	save[0]



- Compiled RISC-V code:

```
Loop: slli   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:
```

baiyh@sustech.edu.cn

Summary of Design Principles

1: Simplicity favors regularity
- Keep all instructions the same size.

2: Smaller is faster
- Register vs memory
- Number of registers is small

3: Make the common case fast
- Immediate operand

4: Good design demands good compromises
- Keep formats as similar as possible

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 (RV64)
- Or on Blackboard (RV32)

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ˆ rs2
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 « rs2
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 » rs2
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 » rs2 (Arith*)
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ˆ imm
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm
slli	Shift Left Logical Imm	I	0010011	0x1	imm[11:5]=0x00	rd = rs1 « imm[4:0]
srli	Shift Right Logical Imm	I	0010011	0x5	imm[11:5]=0x00	rd = rs1 » imm[4:0]
srai	Shift Right Arith Imm	I	0010011	0x5	imm[11:5]=0x20	rd = rs1 » imm[4:0] (Arith*)
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0
lb	Load Byte	I	0000011	0x0		rd = {24'bM[rs1+imm][7],M[rs1+imm][7:0]}
lh	Load Half	I	0000011	0x1		rd = {16'bM[rs1+imm][15],M[rs1+imm][15:0]}
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][31:0]
lbu	Load Byte (U)	I	0000011	0x4		rd = {24'b0,M[rs1+imm][7:0]}
lhu	Load Half (U)	I	0000011	0x5		rd = {16'b0,M[rs1+imm][15:0]}
sb	Store Byte	S	0100011	0x0		M[rs1+imm][7:0] = rs2[7:0]
sh	Store Half	S	0100011	0x1		M[rs1+imm][15:0] = rs2[15:0]
sw	Store Word	S	0100011	0x2		M[rs1+imm][31:0] = rs2[31:0]
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC = PC + {imm,1'b0}
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC = PC + {imm,1'b0}
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC = PC + {imm,1'b0}
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC = PC + {imm,1'b0}
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC = PC + {imm,1'b0}
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC = PC + {imm,1'b0}
jal	Jump And Link	J	1101111			rd = PC+4; PC = PC + {imm,1'b0}
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm
lui	Load Upper Imm	U	0110111			rd = imm « 12
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm « 12)
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger